Datenbanken
und
Algorithmen

Inhaltsverzeichnis

1. Allgemeines Datenbankmodel

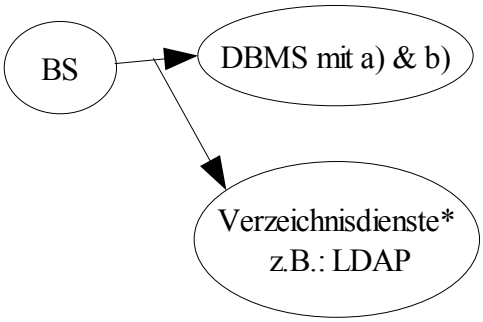
Zunächst werden einige Grundbegriffe erklärt:

Datenbank (DB): Menge aller gespeicherter Daten des Datenbanksystems
Datenbankmanagementsystem (DBMS): Menge der Software zur Verwaltung der in einem DB-System gespeicherten Daten

Datenbanksystem (DBS): Datenbank + Datenbankmanagementsystem

Gegeben: Rechner mit Betriebssystem (BS), insbesondere mit einem Dateisystem und Diensten, die auf dem Dateisystem operieren (z.B.: Dienste für das Anlegen, Löschen, Umbenennen von Dateien; Dienste auf Verzeichnisse: ls, rmdir, ...)

Was fehlt?:
a) Programmunabhängigkeit der Datei (d.h.: es fehlt ein Dienst, der die gespeicherten Daten mit der Beschreibung ihres Datensatzaufbaus verbindet)
b) Software für den Direktzugriff auf einzelne gespeicherte Datensätze



* = mit Unterstützung von a)

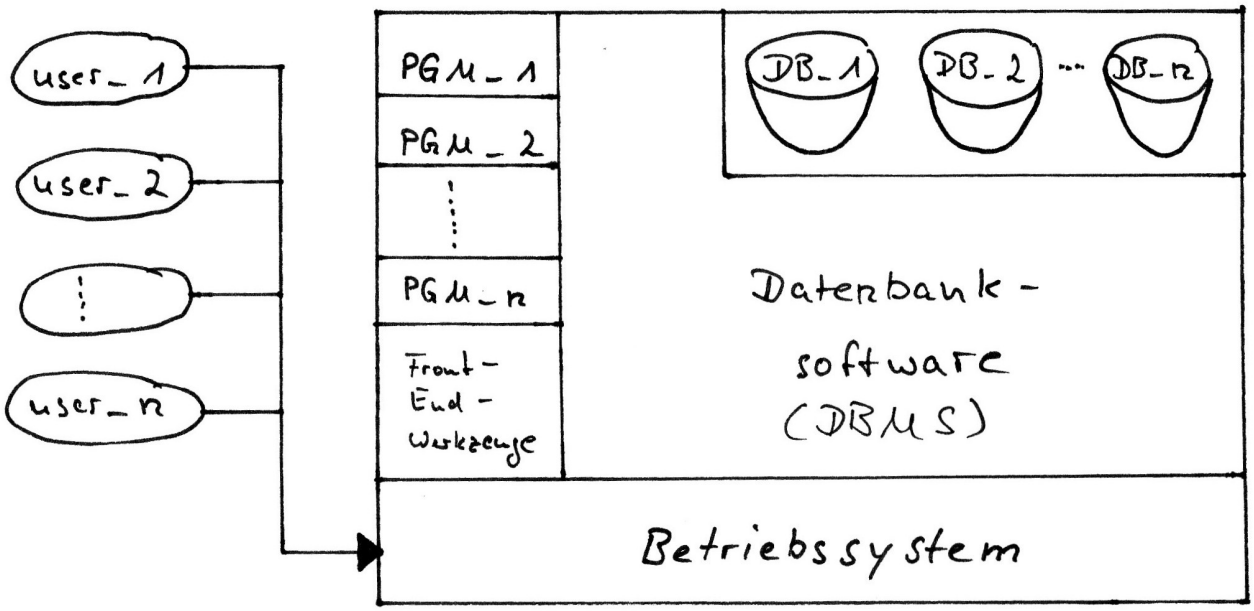


Abbildung 1: Allgemeiner Aufbau eines DBS

Anm.: Eine Datenbank DB_i ($1 \leq i \leq m$) besteht aus einer oder mehreren Datenbanksegmenten

bzw. Tabellen. Vergleicht man die DB-Datenorganisation mit dem BS-Dateisystem, dann entspricht eine Tabelle einer Datei.

Anforderungen an die Funktionalität eines DBMS

1. Persistenz: dauerhafte und strukturtreue Speicherung von Datenbeständen (d.h.: insbesondere müs sen auch komplexe Datentypen ohne Programmieraufwand gespeichert und gelesen werden können).
   Bsp.: class Artikel { String artname;double preis; LinkedList zutaten; }

   Ohne DBMS müssen Instanzen dieser Klasse, wenn man sie in eine Datei schreibt bzw. sie ausliest, durch einen besonderen Programmieraufwand verwaltet werden. Bei einem DBS genügt es, diesen Datentyp in einen Datentyp zu transformieren, der im Data-Dictionary des DBMS eingetragen wird (s. 3)

3. Sekundärspeicherverwaltung: Direktzugriffe werden implementiert, d.h. Lese-Operationen auf der DB sollen mit wenigen Zugriffen auf Sekundärspeicher (z.B.: Harddisk) den/die gewünschten Datensätze zurückgeben.

Anm.: Um beim Lesen Direktzugriffe ausführen zu können, ist beim Schreiben ein erhöhter Programmieraufwand erforderlich.

Bsp.: Direktzugriff mit binärer Suche

Gegeben: Eine Datei mit Personendaten und eine Indexdatei, die die Offsets der Datensätze der Personendatei verwaltet.

PERS.DAT

Offset	PNR	Name
0	137	Müller
100	107	Schulze
170	244	Huber
230	224	Meier

PERS.IDX (sortiert nach Ortnungsbegriff PNR)

PNR	Offset
107	100
137	0
224	230
244	170

Suche: Person mit PNR = 224
=> 2 Vergleichsschritte
=> 1 Direktzugriff auf Nutzdatensatz (hier Offset = 170)

Aufwand um neuen Datensatz einzufügen:
=> 1 Datensatz anhängen (Datensatz: 199/Huber wird bei OFFSET 440 eingefügt)
=> Indexdatensatz einfügen (Das Paar (199/440) in PERS.IDX)
=> Indexdatei sortieren (~ n^2 Operationen)

Strategien von DBMS, um den Sortieraufwand beim Einfügen zu minimieren:

a) die Indexdatei wird während einer Arbeitssitzung komplett in den RAM geladen. => der Aufwand von n^2 Operationen findet als RAM-Operation statt.

b) die neuen Index-Sätze werden während einer Arbeitssitzung in eine LOG-Datei gesammelt, die sortiert wird (Aufwand k^2 ; $k < n$). Am Ende der Arbeitssitzung wird die sortierte LOG-Datei im Reißverschlussverfahren zusammen geführt (Aufwand n)

3. Verwaltung eines Schema-Katalogs (Data-Dictionary)

Der Schema-Katalog enthält sämtliche Metadaten einer DB. Für alle Tabellen werden dort der Tabellenaufbau, d.h., die Datentypen aller Spalten, die Verknüpfungen der Tabellen untereinander und Einschränkungen der Wertebereiche der Spalten hinterlegt.

Bsp.: Die Instanzen der Klasse Artikel (s.o.) sollen in einer DB gespeichert werden (hier mit relationalen Datenmodell). Jeder Knoten der Zutaten-Liste soll von folgendem Datentyp sein:

```
class Zutaten{
        String zu_bez;
        double quant;  /* Menge der Zutaten  */
        String einheit;  /* Maßeinheit der Zutat  */
}
```

Eintragen ins Data-Dictionary

E1) DB-Name = ARTIKELDB
E2) Tab.-Name1 = ARTTAB
E3) für jedes Attribut einer Artikel-Instanz einen Eintrag machen:
- 3.1)Attribut: artname Datentyp: char(50) zusätzl. Bedingungen: Ø
- 3.2)Attribut: preis Datentyp: int zusätzl. Bedingungen: Ø
- 3.3) Attribut: art_id Datentyp: int zusätzl. Bedingungen: Eindeutigkeit

Anm.: Unter der Menge der Attribute mit eindeutiger Wertefolge innerhalb einer Tabelle wird ein Attribut als eindeutiger Identifikator der Tabellenzeilen ausgewählt. Dieser ist das Primärschlüsselattribut (primary key = PRIK).

E4) Tabellenname 2 = Zutatab
E5) Attribut-Einträge für Zutaten (ZUTATAB)
- 5.1)Attribut: zubez Datentyp: char(100) zusätzl. Bedingungen: Ø
- 5.2)Attribut: quant Datentyp: double zusätzl. Bedingungen: Ø
- 5.3)Attribut: einheit Datentyp: char(15) zusätzl. Bedingungen: Ø
- 5.4)Attribut: art_id Datentyp: int zusätzl. Bedingungen: FKEY (-> Foreign Key)
- 5.5)Attribut: zutat_id Datentyp: int zusätzl. Bedingungen: PRIK

Anm.: Ein FKEY-Wert muss bereits als PRIK-Wert in einer anderen Tabelle gegeben sein. Z.B.: Ein Wert für art_id, der in ZUTATAB eingetragen werden soll, muss vorher bereits als PRIK-Wert in ARTTAB enthalten sein.

4. Interpretation einer Anfragesprache

Um auf Daten einer DB lesend oder schreibend zugreifen zu können, stellt das DBMS eine Anfragesprache zur Verfügung. Das DBMS enthält einen Interpreter für Kommandos der Anfragesprache. Die Anfragesprache ist abhängig vom Datenmodell des DBMS. (Datenmodell = Gesamtheit der möglichen Dateitypen, die durch das Data-Dictionary des DBMS implementiert werden können.)

Bsp.:
- relationales Datenmodell => Anfragesprache: SQL <*Structure Querry Language*>
- hierarchisches Datenmodell => Anfragesprache: COBLDLI <*Data Language Interface*>

- objektorientiertes Datenmodell => Anfragesprache: OQL <*Object Querry Language*>

Anm.: für Anfragesprachen gibt es Einbettungen in höhere Programmiersprachen:

a) für SQL: ESQL/C (in C)
ODBC (in C++) <*Open Database Connectivity*>
JDBC (in Java) <*Java to Database Connectivity*> (vgl. Package: java.sql)
b) für OQL: POET/OQL
[ONTOS/OQL,.....]

5. Integrität

Integrität bedeutet, dass schreibende Operationen eine DB von einem korrektem Zustand der Daten nur in einen neuen korrekten Zustand überführen dürfen. Um die Integrität zu sichern, gibt es im Data-Dictionary die Möglichkeit, Integritätsbedingungen festzulegen: Man unterscheidet drei Arten von Integritätsbedingungen:

(1)Typintegrität: Bei den schreibenden Operationen (Insert, Update, Delete) dürfen in ein Attribut a mit Datentyp A nur Werte vom Typ A geschrieben werden.

(2)Referentielle Integrität: Datensätze (z.B.: Tabellenzeilen) aus verschiedenen Segmenten einer DB (z.B.: Tabellen) können durch Fremdschlüsselbedingungen (FKEY) aufeinander Bezug nehmen. Referentielle Integrität verlangt, dass bei schreibenden Operationen die Fremdschlüsselbeziehungen nicht zerstört werden.
Bsp.: T1: Artikeltab <---------- T2: Zutatentab
FEKY = art_id

Beispiel für Kontrolle der referentiellen Integrität:
(1)kein DELETE auf T1 für art_id = xxx, falls xxx als FKEY-Wert in T2-Zeilen vorkommt.
(2)kein INSERT auf T2 für art_id = zzz, falls zzz nicht als PRIK-Wert in T1-Zeilen vorkommt.

(3)Wertintegrität: kontrolliert anwendungsbezogene Einschränkungen des Wertebereichs des Datentyps. Bsp.: das Attribut PLZ ist mit Datentyp = int ins Data-Dictionary der DB eingetragen.

Wertebereich $W_{int} = \{z \in R \mid -2^{31} \leq z \leq 2^{31}-1\}$ ($2^{31} \approx 2{,}1$ Milliarden)
Integritätsbedingung: $1000 \leq z \leq 99999$
=> $W_{PLZ} = \{ W_{int} \mid 1000 \leq z \leq 99999\}$
Es gilt $W_{PLZ} \subset W_{int}$

Allgemein: Standard-Eintrag ins Data-Dictionary ist ein Tripel:
(A, dt(A), IBed(A)) =>A: Attribut; dt(A): Datentyp von A; IBed(A): Integritätsbedingung von A

Falls IBed(A) nicht leer ist => IBed(A) schränkt den Wertebereich von $W_{dt(A)}$ ein. D.h. $W_a \subset W_{dt(A)}$
Deshalb ist äquivalent (A, dt(A), Ibed(A)) <=> (A, dt(A), W_a)

Anm.: Längenbedingungen, wie z.B.: n beim da(A) = char(n) oder p,q beim dt(A) = decimal(p,q) gehören zur Datentypangabe dt(A).

Bsp.:
a) (A, dt(A), IBed(A)) = (name, char(30), Ø) = (name, char(30), {})
b) (preis1, float, $\{p \in W_{float} \mid p \geq 0\}$)

c) (steuernachzahlung, decimal(13,2), Ø) = (steuernachzahlung, decimal(13,2), $W_{decimal(13,2)}$)

6. Verwalten von Benutzersichten

Jede DB hat in der Regel verschiedene Benutzergruppen. Bestimmte Benutzergruppen sollen auf bestimmte Attribute lesenden und schreibenden XOR nur lesenden XOR keinen Zugriff haben. Weiterhin kann für bestimmte Benutzergruppen festgelegt sein, dass sie nur verdichtete Attributwerte lesen können. Diese unterschiedlichen Formen des Zugriffs nennt man Benutzersicht.

Anm.: Das atomare Element der Zuordnung: Benutzersicht → Daten
- ist im Dateisystem eines BS die Datei
- ist im DBMS das Attribut

7. Datenschutz

Datenschutz: Schutz der DB vor Zugriff unbefugter Dritter. D.h. das DBMS muss über Sicherungsmechanismen, wie Authentifizierung usw. verfügen.

8. Transaktionsverwaltung

Eine Transaktion ist eine Folge von (schreibenden) DB-Zugriffen, die zu einer Gruppe zusammen gefasst werden. Für diese Gruppe gilt: Entweder werden alle Zugriffe ausgeführt oder es wird kein Zugriff ausgeführt.
Bsp.: eine Transaktion T besteht aus Löschoperationen auf die Zutaten-Tabelle und Artikel-Tabelle:

T = [dz_1, dz_2, ..., dz_N, da]

dz_i: DELETE auf eine Zutatenzeile ($1 \leq i \leq N$)
da: DELETE auf eine Artikelzeile

Die Realisierung der Transaktionsverwaltung verlangt, dass alle Schreib-Operationen in einer LOG-Datei notiert werden, mit der im Falle eines fehlerhaften Transaktionsendes die DB in ihren ursprünglichen Zustand zurückversetzt werden kann.

9. Synchronisation

An das DBMS besteht die Anforderung, zeitlich konkurrierende Zugriffe (Delete, Insert, Update, Select) auf einen Datensatz zu steuern. Für diese Steuerung müssen Regelmechanismen eingerichtet sein (z.B.: „Schreiben geht vor Lesen“). Diese Mechanismen werden zusammen mit den Methoden der Sekundärspeicherverwaltung implementiert.

10. Datensicherung (Recovery)

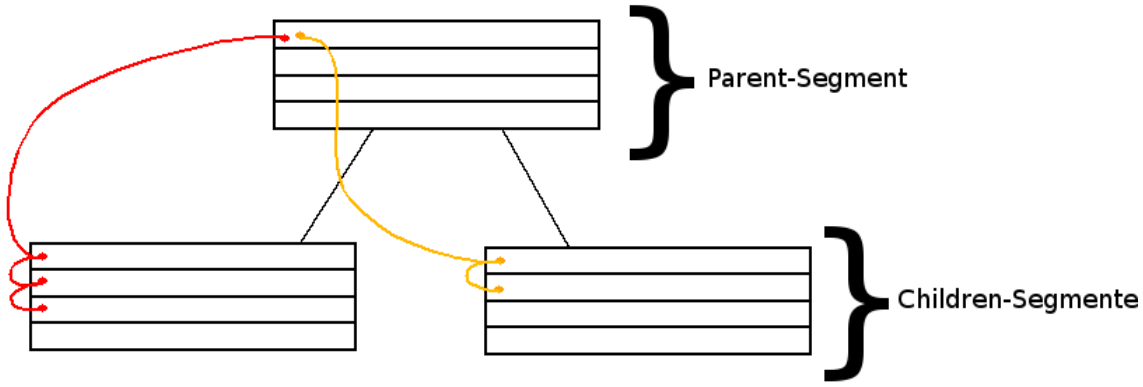
In festgelegten Arbeitsperioden (Arbeitstagen, Stunden, Minuten) werden bestimmte oder alle Segmente einer DB auf schnellschreibenden Sekundärspeichermedien gesichert. Dafür verfügt das DBMS über Import- und Exportmechanismen. Die Datensicherung ist die Grundlage für die Recoveryfähigkeit des DBMS nach Systemausfall.

Anm.: Die Import- / Exportmechanismen sollen insbesondere die strukturtreue Speicherung in sequentiellen Dateien unterstützen.

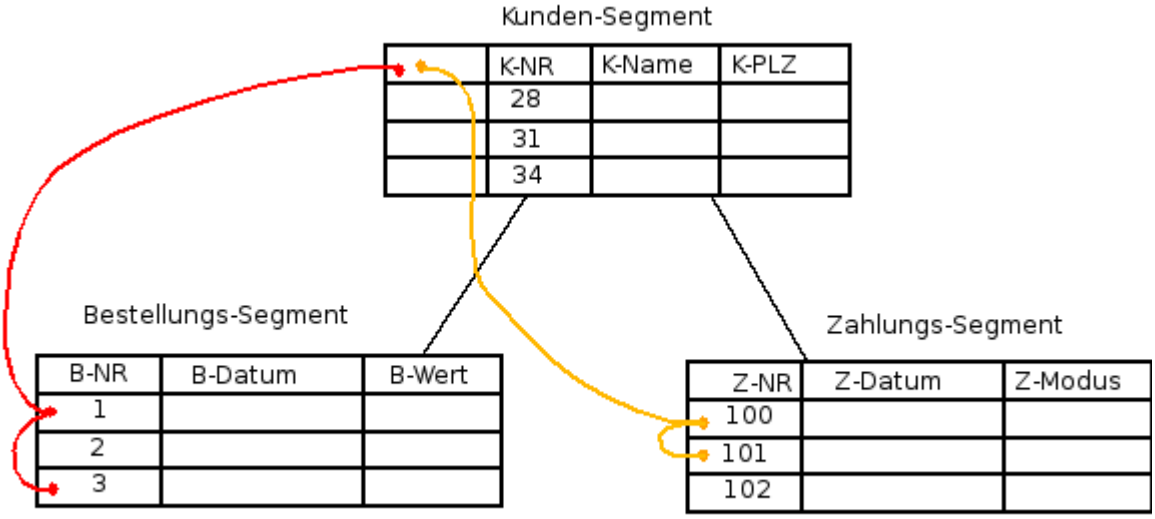
2. Spezielle Datenbankmodelle

2.1. Hierarchische Datenbankmodelle

Hierarchien können durch Baumgraphen beschrieben werden.
Datensätze einer hierarchischen Datenbank (HDB) sind in Segmenten organisiert (1 Segment ≡ 1 Tabelle). Die Segmente stehen in einer Baumgraph-artigen Anordnung:



Beispiel: Konten-Datenbank



Das hierarchische Datenmodell bestimmt die Ausgestaltung der Anfragesprache sowohl in Hinsicht auf die schreibenden als auch auf die lesenden Operationen: (Beispiel: Sprachklausuel aus der Anfragesprache DLI für das HDB=IMS)

BSP1: Einfügeoperation: ISRT : notwendige Parameter: (DS, E_Seg [, P_Seg, PRIK_Parent])

mit:

DS = einzufügender Datensatz
E_Seg = Name des Segments, in das der DS eingefügt werden soll
P_Seg = Name des Parentsegmentes
PRIK_Parent = PRIK-Wert des übergeordneten Parent-Datensatzes

BSP2:

Leseoperationen:

a) Direktzugriff auf einen Datensatz in einem Segment (GetUnique)
   Parameter: Segmentname, PRIK-Segment
b) Sequentielles Weiterlesen im Segment nach einem erfolgreichen GetUnique (GetNext)
   Parameter: Segmentname
c) Lesen aller Datensätze in einem Children-Segment, die einem Parentdatensatz untergeordnert sind (GetNext within Parent)
   Parameter: PRIK_Parent, Ch_Segmentname

Anmerkung: Der Aufbau des Datenbank-Modells hat Folgen für die Syntax der Anfragesprache

Anmerkung: Die hierarchische Anordnung der Datensätze in einer HDB wird durch Referenzen (Pointer) implementiert

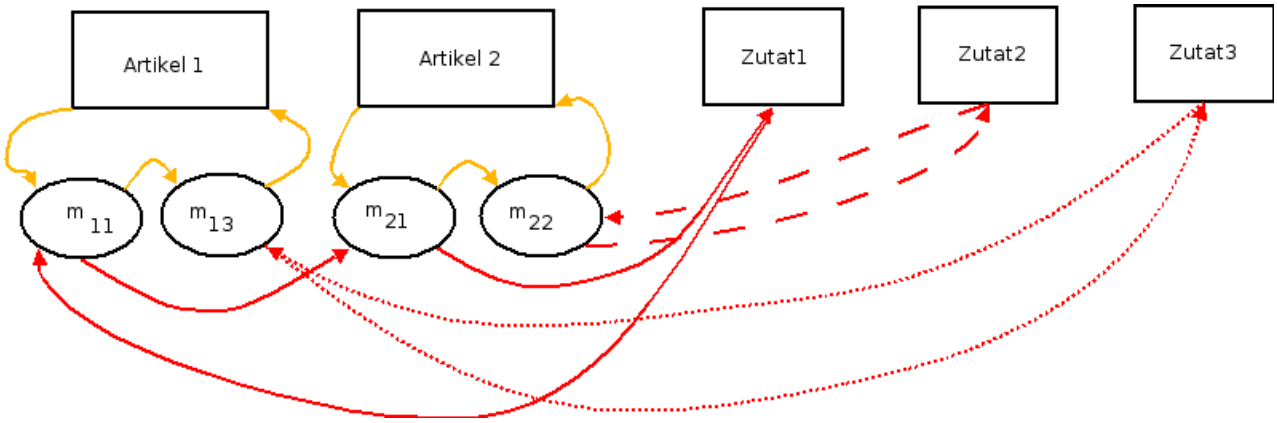
2.2. Netzwerkartige Datenbanken

(vgl. Vossen, S.83-114)

Netzwerkartige Datenbanken sind standardisiert worden durch CODASYL (Conference on Data Systems Language). Das NW-Datenbankmodell besteht aus folgenden Elementen:

(1) Segmente: Datensätze gleicher Art werden jeweils in einem Segment organisiert.
(Bsp.: Segment = ARTIKEL, Segment = ZUTAT,
Segment = KOMPONENTE)

(2) Sets (Mathematisch: Mengen): 1 Datensatz eines Segmentes A kann mit n Datensätzen eines Segmentes B verknüpft sein.
(Bsp.: Set1: Die Bezeiehung „Inhaltsstoff“: 1 Artikel enthält n Zutaten;
Set2: Die Beziehung „Wo enthalten“: 1 Zutat ist in m Artikeln enthalten)



Legende:
m_{ij} : Zutat j ist in Artikel i mit Menge m_{ij} enthalten

Die Menge m_{ij} ist ein Attribut des Segments KOMPONENTE (siehe oben)

3. Relationale Datenbanksysteme

3.1. Das relationale Datenbankmodell

Eine relationale Datenbank (RDB) ist eine Vereinigungsmenge von Tabellen T: $RDB = \bigcup_{i=1}^{M} T$

Jede Tabelle T ist nach Spalten und Zeilen aufgebaut:

	Sp_1	Sp_2	...	Sp_j	...	Sp_k
Z1						
Z2						
...						
Zi	w_{i1}	w_{i2}	...	w_{ij}	....	w_{ik}
...						
Zn						

} Einträge ins Data Dictionary

Nutzdaten der Tabelle T

Für alle Spalten Sp_j ($1 \le j \le k$) gilt:

1) Jede Spalte Sp_j wird durch einen Attributnamen A_j identifiziert
2) Jeder Spalte A_j ist ein Datentyp zugeordnet. dtj = dt(A_j)
   Für relationale Datenbanken sind folgende Standard-Datentypen festgelegt:
   a) char (n) : Zeichenketten mit Länge n ($n \in \mathbb{N}$)
   b) integer : ganze Zahlen z ($z \in \mathbb{Z}$) mit $-2^{31} \le z \le 2^{31}-1$
   c) float : für Gleitpunktzahlen (= Datentyp double in C oder Java für rationale Zahlen $q \in \mathbb{Q}$)
   d) decimal(p,q) : für rationale Festpunktzahlen mit p Dezimalstellen, davon q Nachkommastellen
   e) date : für Datumsangaben

Je nach Datenbankprodukt werden darüberhinaus auch noch folgende Datentypen unterstützt:
f) serial : wie integer, aber mit streng monoton steigender Reihenfolge
g) varchar [([a,]b)] : Zeichenkette mit variabler Länge, mit minimal a und maximal b Zeichen
h) datetime : Für Datums- und Zeitangabe (vgl. Petkovic, S.31ff)

Diese Datentypen werden atomar genannt, da sie aus Sicht des Datenmodells nicht weiter zerlegbar sind.

3) Für jede Spalte Sp_j ist ein Wertebereich W_j festgelegt. W_j ist entweder der Wertebereich, der durch den Datentyp dt_j festgelegt ist, oder er ist eine Teilmenge davon, wenn für das Attribut A_j Integritätsbedingungen wirksam sind.

Fasst man die Bedingungen 1), 2) und 3) zusammen, so ist das Relationenschema RS(T) der Tabelle T gegeben: RS (T) = { (A_j, dt_j , W_j) | $1 \le j \le k$ }

Anmerkung: Eine Zeile z_i kann als k-Tupel dargestellt werden: $z_i = (w_{i1}, w_{i2}, ..., w_{ij}, ..., w_{ik})$ (mit w_{ij} ist der Wert des Attributs A_j) Also gilt $z_i \in W_1 x W_2 x ... x W_k$, denn $w_{ij} \in W_j$

Die Menge $W_1 x W_2 x ... x W_k$ ist das kartesische Produkt der Mengen W_1 , W_2 ,...,W_k .
Die Menge $W_1 x W_2 x ... x W_k$ ist der Wertebereich der Tabelle T

3.2. Der Sprachumfang von SQL

SQL: Structured Query Language
SQL ist die Anfrage-, Definitions- und Transaktionskontrollsprache, die kanonisch mit einem RDBMS verknüpft ist. Der Sprachumfang von SQL kann in drei Teilmengen zerlegt werden:

a) Data Definition Language (DDL): enthält Befehle die auf dem Data-Dictionary des DBS agieren. Befehle zum Anlegen von Tabellen: CREATE TABLE
Anlegen von Indizes: CREATE INDEX
Anlegen von Benutzersichten: CREATE VIEW

Befehle zum Ändern von Tabellen: ALTER TABLE
Löschen von Tabellen: DROP TABLE

Befehle zum Verwalten von Zugriffsrechten auf Tabellen:
- GRANT → verteilt Zugriffsrechte
- REVOKE → widerruft Zugriffsrechte

Bsp.: Allgemeine Syntax des CREATE TABLE – Befehls:

CREATE TABLE tabname (sp1 dt1 [NOT NULL] [sp1_IntegrBed] [, ... , ... , ..., spN dtN [NOT NULL] [spN_IntegrBed], [Tab_IntegrBed]])

Erläuterung:

sp1, ..., spN:	Attribut (Spaltenname) 1 bis N der Tabelle
dt1, ..., dtN:	Datentyp der Spalten 1 bis N
sp_i IntegrBed:	Integritätsbedingung der Spalte i ($1 \leq i \leq N$)
Tab_IntegrBed:	Integritätsbedingung, die für die ganze Tabelle wirksam ist.

[] → Diese Bedingungen sind nicht zwingend erforderlich, um eine Tabelle anlegen zu können.

Bsp.: CREATE TABLE kunde (knr Integer NOT NULL PRIMARY KEY [CONSTRAINT prik_knr], kname char (30), kalter Integer)

- PRIMARY KEY CONSTRAINT prik_knr → Integritätsbedingung für das Attribut knr
- prik_knr → Regelname

CREATE TABLE auftrag (aufnr Integer NOT NULL PRIMARY KEY, aufsum decimal (7,2), knr Integer NOT NULL, FOREIGN KEY (knr) REFERENCES kunde(knr) CONSTRAINT fkey1)

Hier ist FOREIGN KEY – Bedingung Beispiel einer Integritätsbedingung, die auf das Attribut knr (Kundennummer) wirkt.

b) Data Manipulation Language (DML):
DML enthält Befehle zum lesenden und schreibenden Zugriff auf Tabellenzeilen (SELECT, INSERT, UPDATE, DELETE).

c) Data Control Language (DCL):

Befehle für:

(1) Transaktionsverwaltung:

- BEGIN WORK → für den Transaktionsanfang
- COMMIT WORK → für das Wirksamwerden aller Operationen der Transaktion auf der DB (entspricht regulärem Ende der Transaktion)
- ROLLBACK WORK → DB wird in ihren ursprünglichen Zustand zurückversetzt (entspricht irregulärem Ende der Transaktion)

(2) Datensicherung:

- UNLOAD → Sicherungskopie einer Tabelle erstellen
- LOAD → Laden einer Sicherungskopie in die DB

3.3. Befehle der DML

a) Der INSERT-Befehl dient zum Einfügen neuer Tabellenzeilen:

Allgemeine Syntax: INSERT INTO tabname [(sp_1, sp_2, ..., sp_N)] VALUES (wert_1, wert_2, ..., wert_N)

Bsp.: INSERT INTO (art_nr, art_preis, art_datum) VALUES (4711, 17.55 , '10/27/2005')

Anm.: SQL-Konstanten:

- Integer-Konstanten: ganzzahlige Dezimalkonstanten → z.B.: 31, -421
- Float-, Dezimal-Konstanten: Rationalzahlkonstanten → z.B.: 3.14, -255.89
- Zeichenkettenkonstanten: 'XYZ....ABC' (es gibt auch Datenbank-Produkte, die die normale Notation „XYZ....ABC“ akzeptieren)
- Datumskonstanten (TT,MM,JJJJ): 'MM\TT\JJJJ' (es geht auch bei guter Systemeinstellung 'TT.MM.JJJJ')

b) Der UPDATE-Befehl ändert Werte von vorhandenen Tabellenzeilen:

allgemeine Syntax: UPDATE tabname SET sp_1 = ausdruck_1 [, sp_2 = ausdruck_2 , ..., sp_N = ausdruck_N] [WHERE logische Bedingung]

Bsp.: UPDATE art_tab SET art_bez = 'Bleistift' WHERE art_id = 4711

Die Tabellenzeile zum Artikel mit art_id = 4711 wird mit dem neuem Wert von art_bez überschrieben.

Bsp.2: UPDATE art_tab SET art_preis = 1.03 * art_preis

Alle Tabellenzeilen von art_tab werden mit dem 1,03 fachem des ursprünglichem Wertes überschrieben.

Anm.: Arithmetische Ausdrücke in SQL können gebildet werden aus:

- Numerischen Konstanten
- Attributnamen, die einen numerischen Datentyp haben
- arithmetischen Operatoren (+, -, *, /)

  Weiterhin können in solchen Ausdrücken auch numerische Skalarfunktionen auftreten (je nach RDBMS-Produkt).

c) Mit dem DELETE-Befehl werden Zeilen einer Tabelle gelöscht.

Allgemeine Syntax: DELETE FROM tabname WHERE [logische Bedingung]

Bsp.: Löschen des Artikels 4712:

DELETE FROM art_tab WHERE art_nr = 4712

3.4. Das SELECT-Kommando für Abfragen auf eine Tabelle

Allgemeine Syntax: SELECT [select_art] spaltenauswahl FROM tabname [WHERE logische Bedingung][GROUP BY spaltenname] [HAVING logische Bedingung] [ORDER BY spaltenfolge] [INTO TEMP temptabname]

Zu den einzelnen Klauseln der SELECT-Anweisung:

a) select_art := ALL / DISTINCT
b) spaltenauswahl := * / Spaltenliste / sp_Auswahlausdruck

Eine Spaltenliste ist eine Folge von Attributnamen, die untereinander mittels Kommata getrennt sind.

sp_Auswahlausdruck := Arithmetischer Ausdruck / Zeichenkettenausdruck / Ausdruck mit Agregatfunktion / Ausdruck mit Datumsfunktion

b.1)
Arithmetischer Ausdruck:

Der Aufbau arithmetischer SQL-Ausdrücke wurde bereits in der Anmerkung zu Bsp.2 zum UPDATE-Befehl beschrieben. Zu ergänzen sind:

Numerische Skalarfunktionen:

POW (a,b) = a^b mit a,b arithmetische Ausdrücke
MOD (a,b) = a(mod b) mit a,b ganzzahlige arithmetische Ausdrücke
ROUND(x,n) = Runden von x auf n Nachkommastellen mit x als Ausdruck vom Typ float, decimal(p,q)

(weitere Skalarfunktionen, siehe Petkovic Seite 36 ff.)

b.2) Zeichenkettenausdruck:
Ein Zeichenkettenausdruck wird gebildet aus Attributen, die von einem Zeichenkettendatentyp sind (z.B. char(n)) und unter Verwendung von Operatoren für Zeichenketten:

(i) Substring: spn[v,b] v: Position „von“ des Substrings
b: Position „bis“ des Substrings

(ii) Verkettung: sp1 || sp2 (| = Pipe-Symbol)

Bsp.: Tabelle = Kunde mit den Attributen name, vorname, ort (alle Attribute haben den Datentyp char(30))

SELECT name || ',' || vorname, ort[1,3] from Kunde

b.3) Ausdrücke mit Aggregatfunktionen:
(werden häufig in Kombination mit der GROUP BY Klausel verwendet)
count (*) : Alle Ergebniszeilen werden gezählt.
count (DISTINCT x) : Nur die verschiedenen Attributwerte von x werden gezählt.
sum (x) : (dtyp(x) : numerisch) Die Summe über alle Werte des Attributs x werden berechnet.
avg (x) : [average = (eng.) arith. Mittelwert] Der arith. Mittelwert aller Werte von x
min (x) : Das Minimum aller Werte von x
max (x) : Das Maximum aller Werte von x

Bsp.: Tabelle = artikel mit Attribut preis (dtyp(preis) = decimal(9,2))
SELECT sum(preis), avg(preis), min(preis), max(preis) from artikel

b.4) Ausdrücke mit Datumsfunktion
current: aktuelles Datum wird angezeigt
weekday (sp1): [Vorraussetzung : dtyp(sp1) = date] Der Wochentag des Datumwertes wird ausgegeben (von 0 für SO, bis 6 für SA)
mday(sp1, sp2, sp3) : [Vorraussetzung: dtyp(sp_i)=int (für i=1,2,3)] Aus der Monatsnummer von sp1, der Tagesnr. von sp2 und der Jahresnummer von sp3 wird ein Datum vom dtyp date erzeugt.

c) Die logische Bedingung (log_Bed) der WHERE-Klausel:
Jede logische Bedingung besteht aus einem oder mehreren Vergleichsausdrücken, die durch bool'sche Operatoren verknüpft sind.
Bool'sche Operatoren: AND, OR, NOT
Vergleichsoperatoren:
c.1) elementar : < , > , >= , <= , = , !=
c.2)Vergleich mit dem NULL-Wert : IS [NOT] NULL
Bsp.: SELECT * from Kunde where Ort is NULL
c.3)BETWEEN : Kontrolle, ob ein Wert in einem Intervall liegt:
Bsp.: WHERE plz BETWEEN 1000 AND 99999
c.4)LIKE-Operator für den Muster-Vergleich von Zeichenketten:
Wildcard-Zeichen: % : für beliebigen Teilstring
_ : für EIN beliebiges Zeichen
Bsp.: SELECT * from Kunde WHERE name like 'Z%'
Bsp.: SELECT * from Kunde WHERE name like 'Schmi__'
Bsp.: SELECT * from Kunde WHERE name[1,4] = 'Schm'
<=> SELECT * from Kunde WHERE name = 'Schm%' OR name = 'Schm'

c.5)IN-Operator vergleicht, ob ein Attributwert in einer Vergleichsmenge enthalten ist.
Bsp.: Tabelle Veranstaltung mit Attribut v_art (Veranstaltungsart)
SELECT * from Veranstaltung WHERE v_art IN('SPT , 'FIL')

Bsp.: SELECT * from Kunde WHERE ort NOT IN ('koeln', 'bonn', 'Aachen', 'Dueren')

1. Die GROUP BY-Klausel wirkt auf „Gruppen“ : Eine Gruppe besteht aus mehreren Tabellenzeilen, die bezüglich eines Attributs den gleichen Wert haben (z.B. in der Tabelle Veranstaltung haben mehrere Zeilen den gleichen v-art-Wert 'XXX')

   Bsp.: SELECT v_art, avg(preis) from Veranstaltung GROUP BY v_art

2. HAVING-Klausel : Diese Klausel prüft eine logische Bedingung für Gruppen (im Unterschied zur WHERE-Klausel, die logische Bedingungen für einzelne Tabellenzeilen prüft)
   Bsp.:
   SELECT V_art, avg(preis) FROM Veranstaltung GROUP BY v_art HAVING min(preis)>5

   Anmerkung: Wenn die GROUP BY-Klausel verwendet wird, können in der Spaltenauswahl des Selects nur noch Ausdrücke stehen, die von Attributen abgeleitet werden können, die in der GROUP BY-Klausel angegeben sind

   Bsp.: SELECT v_art, sum(preis), v_ort FROM veranstaltung GROUP BY v_art, v_ort

3. ORDER BY-Klausel dient der Sortierung der SELECT-Ausgabe. In der ORDER BY-Klausel gibt man die Attribute an, nach denen sortiert wird. Und man gibt an, ob aufsteigend (ASC[default]) oder absteigend sortiert werden soll (DESC)

   Bsp.: SELECT * FROM Kunde ORDER BY ort, name DESC, vorname

4. INTO TEMP-Klausel: Das Ergebnis einer SELEC-Anfrage ist immer eine Tabelle, die normalerweise nur auf der Standard-Ausgabe ausgegeben wird. Während einer Arbeitssitzung kann diese Tabelle temporär gespeichert werden. In der TEMP-Klausel wird der Name der temporären Tabelle angegeben

   Bsp.: SELECT name, ort, plz FROM Kunde INTO TEMP txxx

3.5. Mehrtabellenverarbeitung in SQL

A) Geschachtelte SELECT-Anfragen
   z.B. bei Verwendung des IN-Operators:

   SELECT * from Artikel WHERE gfken IN (SELECT gfl FROM gefahrgut)
   => Ergebnis: Alle Artikel, die einen gfken-Wert haben, der in der gefahrgut-Tabelle enthalten ist

B) JOIN : Ein JOIN ist eine SELECT-Anfrage, die sich in der FROM-Klausel auf mehrere Tabellen bezieht

   Bsp.: SELECT kunde.*, artikel.* FROM kunde, artikel
   => Ergebnis ist das kartesische Produkt der Menge der Tabellenzeilen der Kunde- (K) und Artikeltabelle (A)

   $K = \{k_1, k_2,...,k_m\}$, $A = \{a_1,...,a_n\}$ => $E = \{k_i, a_j\} \in KxA\} = KxA$

|E| = n * m

Allg. Syntax (für Spaltenauswahl) : TABELLENNAME.SPALTENNAME oder
ALIAS.SPALTENNAME
wobei der ALIAS in der FROM-Klausel zu definieren ist.

C) Der natürliche JOIN ist auf ein Attribut bezogen, das in 2 Tabellen vorkommt. Dieses Attribut kann in der WHERE-Klausel verwendet werden, um zusammengehörende Zeilen beider Tabellen zu verbinden.

Bsp.: Tabelle Auftrag mit den Attributen aufnr, aufdat, *knr* und aufsum (A)
Tabelle Kunde mit den Attributen *knr*, name, plz, ort (B)

Ges.: Zu einem gegebenen knr-Wertsollen Kunden-name, Kunden-Ort und alle Aufträge, die er bestellt hat, angezeigt werden

SELECT B.name, B.ort, A.* from Auftrag A, Kunde B WHERE B.knr=107
AND B.knr=A.knr

4. Zugriffe auf relationale DB mit JDBC

JDBC: Java to Database Connection.
Die JDBC-Funktionalität ist im Java-Paket java.sql enthalten.

4.1. JDBC-Treiber laden:

RDB-Produktanbieter (Oracle, DB2, Informix, mySQL, ...) stellen JDBC-Treiber zur Verfügung. Vor den DB-Zugriffen ist der JDBC-Treiber für die JVM zu laden.

Für Informix: String treiber = "com.informix.jdbc.IfxDriver";

Treiber registrieren:
```
try
{
        class.forName(treiber).newInstance();
}
catch (Exception ex1)
{
        System.out.println("Fehler: Treiber kann nicht geladen werden.");
        ex1.printStackTrace();
}
```

4.2. Verbindungen zur DB aufbauen

Nachdem der DB-Treiber registriert wurde, kann mit der Methode getConnection( ) der allgemeinen Treiber-Managerklasse DriverManager eine Instanz der Klasse Connection erzeugt werden.

a) Deklarieren und Initialisieren einer Connection-Instanz: Connection dbConnection = null;

b) URL-Parameter für den Verbindungsaufbau vorbereiten:

b1) String protokoll = "jdbc:informix-sqli::///"; /* Herstellerspezifisches DB-Protokoll */

Bsp.: andere DB-Protokolle sind:
- "jdbc:oracle:thin@" → Oracle
- "jdbc:mysql://" → mySQL
- "jdbc:odbc:" → für Zugriff auf eine ODBC-Brücke

b2) DB-Host: String host = "demokrit" + ":";
b3) Port des Hosts: String port = "5100";
b4) DB-Name: String dbName = "dbNN";
b5) DBMS-Server-Name (proprietär gebildet): String ifxServer = "informixserver = demo_se";
b6) String dbURL = protokoll + host + port + "/" + dbName + ":" + ifxServer;

c) Verbindungsaufbau:

```
try{
        dbconnection = DriverManager.getConnection(dbURL);
        System.out.println("Verbindung aufgebaut");
}
catch (SQLException sqlex){
        System.out.println("Fehler beim Verbindungsaufbau");
        System.out.println("Connection Fehler:" + sqlex.getMessage( ));
}
```

Anm.: Neben der obigen getConnection( )-Methode für Host-Programme gibt es folgende Prototypen (z.B. für Applett-Zugriffe):

a) Connection getConnection(String url, String user, String Password);
b) Properties p = new Properties( );
   p.put ("password", "....."); /* ..... = Passwort */
   Connection getConnection(String url, Properties p);

4.3. SQL- und Java-Datentypen

Die folgende Tabelle gibt es eine Übersicht, welche SQL-Datentypen welchen Java-Datentypen entsprechen.

SQL	***JAVA-DATENTYPEN***
integer, serial	int
real	float
float	double
decimal(p,q)	double [in Annäherung] BigDecimal [Stellengenau]
date, datetime	[Sonderbehandlung]

4.4. Erzeugung eines Anweisungsobjektes

Für jede SQL-Anweisung, die in einem JDBC-Programm ausgeführt werden soll, muß ein Anweisungsobjekt erzeugt werden:

```
Statement st1;
st1 = dbConnection.createStatement( );
```

4.5. Ausführen von SELECT-Anfragen

Ist ein Anweisungsobjekt gegeben, dann können lesende SQL-Anfragen mit executeQuery( ) an die DBMS gesendet werden. Die Methode executeQuery( ) gibt eine Ergebnismenge zurück. Die Ergebnismenge ist vom JDBC-Datentyp ResultSet.

Bsp.: SELECT auf die Artikeltabelle mit den Attributen art_nr, art_bez und preis.

```
ResultSet rs1;
rs1 = st1.executeQuery("SELECT art_nr, art_bez, preis FROM t35artikel");
```

Hinweis: SQL-Anweisung wird als String-Argument an executeQuery( ) übergeben!

Verarbeitung des ResultSet:

Folgende Schritte werden im allgemeinen ausgeführt.

a) Navigation zum nächsten Element der Ergebnismenge (= nächste Zeile der Ergebnistabelle des SELECT) mit der ResultSet-Methode next( ).

b) Ein Element der Ergebnismenge entspricht einer Zeile der durch das SELECT erzeugten Ergebnistabelle. Eine solche Zeile hat den Aufbau (w_1, w_2, ..., w_n). Dieser Aufbau ist bestimmt durch die Spaltenauswahl-Klausel der SELECT-Anweisung. Hier hat man folgende Spaltenauswahl:

SELECT a_1, a_2, ..., a_n FROM ...

Das heißt der Wert w_i korrespondiert mit dem Attribut a_i und hat dessen SQL-Datentyp dt_i (für alle i mit $1 \leq i \leq n$).

c) Jeder Wert w_i wird mit einer getXXX( )-Methode gelesen, wobei XXX durch den Java-Datentyp bestimmt ist, der den SQL-Datentyp dt_i gemäß Tabelle 4.3 zugeordnet ist.

Bsp.:

Attribut	*dt_i*	*Java-Datentyp*	*getXXX( )*
art_nr	integer	int	getInt( )
art_bez	char(30)	String	getString( )
preis	decimal(7,2)	double	getDouble( )

Die getXXX( )-Methoden gehören zur Klasse ResultSet. Zunächst werden hier die Prototypen benutzt, deren Argument der Attributname a_i ist.

Bsp.: Verarbeitung von rs1 (s.o.)

```
while(rs1.next( )){
int e_art = rs1.getInt("art_nr");
String e_bez = rs1.getString("art_bez");
double e_preis = rs1.getDouble("preis");
.....
}       /* Verarbeitung der Werte */
```

Schließen des Anweisungsobjektes: st1.close( )

Anm.: (Navigationsmethoden): Die Navigationsmethoden (wie z.B.: next( )) sind vom Typ boolean. Ausnahme: Die Methoden beforeFirst( ) und afterLast( ) sind vom Typ void.

Methoden zum Navigieren in der Ergebnismenge:

```
next( ) = nächstes Element
first( ) = erstes Element
last( ) = letztes Element
previous( ) = vorheriges Element
absolute(int p) = p-te Element
relative (int r) = auf das r-te Element vorwärts bei (r > 0) oder rückwärts (r < 0)
beforeFirst( ) = Positionierung vor dem 1. Element
afterLast( ) = Positionierung nach dem letzten Element
```

Bsp.: SELECT-Anfrage, die aus Benutzerdaten zusammengebaut wird: Hier: Eingabe Mindestpreis und Vergleichsmuster für Artikelbezeichnung:

"Pseudocode" für die Versorgung des JDBC-SELECT mit Benutzereingaben:

```
String amuster;
double preis;
amuster = IO1.einstring( );
mpreis = IO1.double( );
Statement st3;
ResultSet rs3;
st3 = dbConnection.createStatement( );
String h;
h = " ´ " + amuster + "%";
rs3.executeQuery("SELECT art_nr, art_bez, preis FROM t35artikel WHERE preis >      " +
mpreis + " AND art_bez LIKE " + h);
/* Weiterverarbeitung der Ergebnismenge wie oben oder mit anderen     Navigationsmethoden*/
```

4.6. Schreibende SQL-Anweisungen

JDBC-Verwaltung von INSERT, UPDATE und DELETE wird mit der Statement-Methode executeUpdate() durchgeführt.

Signatur: int executeUpdate(String SQL-Anweisung)
^gibt die Anzahl der erfolgreich eingefügten, überschriebenen der gelöschten Tabellenzeilen zurück

Bsp.1: (Senkung aller Sozialbeiträge um 10%)
```
Statement st3;
st3=dbc1.createStatement();  //dbc1: eine bereitsexistierende DB-Verbindungsinstanz
n=st3.executeUpdate("Update STUDENT set sozialbt=0,9*sozialbt");
System.out.println(n+“Zeilen der Tabelle STUDENT wurden überschrieben.“);
```

Bsp.2: (Löschen aller Kunden mit einer bestimmten PLZ)
```
Geg.:   int eplz = 53115;
        n = st5.executeUpdate("DELETE FROM kunde WHERE k_plz="+e_plz);
```

4.7. Metadatenabfrage

Geg.: eine ResultSet-Instanz rs1, die durch eine lesende SQL-Anfrage erzeugt wurde.
(z.B. rs1=st7.executeQuery("SELECT * FROM kunde");)

Ges.: Metadaten der Ergebnismenge (= Metadaten der gegebenen Tabelle, wenn die SELECT-Anfrage ohne Einschränkungen war)

- Attributnamen A_i
- SQL-Datentypen $dt(A_i)$
- Anzahl N aller Zeilen der Tabelle

1) Metadateninstanz anlegen:
```
ResultSetMetaData rsmd1;
rsmd1=rs1.getMetaData();
```
2) Metadatenistanz auswerten:
```
int m; //Spaltenanzahl
String attnam, dtnam; //Attributname, Datentypname
int i;
m=rsmd1.getColoumnCount();
for (i=1; i<=m; i++) {
 attnam=rsmd1.getColoumnName(i);
 dtnam=rsmd1.getColoumnTypeName(i);
 System.out.println("Spalte"+i+" "+attnam+" hat Datentyp "+dtnam);
}
int N;
N=rsmd1.getRowCount();
```

4.8. Verarbeitung von decimal(p,q)-Attributen mit der Java-Klasse BigDecimal

1) Lesender Zugriff auf einen decimal(p,q)-Wert in einem ResultSet und Schreiben dieses Wertes in eine BigDecimal-Instanz:
   a) Deklaration einer BigDecimal-Instanz:
      BigDecimal bd1; //vorher: Paket java.math importieren

b) Lesen und Schreiben:
bd1 = rs1.getBigDecimal("k_Kreditlimit");
^ ^SQL-Attribut vom Datentyp decimal(p,q)
^erzeugt eine BigDecimal-Instanz

2) Durch SQL-Schreiboperationen sollen decimal(p,q)-Attribute einer Tabelle mit korrekten Festpunkt-Zahle gefüllt werden:
Bsp.: i) in Tabelle ARTIKEL soll das Attribut preis überschrieben werden.
ii) in Tabelle KUNDE soll das Attribut K_Kreditlimit überschrieben werden

Geg.: String epreis; // (aus einer seq. Datei mit DS-Aufbau eartnr, epreis)
double e_klimit; // (aus einer Dialogschnittstelle)

Arbeitsschritte:
a) Konvertieren der Eingangsgrößen (epreis, e_klimit) nach BigDecimal
b) Eventuell BigDecimal-Rechenoperationen
c) Schreiben von BigDecimal-Werten in eine Tabelle

zu a)
double -> BigDecimal : BigDecimal bd1 = new BigDecimal(e_klimit);
String -> BigDecimal : BigDecimal bd3 = new BigDecimal(epreis);
double <- BigDecimal : double x = bd3.doubleValue();
String <- BigDecimal : String h = bd3.toString();

zu b)
Geg.: BigDecimal-Instanzen bdx, bdy mit korrekten Festpunktzahlen.
Berechnen von BigDecimal-Summen (bdz) , -Differenzen (bdw) , -Produkten (bdp) , -Quotienten (bdq)

b1) bdz = bdx.add(bdy);
b2) bdw = bdx.sub(bdy);
b3) bdp = bdx.mul(bdy);
b4) bdp = bdx.div(bdy, ROUND_HALF_UP);
^kaufmännisches Runden

Signatur: BigDecimal div(BigDecimal b1, int rundungsmodus)
Anmerkung: (Stelligkeit)
bdx hat q_1 Nachkommastellen
bdy hat q_2 Nachkommastellen
=> bdz, bdw haben max(q_1,q_2) Nachkommastellen
=> bdp hat (q_1+q_2) Nachkommastellen
b5) Abfragen der Anzahl der Nachkommastellen einer BigDecimal-Zahl
int n = bdp.scale();
b6) Setzen der Nachkommastelligkeit einer BigDecimal-Zahl mit setScale()
Signatur: BigDecimal(int letzteStelle, int rundungsmodus)
Bsp.: BigDecimal c, d = new BigDecimal(3.1415)
c=d.setScale(2,4); //4 = Rundungsmodus kaufmännisches Runden
// => c = 3.14
c)
c1) UPDATE der Artikeltabelle
K = st9.executeUpdate("UPDATE ARTIKEL set preis="+bd3+" WHERE artnr="+eartnr);
c2) UPDATE der Kundentabelle
K1=st10.executeUpdate("UPDATE KUNDE set K_Kreditlimit="+bd1+" WHERE

k_plz="+50678);

4.9. Verarbeitung des SQL-Datentyps DATE, Prepared Statements

a) Lesender Zugriff auf DATE-Attribute:
a1) SQL-Date-Attributwert in eine Instanz von java.util.Date einlesen:

```
java.util.Date dx1;
dx1 = rs1.getDate("AUFDAT");
```

a2) Konvertierung von java.util.Date-Werten in Integer-Werte für Tag, Monat, Jahr:
i) Kalenderinstanz :

```
Calendar cal1;
cal1=Calendar.getInstance();
```

ii) Kalenderinstanz mit Date-Werten füllen:

```
cal1.setTime(dx1);
```

iii) Kalenderinstanz auswerten:

```
int j,m,t;
j=cal1.get(Calendar.YEAR);
m=cal1.get(Calendar.MONTH);
t=cal1.get(Calendar.DAY_OF_MONTH);
```

Bsp.: Kontrolle auf Aufträge, die älter als 1 Jahr sind

```
int d; a=2005;
d=a-j;
```

a3) Konvertierung: DATE => String

```
String s1;
s1=dx1.toString( );
```

b) Schreibender Zugriff auf SQL-Date-Attribute / Prepared Statement

(1) Benutzerwert für ein in die DB zu schreibendes Datums-Attribut von der Tastatur einlesen:

Einlesen als String-Wert in der Form JJJJ-MM-TT:

```
String edat;
edat = new String ("2005-02-08");
```

→ (alternativ: edat in diesem Format von der Tastatur mit System.in.readLine( ), bzw. IO1.einstring( ) einlesen.)

(2) String in eine Instanz vom Typ SQL-Date umwandeln:

```
java.sql.Date dn1 = java.SQL.Date.valueOf(edat);
```

(3) Den Wert der SQL-Date-Variable in ein UPDATE-, INSERT- oder DELETE-Kommando einbauen.
(3.1) innerhalb einer executeUpdate( )-Methode (vgl. 4.6)
(3.2)innerhalb eines Prepared-Statements:
Ein Prepared-Statement ist ein SQL-Kommando, in dem für bestimmte Werte (z.B.: Werte in Vergleichsausdrücken wie bei logischen Bedingungen, in der SET- oder VALUE-Klausel von UPDATE bzw. INSERT, ...) Platzhalter vorgesehen werden. Das Platzhaltersymbol ist das ?-Symbol.

Bsp.: a) String des Prepared-Statements:

```
String pstxt = "UPDATE art_tab SET art_dat = ? WHERE art_nr = ?";
```

b) Anlegen einer Prepared-Statement Instanz:

PreparedStatement ps = dc1.prepareStatement(pstxt);
↑
eine Connection-Instanz

Beim Anlegen von ps wird pstxt geparst und dabei werden die Platzhalterpositionen gespeichert.

c) Platzhalter mit Werten füllen. Hierzu gibt es in der Klasse Prepared-Statement für jeden aus SQL-Sicht elementaren Datentyp eine Set-Methode (z.B.: setInt( ), setDouble( ), ...). Hier werden die Platzhalter für den art_dat- und art_nr-Wert gefüllt:

```
ps.setDate(1,dn1);      /* allgem. Syntax: ps.setXXX (Platzhalterposition, -wert) */
ps.setInt(2,4711);
```

d) Prepared-Statement ausführen:

```
int K =  ps.executeUpdate( );
```

Anm.: Entsprechendes Vorgehen bei Prepared-Statements für lesenden Zugriff: a), b), c) analog
d) ResultSet rs1 = ps.executeQuery( );

5. Logische Datenanalyse

Die logische Datenanalyse behandelt aus dem Software Engineering die Aspekte der Anforderungsanalyse und der Spezifikationen, die zum Design von DBS erforderlich sind. Beim Design von DBS werden die Prinzipien und Methoden behandelt, die im Hinblick auf unterschiedliche DBMS als Zielsysteme erforderlich sind.

==> Folie 01: Phasenmodell: Grobentwurf, Spezifikationen, Design und Implementation eines DBS
URL: http://www.nt.fh-koeln.de/fachgebiete/inf/buechel/skript5b.pdf

5.1. Phasenmodell des Software-Engineering und typische Dokumenttypen der Phasen

A) Phase 1: Anforderungsanalyse

Typische Dokumente:

a) Methodik der Strukturierten Analyse (Str.A):
- IFX (Schnittstellenübersicht)
- IF0 (Datenflussdiagramm [Systemübersichtsdiagramm])

b) Methodik der objektorientierten Analyse (OOA):
i) Klassendiagramm (grobe Form)
ii) Anwendungsfalldiagramm

c) Phasenabschlussdokument: Das Lastenheft

B) Phase 2: (System)- Definition:

(Ziel: Die Komponenten des zu entwickelnden Systems sollen aus fachlicher Sicht (= Sicht des Anwenders) vollständig beschrieben sein und in ihrer Funktionalität (aus Sicht des Entwicklers) logisch korrekt beschrieben sein.)

Typische Dokumente:

b) OOA: i) Klassendiagramm (verfeinerte Form: Klassen mit vollständiger Attributliste, Vererbungsbeziehungen und Assoziationen)
ii) Sequenzdiagramm (für das zeitliche Verhalten von Methoden untereinander)
....

a) Str.A: i) Pro logischer Funktion F_n des IF0 ein spezifisches DFD: If_n
ii) Data-Dictionary mit Spezifikationen der Informationsflüsse und -speicher

c) Str.A und OOA: i) Zustandsdiagramme (zur Beschreibung von Funktionen und Methoden)
.....

d) Phasenabschlussdokument: Das Pflichtenheft

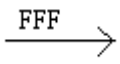
Bsp.: Aktivitäten der Phase 1 am Beispiel des Projektes W3-Buchhandel (W3-BUCHH) dargestellt.

1) Anforderungskatalog (Mitteilung des Anwenders)
→ Folie 02: Anforderungskatalog W3-BUCHH
URL: http://www.nt.fh-koeln.de/fachgebiete/inf/buechel/skript5e.pdf

2) Abgrenzung des zu entwickelnden Systems „nach außen“: Schnittstellenübersicht. Das zu entwickelnde System bekommt oder sendet Daten an externe Instanzen (= externe Schnittstellen).

→ Symbol für externe Schnittstellen mit Namen xxx

→ Symbol für das zu entwickelnde System SYS



→ Symbol für einen Datenfluss FFF

→ Folie 03: Schnittstellenübersicht
URL: http://www.nt.fh-koeln.de/fachgebiete/inf/buechel/skript5c.pdf

3) Innerhalb des zu entwickelnden Systems müssen
- die logischen Hauptfunktionen des Systems identifiziert werden
- die Datenflüsse zu ihrer Kommunikation untereinander und mit den externen Schnittstellen bestimmt werden.
(Kommunikationsarten: - synchron

- asynchron (→ Informationsspeicher))

IF0-Symbole:

→ Symbol für eine logische Funktion n

→ Symbol für einen Informationsspeicher ISP

→ Folie 04: Systemübersichtsdiagramm (IF0)
URL: http://www.nt.fh-koeln.de/fachgebiete/inf/buechel/skript5d1.pdf

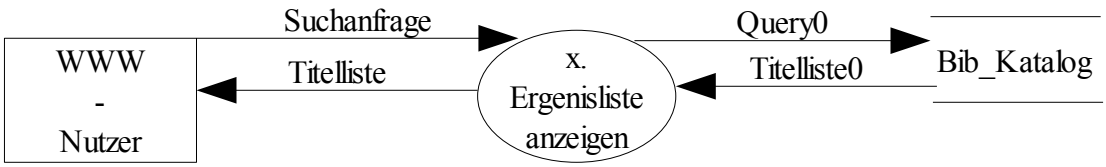
Lit.Hinweis: Rupp/Hahn... „UML 2-glasklar“
München, Wien (Hanser), 2005

Anm.:
JDBC Datum-String: JJJJ-MM-TT
In dieser Form kann der Datums-String (in einfachen Hochkommata) von JDBC aus mit INSERT eingefügt werden.

5.2. Spezifikation von Informationsflüssen (IFL) und Informationsspeichern (ISP) un Data-Dictionary-Notation

In der Phase 2 (Systemdefiniton) geht es u.a. darum, die Informationsflüsse und ISP aus fachlicher Sicht vollständig zu beschreiben. Insbesondere geht es darum, den Aufbau der IFL und ISP aus Informationselementen vollständig anzugeben. Informationselemente IE eines ISP sind die Kandidaten für die später in der Datenbank zu speichernden Attribute.(Die IE sind die atomaren Bestandteile der IFL und ISP).

Bsp.:



Aufgaben der Spezifikation eines IFL bzw. ISP:

(a.1) Alle benötigten IE werden aus fachlicher Sicht für den IFL definiert (IE wird identifiziert mit einem systemweit eindeutigen IE-Namen). Diese IE-Definition wird in das Data-Dictionary der Phase 2 eingetragen.

(a.2) Im Data-Dictionary prüfen: liegt ein IE bereits dort mit gleicher Bedeutung aber anderem Namen vor?

(a.3) IFL bzw. ISP in ihrem syntaktischen Aufbau aus IE zusammensetzen. Für jeden IFL bzw. ISP wird eine Produktionsregel angegeben. Die Produktionsregeln der Data-Dictionary-Syntax haben einen BNF-ähnlichen Aufbau (BNF: Backus-Naur-Form).

Allgemeiner Aufbau einer Produktionsregel:

(I) für IE: IE_NAME := IE_Bedeutung (IE_Bedeutung: Freitext, der aus fachlicher Sicht eindeutig definiert ist)
(II) für IFL / ISP: IFIS_NAME := rsPROD(IE_1,..,IE_n)

mit: rsPROD( ) ist die rechte Seite der Produktionsregel, in der die IE_1,...,IE_n (Informationselemente) durch folgende Operatoren verknüpft werden können:
(1) Sequenz: + (in BNF: Kein Zeichen)
(2) Alternative: [...|...] (in BNF: |)
(3) Wiederholung: {...} (in BNF: { })
Wiederholung mit Vielfachheiten: (mindestens a mal, höchstens b mal ($a, b \in \mathbb{N} \cup \{0\}$))

${}_a\{...\}_b$ oder $\{...\}_{a:b}$ (wie in der EBNF)
(4) Option: (...) (in BNF: [...])

Bsp. 1: Spezifikation der IFL Suchanfrage.
a.1) Benötigte IE: BE := Benutzereingabe, die aus einer Zeichenkette mit Wildcard-Zeichen * am Ende bestehen darf (z.B. "UML*" , "Elektro*")
KAT := Kategorie der bibliographischen Angabe (z.B. "Titel", "Verfasser", "Erscheinungsort", "Jahr", "Schlagwort")
a.2) => ✓

a.3)Suchanfrage := (KAT) + BE

Anm.1: Keine Angabe von KAT <=> über alle Kategorien wird gesucht
Anm.2: Variante: Suchanfrage := KAT + BE und KAT = "ALLE" <=> über alle Kategorien wird gesucht

Bsp. 2: Benötigte IE
MELD1 := "Kein Treffer"
TITEL := Hauptsuchtitel eines Buches
VERF := Folge der Verfasser eines Buches
EORT := Erscheinungsort
EJAHR := Erscheinungsjahr
SWK := Schlagwortkette
ISBN := ISBN-Nr. des Buches
ANZ := Anzahl der gefundenen Titel

Titelliste0 := ANZ + [MELD1 | { TITEL + VERF + EORT + EJAHR + (SWK) + ISBN }]

Anm.: Beschreibung von Informationsspeicher durch Klassendiagramme

Bsp.1: Fassung des Klassendiagramms für den ISP Bib_Katalog:

KATALOGEINTRAG
TITEL VERF EORT EJAHR SWK ISBN
Suchen( ) ergebnisliste_anzeigen( )

UML-bezogene Regel: Die IE der IFL- bzw. ISP-Spezifikation werden zu Attributen der Klassen im Klassendiagramm.

Weitere Überlegung zur Spezifikation von Informationsspeichern:

In einen ISP können mehrere IFL eingehen (ISP = Senke in dieser IFL) und von einem ISP können mehrere IFL ausgehen (ISP = Quelle dieser IFL). Die ISP werden als passive Elemente des Systems angesehen. D.h. alle IE der ausgehenden IFL müssen durch IFL der eingehenden IFL erzeugt werden können. Ein ISP kann daher durch eine Eingabe- / Ausgabe-Matrix (E/A-Matrix) beschrieben werden (mit Zeilen = angreifende IFL und Spalten = Informationselemente). Für diese E/A-Matrix gilt dann: Für jede Spalte muss es mindestens einen eingehenden IFL geben, der Werte in das betreffende IE schreibt. (R1)

$$\text{EA\_MATRIX (ISP)} = \begin{pmatrix} m_{11} & m_{12} & \dots & m_{1K} \\ m21 & m_{22} & \dots & m_{2K} \\ \dots & \dots & \dots & \dots \\ m_{n1} & m_{n2} & \dots & m_{nK} \end{pmatrix}$$

mit K := Anzahl der im ISP vorkommenden Attribute
n := Anzahl der angreifenden IFL
$m_{ij} \in \{E, A, \_\}$ für alle $i \in \{1, ..., n\}$, $j \in \{1,...,k\}$

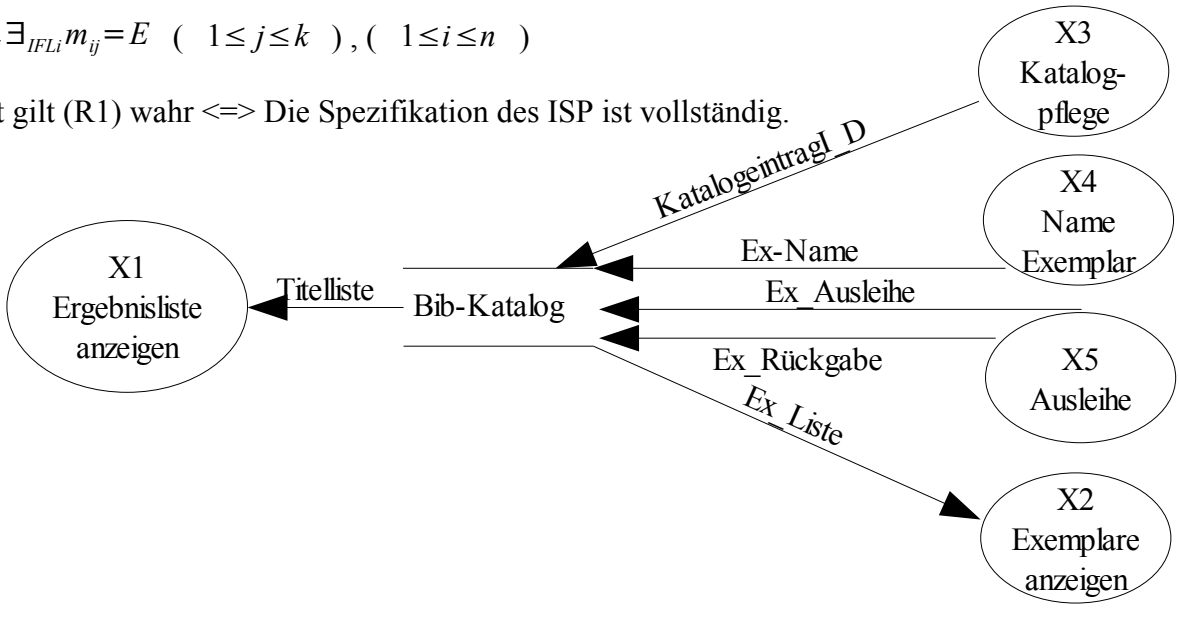
m_{ij} = { E <=> der IFL_i wirkt auf das IE_j schreibend
A <=> der IFL_i wirkt auf das IE_j lesend
_ <=> der IFL_i wirkt auf das IE_j nicht }

(R1) präzise formuliert, heisst:

$$\forall_{IEj} \exists_{IFLi} m_{ij} = E \quad (\quad 1 \le j \le k \quad) , (\quad 1 \le i \le n \quad)$$

Damit gilt (R1) wahr <=> Die Spezifikation des ISP ist vollständig.

Bsp.:



Für die Verwaltung der Exemplare benötigt man nun weitere IE:

SIGNATUR:=Zeichenfolge der Form BBBBNNNNNZZ, mit der jedes Exemplar eindeutig gekennzeichnet ist (Klebeschildeintrag auf dem Buchrücken).
STATUS := Kennzeichen, die den Verleih-Zustand des Exemplars kennzeichen (z.B. 1 = ausleihbar, 2 = vorbestellt, 3 = ausgeliehen, ...)
RDAT := Rückgabedatum

Für den IFL KatalogeintragI_D benötigt man folgendes IE:
KATKEY := Identifikator eines Katalogeintrags

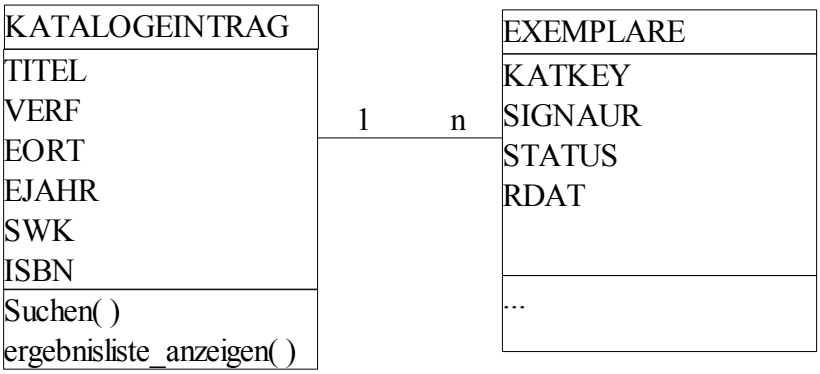
EA_MATRIX (Bib_Katalog) =

	KATKEY	TITEL	VERF	...	ISBN	SIGNATUR	STATUS	RDAT	...
Katalog_ID	E	E	E	...	E	-	-	-	
Titelliste0	-	A	A	...	A	-	-	-	
Ex_neu	A	-	-	...	-	E	E	-	
Ex_Ausleihe	-	-	-	...	-	A	E	E	...
Ex_Rückgabe	-	-	-	...	-	A	E	E	...
Ex_Liste	A	A	-	...	-	A	A	A	...
...	...	...	...	...	...	...	...	...	...

=> Diese Matrix erfüllt (R1)

Anm.: (Klassendiagramm 2. Fassung) Ordnet man die IE des Bib_Katalog nach fachlich gleichartigen Objektmengen (in der Bibliothek hat man a) die Menge der Katalogeinträge und b) die Menge der Buchexemplare),

dann kann man folgendes Klassendiagramm aufstellen:



Beziehungsart zwischen diesen Klassen hier:
Assoziation
1 Objekt der Klasse KATALOGEINTRAG ist mit n Objekten der Klasse EXEMPLARE

verbunden.

5.3. Spezifikation von Informationsspeichern mit Entity-Relationship-Diagrammen (ERD)

Man ordnet die IE eines ISP nach fachlich gleichartigen Objektmengen. Diese Objektmengen heißen Entitätenmengen. Ihre Elemente heißen Entitäten (entities). Entitätenmengen enthalten Objekte mit gleichen Attributen. Die IE, die in einer Entitätenmenge zugeordnet sind, sind die Attribute dieser Entitätenmenge. Zwischen Entitätenmengen können Beziehungen bestehen, die mathematisch als Relationen modelliert werden.

Geg.: Zwei Entitätenmengen E_1 , E_2 . Eine Relation R ist Teilmenge des kartesischen Produkts
$R \subset E_1 x E_2$ und es gilt: $R = \{(a,b) \in E_1 x E_2 | eine\ Regel\ r(a,b)\ ist\ wahr\}$

Bsp.: E = Bahnhöfe im VRS

R = $\{(a,b) \in ExE | a\ und\ b\ sind\ durch\ eine\ S-Bahnstrecke\ verbunden\}$

Weitere Beispiele zu Relationen:

Bsp.2: A = {3, 5, 7, 11}
B = {3, 5, 7, 11, ..., 31, 33, 35}

Relation R_2: $(a,b) \in A x B$ ist in R_2 enthalten <=> $b \in B$ hat $a \in A$ als Teiler.
Beschreibung von R_2: a) durch Angabe einer Regel (hier: eine prädikatenlogische Regel)
$R_2 = \{ \ (a,b) \in A x B \ | \ \exists_{a \in A} \equiv 0 (mod\ a) \ \}$
b) in aufzählender Form: [entspricht Menge der Einträge in einer Tabelle T, wenn R_2 durch T implementiert würde]

Bsp.3: A = { Studenten eines Studienganges }
B = { Vorlesungen in einem Studiengang }

Relation R_3: $st \in A$ hört $v \in B$.

Bsp.4: A = R^+_0 = { $x \in R$ | $x \geq 0$ }
B = R
R_4 { (x,y) $\in$ R^+_0 x R | $y^2 = x$}
$y^2 = x$ <=> y = $\sqrt{x} \vee y = -\sqrt{x}$
B = { $y \in R$ }

Zur Beschreibung einer Beziehung zwischen Entitätenmengen A und B können Relationen verwendet werden. Zur Kennzeichnung einer Beziehung sind drei Aspekte von Bedeutung:

(1) der Name der Relation (bzw. die Relationsregel) R.
(2) Quantitätenpaare (wieviele $a \in A$ sind wievielen $b \in B$ zugeordnet? Bzw.: Für ein beliebiges aber festes $a_0 \in A$ wird gezählt: Wieviele (a_0,b) gibt es in R ?)
(3) [Attribute der Relationen] (Die Attributangaben zu R können leer sein (vgl. Bsp.4), sie können ein Attribut enthalten (Faktor x in Bsp.2; S-Bahn-Name in Bsp.1; Kennzeichen_Praktikumsbesuch (j/n) in Bsp.3).

Def.: Sind A und B Entitätenmengen und ist $R \subseteq A x B$ eine Relation, dann heißt der Tripel RP =

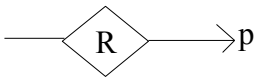
(R, q, C) der Bezeichnungstyp von R. (Relationship := (engl.) Bezeichnungstyp). Hierbei ist R die Relation, q das Quantitätenpaar von R und C die Attributfolge von R.

Anm.: Quantitätenpaare haben die Form (r,q) mit r,q $\in$ $\{1,c,n,m\}_1$ mit c $\in$ $\{0,1\}$
($\{...\}_1$ ist eine Symbolmenge). Damit können folgende Paare gebildet werden:

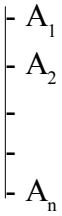
(1,1): 1 $a \in A$ gehört zu 1 $b \in B$
(1,n): 1 $a \in A$ gehört zu n $b \in B$ $n \geq 1$
(1,c): 1 $a \in A$ gehört höchstens zu 1 $b \in B$
(n,m): n $a \in A$ gehören zu m $b \in B$

Aufbau eines Entity-Relationship-Diagramms (ERD):

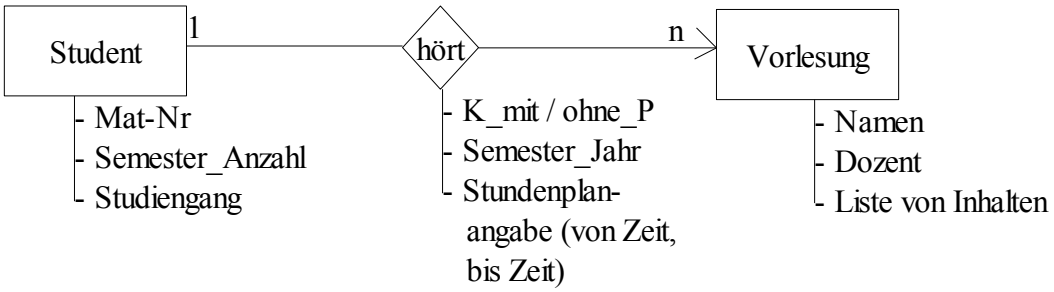
(1) Symbol für Entitätenmengen A: 

(2) Symbol für Beziehungstypen mit
Relationsname R und Quantitätenpaar (r,p): 

(3) Symbole für Attribute A_1, ..., A_n die einer Entitätenmenge A oder einem Beziehungstyp RP zugeordnet sind:



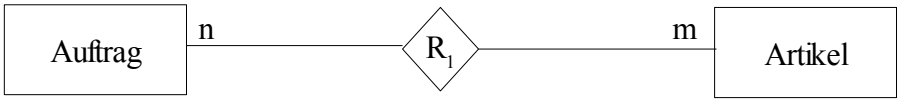
Bsp.: (zu Bsp. 3)



Anm. zu 2: Schreibweisen:

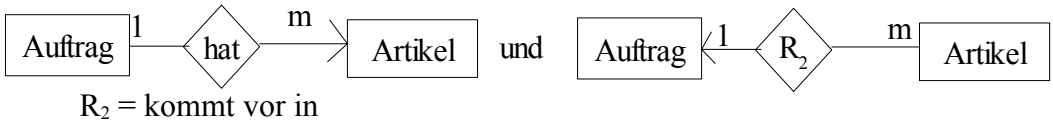
Beziehungstypen (ohne Leserichtung) häufig verwendet bei nicht spezifizierten (n;m)- Beziehungstypen (d.h. ohne Kante ⟶ gerichtete, werden Kanten der Form verwendet).

Bsp.:



R_1 = enthalten

Dieses kann genauer spezifiziert werden:

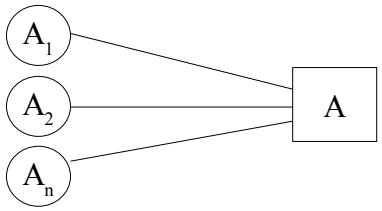


R_2 = kommt vor in

Anm. zu (3) Äquivalente Attributnotation:

(Kommentar: Busnotation ist rationeller

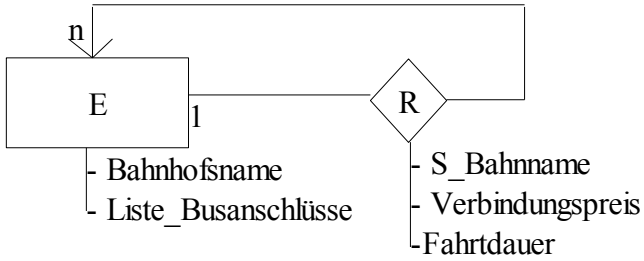
- A_1
- A_2
- ...
- A_n)



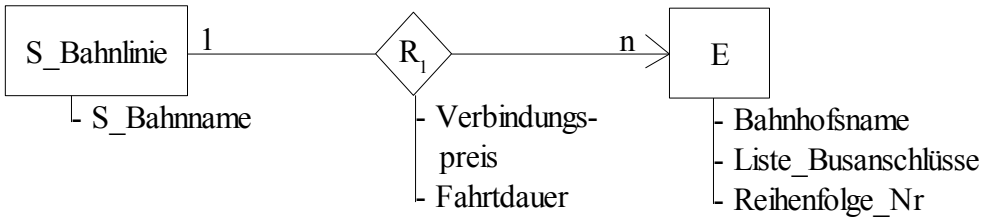
Bsp. (zu Bsp.1): Es gibt verschiedene Möglichkeiten, eine Beziehung zwischen Entitätenmengen in ein ERD umzusetzen:

E = Menge der Bahnhöfe
R = { $(b_1, b_2) \in E \times E$ | $\exists_{S-Bahn-Linie\,S}$ b_1 ist mit b_2 durch S verbunden}

<u>ERD1:</u>



<u>ERD2:</u>



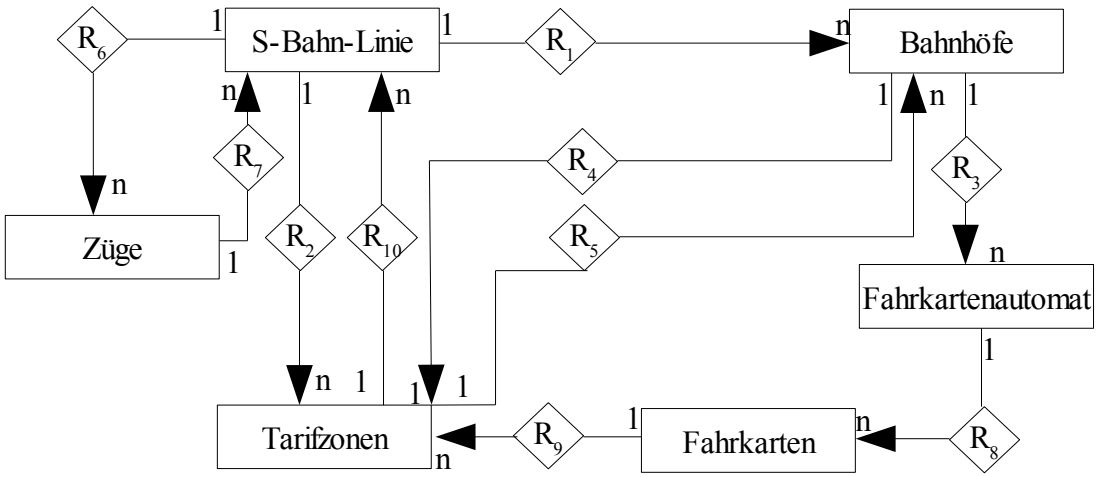
R_1: "verbindet"

<u>Anm.:</u> Die Entity-Relationship-Analyse, die zu einem ERD führt, ist ein Verfahren der <u>semantischen</u> <u>Datenmodellierung</u>. Die semantische Datenmodellierung ist ein Verfahren zur Überprüfung, ob die Datenstruktur, die der Informatiker entwirft, mit den <u>Bedeutungen</u> übereinstimmt, die aus der Sicht des Anwenders in dem Weltausschnitt (*) des geplanten Systems bestehen.

(*) Weltausschnitt: Objekte, die der Anwender als Gegenstände des geplanten Systems

identifiziert.

Anm.: Das ERD ist eine Art von semantischen Datenmodellen. (Semantik = Lehre der Bedeutung)



Beziehungstypen:
R_1: verbindet
R_2: befährt
R_3: besitzt
R_4: liegt in
R_5: beinhaltet
R_6: wird befahren
R_7: wird eingesetzt auf
R_8: erstellt
R_9: berechtigt zum Befahren von
R_{10}: wird befahren von
Entitäten:
- S-Bahnlinie
- Bahnhöfe
- Tarifzonen
- Fahrkartenautomaten
- Züge (physikalisch)
- Fahrkarten

Attribute zu Entitäten:
Bahnhöfe
- Bahnhofsname
- Busanschlüsse
- Gleise

5.4. Datenbank-Design

Gegeben: Phasenergebnis der Phase 2, insbesondere Spezifikation der ISP eines geplanten DBS durch ein ERD

Allgemein: Aufgabe der Design-Phase:

Auf Grundlage der Spezifikation (aus Phase 2) und in Hinblick auf die Hardware- und Softwareeigenschaften des Zielsystems (der Zielplattform) ist das DBS zu entwerfen.
Das Ergebnis ist die Softwarearchitektur des zu entwickelnden Anwendungssystems.

Bsp.: Wesentlicher Bestandteil der Softwarearchitektur eines datenbankgesützten Anwendungssystems ist das Datenbankschema.

Die Aufstellung der Softwarearchitektur ist mit Entscheidungen über das Zielsystem verbunden. Folgende Arten der Entscheidungen sind dabei relevant:

(1) HW-Entscheidungen
(2) Verteilungsentscheidungen (z.B. Host-Terminal, Client-Server, Peer-to-peer,...)
(3) Entscheidungen über die Systemsoftware in den beteiligten Rechnern.
(4) Entscheidungen über zu verwendende Programmiersprachen
(5) Entscheidung über die Organisation der Datenhaltung bzw. über die Art des einzusetzenden DBMS.

Bsp. Zu (5):

- Sequentielle Dateien
- ISAM Dateien
- LDAP
- RDBMS (relationales Datenmodell)
- OODBMS (objektorientiertes Datenmodell)
- HDBMS (hierarchisches Datenmodell)
- ORDBMS (objektrelationales Datenmodell)

In Hinblick auf das Datenbankdesign muss man sich daher mit Abbildungsverfaren beschäftigen: z.B.

A) Wie wird ein ERD auf ein RDBMS-Modell abgebildet?
B) Wie wird ein ERD auf ein OODBMS-Modell abgebildet? [-> Kap. 6]

usw.

Vorgehen

I) Die Attributangaben (zu allen Attributen der Entitätenmengen und der Beziehungstypen) sind mit Datentypangaben und Integritätsangaben (PRIK / FKEY; NULL; Werteintegrität) zu versehen. Hierbei ist zu prüfen, welche Datentypen von der Data-Dictionary-Komponente des DBMS-Zielsystems (=Datenmodell des DBMS) unterstützt werden.
Es ist auch zu prüfen, welche Arten von Integritätsbedingungen werden vom Zielsystem unterstützt?
(z.B. FKEY-Bedingungen in RDBMS-Produkten (vgl. Informix vs. MySQL); Realisierung von Werteintegritätsprüfungen durch CHECK-Klauseln (SQL-Sprachstandard) oder durch Kapselung beim OO-DB-Zugriff (=Zugriff auf eine DB aus einer OO-Sprache (Java, C++))

Bsp.: Betrachte die Attributliste der Entitätenmenge BAHNHOF:

	Ziel: RDBMS
-Bhf-ID	(int, PRIK) ✓
-Bahnhofsnamen	(char(n)) ✓
-Busanschlüsse	|____|$_1$
-Gleise	|____|$_2$

Probleme: |____|$_1$:

a) weitere ERD-Analyse erforderlich (eine Entitätenmenge BUSVERBINDUNG ist im bisherigen ERD „Verkehrsverbund“ (s.o.) noch nicht vorhanden).

b) kein elementarer SQL-Datentyp vorhanden (in SQL gibt es im Unterschied zu Java KEINE Kollektionsdatentypen (Listen, Mengen, Arrays))

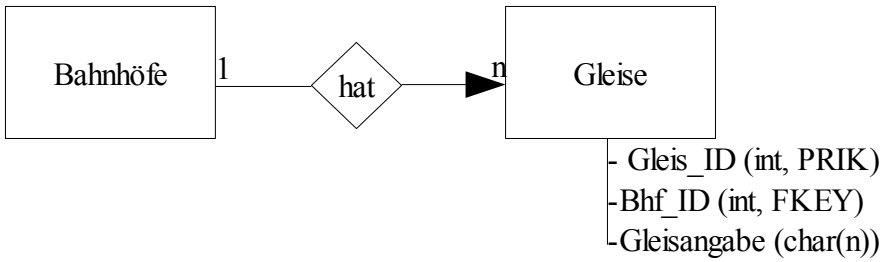
|____|$_2$: nur b) ist das Problem. Da:

Gleise := {Gleisangabe}
Gleisangabe := Gleisnr + [Gleisnr-Zusatz]
Gleisangabe hat Datentyp char (n)
Gesucht: Datentyp (Gleise) = LIST OF (Gleisangabe: char(n))

In SQL ist ein solcher LIST OF(...)-Datentyp nicht gegeben.

II) [für RDBMS als Zielsysteme]: Das ERD ist zu normalisieren, d.h. es gibt eine 1., 2. und 3. Normal-Form für das zu entwickelnde Datenbankschema, die aufzustellen sind. Insbesondere ist als Vorarbeit zur Aufstellung der 1NF (= 1. Normalform des DB-Schemas) das ERD um Entitätenmengen und Beziehungstypen solange zu ergänzen, bis alle Attribute durch elementare SQL-Datentypen verwaltet werden können.

Bsp.: Das ERD Verkehrsverband ist an der Entitätenmenge BAHNHÖFE durch eine Entitätenmenge GLEISE zu ergänzen:



III) [für OODBMS als Zielsysteme] Aus ERD ein Klassendiagramm mit Datentypen und persistenten Kollektionsklassen herleiten.
IV) [für ORDBMS als Zielsysteme] Kombinationen von II) und III) (als Kollektionsdatentypen hat man in ORDBMS i.d.R. nur SET und ARRAY)
V) [für LDAP] : Welche der (1:n)-Beziehungstypen des ERD werden als Verzeichnishierarchien (als Parent-Children-Beziehung) abgebildet?
VI) [für seq. Dateien / ISAM-Dateien] : Logische Datensatzbeschreibungen mit Schlüsselbedingungen sind aus dem ERD für die zugreifenden Programme verbindlich festzulegen. (z.B. als XML-DTD für die zu verwaltenden Dateien).

5.4.1 Umsetzung eines ERD in ein relationales Datenbankschema, das in 1. Normalform (1NF) ist

Def.1: Ein Datentyp heißt atomar (= unzerlegbar), wenn er unmittelbar durch einen Datentyp, der im Data-Dictionary des RDBMS gegeben ist, dargestellt werden kann.

Anm.: In SQL hat man standardmäßig folgende Menge DT von Datentypen:
DT = { integer, float, char(n), date, decimal(p,q), ... }.
Es gilt für jeden Datentypen dt: $dt \in DT$ => dt ist atomar.
Ein ERD, dessen Attribute alle mit einem $dt \in DT$ versehen werden können, ist für die Umsetzung in ein relationales DBS (=: RS(DB)) zureichend vorbereitet.

Def.2: a) Ein relationales DBS RS(DB) ist die Vereinigungsmenge der Schemata RS(T_i) der Tabellen

T_i ($1 \le i \le L$) der Datenbank DB: RS(DB) = $RS(DB) = \bigcup_{i=1}^{L} RS(T_i)$

b) Ein Tabellenschema RS(T) enthält für jede Spalte der Tabelle T die folgenden Einträge:
- Attributname A_k der Spalte
- Datentypen dt_k der Spalte
- Wertebereich W_k der Spalte (der Wertebereich W_k ist bestimmt durch den Wertebereich von dt_k, der durch Integritätsbedingungen, wie PRIK-, FKEY-, Werteintegritäts- oder NULLs-Bedingungen eingeschränkt werden kann; daher werden im RS(T) nur die einschränkenden Integritätsbedingungen angegeben).

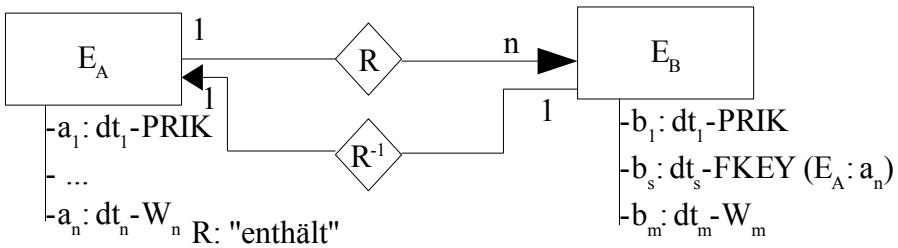
RS(T) = { (A_k, dt_k, Wk_k) | $1 \le k \le M$, M = Spaltenzahl (T) }

Def.3: RS(DB) liegt in 1NF vor <=> für alle Attribute A_{ki}, die in irgendeiner TabelleT_i der DB vorkommen sollen, gilt A_{ki} ist atomar ($1 \le i \le L$, $1 \le k \le M$).

Regeln zur Umsetzung eines ERD in ein RS(DB), das in 1NF ist:

Vor.: Das ERD ist soweit vorbereitet, das alle seine Attribute atomare Datentypen haben.
Regeln:(R1) Jede Entitätenmenge E_i des ERD wird in ein Tabellenschema RS(T_i) umgesetzt.
(R2) (1:1)-, (1:c)-, (c:1)-Beziehungstypen (ohne eigene Attribute) führen zu keinem weiterem Tabellenschemata. Allerdings ist zu prüfen, ob in den beteiligten Entitätenmengen die entsprechenden FKEY-Bedingungen gesetzt sind.
(R3) (1:n)-Beziehungstypen, die ohne eigene Attribute sind und deren Umkehrbeziehungstypen zu (R2) gehören, führen ebenso nicht notwendig zu weiteren Tabellenschemata. Bei der n-seitigen Entität E_B muß im zugehörigen Relationsschema RS(T_B) ein FKEY-Attribut eingetragen sein, das dem PRIK-Attribut in RS(T_A) entspricht (RS(T_A) ist das Tabellenschema für die 1-seitige Entitätenmenge des Beziehungstyps).

Bild (zu(R3)):

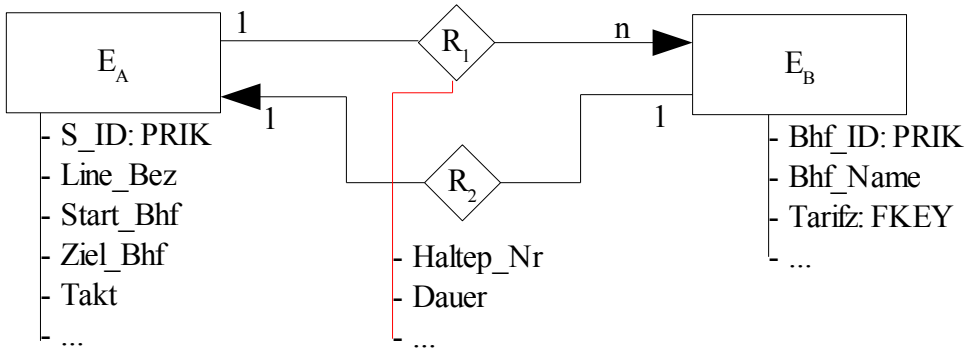


Bsp.: E_A : Tarifzone R: "enthält"
E_B: Bahnhof R^{-1}: " gehört zu"
PRIK(E_B) = bhf_nr
PRIK(E_A) = tfz_nr
In E_B muß ein Attribut bj = tfz_nr mit Integritätsbedingung FKEY(Bahnhof.bfh_nr)

vorhanden sein.

(R4) (1:n)-Bedingungstypen R, die nicht (R3) unterliegen (d.h. solche, die einen (1:m)-Umkehrbeziehungstypen haben oder die eigene Attribute haben), führen zu weiteren Tabellenschemata RS(T_i). Diese Tabellen haben mindestens zwei FKEY-Attribute, die jeweils einem PRIK-Attribut in E_A bzw. in E_B entsprechen.

Bsp.: Man betrachte das folgende Teil-ERD des ERD "Verkehrsverbund":



E_A : S-Bahnlinie
E_B : Bahnhof

R_1 : "hält in"
R_2 : "ist Haltepunkt von"

Aus (R1) folgen:

(1) RS(S-Bahnlinie) = RS(T_1)

A_k	S_ID	Linie_Bez	Start_Bhf	Ziel_Bhf	Takt	...
Dt_k	int	char(30)	int	int	int	...
W_K	PRIK		FKEY(E_B.Bhf_ID)	FKEY(E_B.Bhf_ID)		...

(2) RS (Bahnhof) = RS(T_2)

A_k	Bhf_ID	Bhf_Name	Tarifz	...
Dt_k	int	char(50)	int	...
W_K	PRIK		FKEY(Tarifzone.tfz_nr)	...

Aus (R4) folgt:

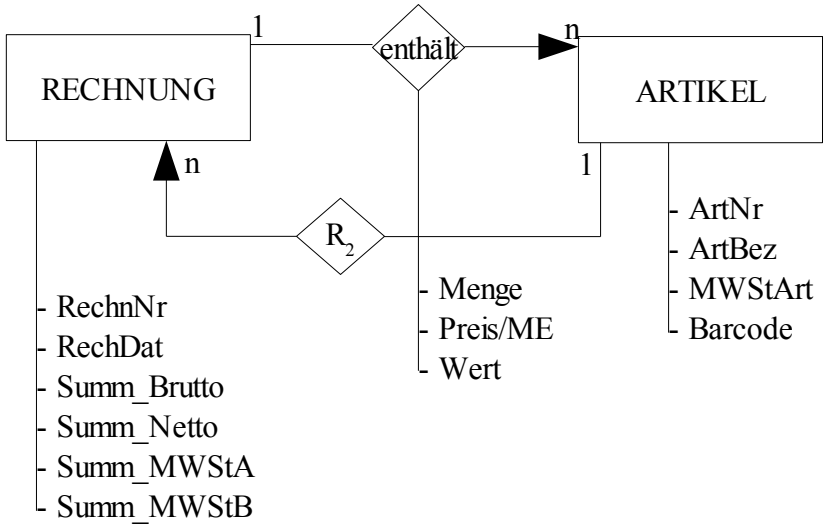
(3) RS(T_hält_in)

A_k	Halte_ID	S_ID	Bhf_ID	Haltep_Nr	Dauer	...
Dt_k	int	int	int	int	int	...
W_K	PRIK	FKEY(T_1_S_ID)	FKEY(T_2_Bhf_ID)	>0	≥ 0	...

5.4.2 Relationenschemata in 2. und 3. Normalform (2NF, 3NF)

Bei der Bildung eines RS(DB) in 2NF oder 3NF geht es darum, Redundanzen, die bei einem RS(DB), das in 1NF vorliegt, noch auftreten können, zu vermeiden.

BSP1:ERD für Rechnungssystem für Warenvertrieb an einer Scanner-Kasse



R_2: wird umgesetzt mit

Dieses ERD kann, da es nur aus atomaren Datentypen besteht, in ein RS(DB), das in 1NF ist, umgesetzt werden.

T_1: Rechnung RS(T_1) : PRIK(T_1) = RechNr
T_2: Artikel RS(T_2) : PRIK(T_2) = ArtNr
T_3: Rechnungsposition (für den Beziehungstyp "enthält")

RS(T_3)

A_i	ArtNr	RechNr	Menge	Preis/ME	Wert
d_i	int	int	int	decimal(p,2)	decimal(p,2)
W_i	TPRIK/FKEY |	TPRIK/FKEY	>0		
	PRIK				

Anm: TPRIK := Teilschlüssel. Der PRIK von RS(T_3) ist der Tupel, der aus den Attributen (ArtNr, RechNr) gebildet wird.

Def1: (2NF): Ein Relationenschema RS(T) liegt in 2NF vor, wenn:
(1) RS(T) ist in 1NF und

(2) Alle Attribute A_j von RS(T), die nicht Schlüsselattribute sind, sind vom Schlüsselattribut voll abhängig.

Folgerung: $RS(T_3)$ ist nicht in 2NF. Das Attribut A_j = Preis/ME ist vom Schlüsselattribut (ArtNr, RechNr) nicht voll abhängig. A_j ist nur vom TPRIK = ArtNr abhängig.

1) Verbesserung $RS(T_3')$, das in 2NF ist:

A_i	ArtNr	RechNr	Menge	Wert
d_i	int	int	int	decimal(p,2)
w_i	TPRIK | TPRIK		>0	
	PRIK			

Das Attribut Menge hängt von (ArtNr, RechNr) ab.
Das Attribut Wert hängt von (ArtNr, RechNr) ab.

(Wert = Menge*Preis/ME)

2a) Da das Attribut Preis/ME verwaltet werden muss, muss es dem Relationenschema von Artikel hinzugefügt werden:
$RS(T_2')$

A_i	ArtNr	ArtBez	MWSt_Art	Barcode	Preis / ME
d_i	int	char(n)	int	int	decimal(p,2)
w_i	PRIK		{A, B}		>= 0

Resultat: $RS(DB) = RS(T_1) \cup RS(T_2') \cup RS(T_3')$ in 2NF

2b) Das Attribut Preis/ME wird Tag/Zeitgenau in einer eigenen Tabelle verwaltet.

$RS(T_4)_{(V1)}$

A_i	ArtNr	Tag	Preis / ME
d_i	int	date	decimal(p,2)
w_i	TPRIK/FKEY | TPRIK PRIK		≥ 0
	PRIK		

$RS(T_4)_{(V2)}$

(mehrere Preisänderungen / Tag können verwaltet werden.)

A_i	Preis_ID	ArtNr	Tag	Preis / ME	AbZeit
d_i	int	int	date	decimal(p,2)	int
w_i	PRIK	FKEY		≥ 0	

Resultat: $RS(DB) = RS(T_1) \cup RS(T_2) \cup RS(T_3) \cup RS(T_4)_{Vi}$ ist in 2NF (für i = 1,2).

Für die weitere Betrachtung ist ein RS(DB) in 2NF gegeben. In Hinsicht auf Redundanzen kann es noch Redundanzen zwischen abhängigen Attributen geben.

BSP :

a) Redundanzen aufgrund von Berechnungern: z.B. In RS(T_1) :
SUMME_NETTO = SUMME_BRUTTO – SUMMWSTA – SUMMWSTB.
b) Redundanzen folgender Art:

BSP: Rechnungssysteme mit nicht-anonymen Kunden

RS(T_1'):

A_i	RechNr	RechDat	SUMME_BRUTTO	...	K_Nr	K_PLZ	K_Ort	K_Str
d_i	int	date	Decimal(p,2)	...	int	int	char(n)	char(n)
W_i	PRIK			...	FKEY	{1000, 99999}		

Die Attribute K_Plz, K_Ort, K_Str sind von dem Attribut K_Nr abhängig.

Def2: Ein Relationenschema RS(T) liegt in 3NF vor, wenn
(1) RS(T) liegt in 2NF vor und
(2) alle Nichtschlüsselattribute sind untereinander nicht transitiv abhängig.

Transitive Abhängigkeit bedeutet, dass es ein Nichtschlüsselattribut gibt, von dem andere Nichtschlüsselattribute abhängig sind-

BSP: In RS(T_1') war K_Nr das Attribut, von dem die Attribute K_PLZ,...,K_Str abhängig waren.

Verbeserung: RS(T_1'') enthält als einziges kundenbezogenes Attribut K_Nr.
RS(T_5) enthält als Kundentabelle K_Nr aks PRIK und u.a. Die Attribute K_Plz,...,K_Str

=> RS(DB) = RS(T_1'') $\cup$ RS(T_5) $\cup$ RS(T_3') $\cup$ RS(T_4)$_{Vi}$ liegt in 3NF vor.

Anm.: Neben 1NF, 2NF, 3NF gibt es in der DB-Literatur weitere Normalformen, die auf der 3NF aufbauen:
a) Boyce-Codd NF
b) 4NF, 5NF

Vgl z.B. : M.Schubert : "Datenbanken – Theorie, Entwurf und Programmierung relationaler Datenbanken", Stuttgart, Wiesbaden (Teubner), 2004.

6. Objektorientierte Datenbanksysteme (OODBS)

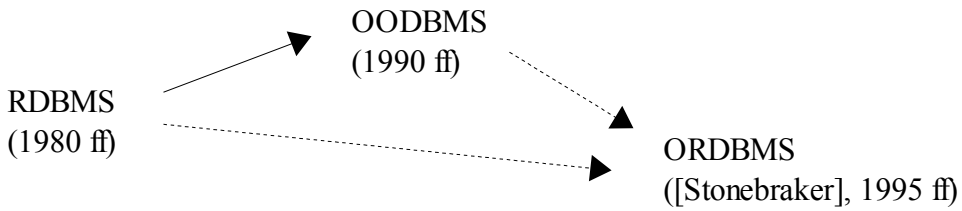
Warum OODBS?

RDBS lassen gewisse Wünsche der Softwareentwicklung offen:

(1) (1-n)-Beziehungstypen können i. d. R. nicht direkt in ein Relationenschema umgesetzt

werden, man muss als Zwischenschritt den Weg der Normalisierung gehen.
Wunsch: Die Data-Dictionary-Komponente eines neuen DBMS-Typs sollte bestimmte Kollektionstypen (Listen, Mengen, ...) enthalten.

(2) Objektorientierte Programmiersprachen (C++, Java), mit denen auf RDB zugegriffen werden kann, bieten ein Klassenkonzept mit Kapselung der Attribute und der Vererbungseingenschaft, solche Vorteile fehlen im Datenmodell von RDB.



ORDBMS := Objektrelationale DBMS
ORDBMS übernehmen gewisse (nicht alle) Wünsche z.B. Mengen (aus(1)), das Klassenkonzept (aus(2)).
Produkte, die ORDBMS-Eigenschaften haben: Oracle (≥ 8i), Informix-Universal-Server, DB2 (neuere Versionen), ...

Beispiel OODBMS-Produkt im Praktikum: (POET) (Fast Objects)

6.1. Anforderungsprofil an ein OODBMS

Ein OODBMS soll gemäß OODBMS-Manifesto ([Atkinson at.al. 1990]) folgende Anforderungen erfüllen:

(1) Das OODBMS soll die allgemeinen Aufgaben eines DBMS erfüllen.
(2) Es soll über ein objektorientiertes Datenmodell verfügen.

Im einzelnen sind das die folgenden Aufgaben:

(1.1) Persistenz
(1.2) Sekundärspeicherverwaltung
(1.3) Multiuserbetrieb (Transaktionskonzept u. ä.)
(1.4) Abfragesprache
(1.5) Datensicherung / Recovery
(1.3/1.5) DBMS-Sicherheit (Rechteverwaltung)

(2.1) Klasse
(2.2) Objektidentität (OID)
(2.3) Abstrakte Datentypen (ADT)
(2.4) Kapselung
(2.5) Vererbung
(2.6) Überladen / Überschreiben von Methoden, spätes Binden

Anmerkungen zu einzelnen Anforderungen:

zu (2.1): Das Data-Dictionary ermöglicht die Definition von Klassen im Umfang, wie es bei

objektorientierten Programmiersprachen möglich ist (z.B.: vordefinierte komplexe Datentypen können bei Attributen verwendet werden, Methoden werden mit Signaturen definiert).

Zu (2.2): Pro persistenter Instanz einer Klasse wird eine Objektidentität (OID) vergeben. Die OID wird vom OODBMS immer nur einmal vergeben, sie ist insbesondere invariant unter UPDATE- und DELETE-Operationen. Die OID, die man sich als eine große Integerzahl vorstellen kann, ist nur dem System bekannt. Man kann für die Anwendungsprogrammierung Surrogate definieren. Das sind Objektnamen, die objekt- eindeutig vergeben werden. Objektnamen entsprechen dann bijektiv den OID der zugehörigen Klasse.

zu (2.3): vgl. Kap 6.2 (nächste Woche)

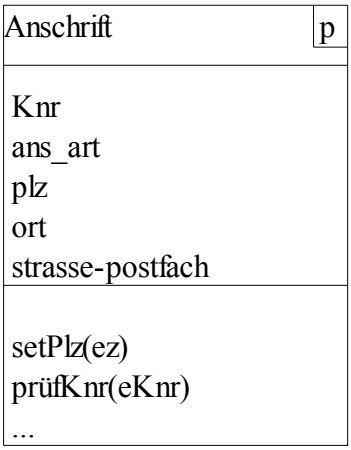
zu (2.4): Die Kapselung von Attributen kann bei OODBS zur Prüfung von Integritätsbedingungen verwendet werden. Attribute, die Integritätsbedingungen unterliegen, werden als private deklariert, dann kann die Integrität bei den schreibenden Methoden auf das entsprechende Attribut gesichert werden.

zu (2.5): Die Data-Dictionary gestaltet die Implementation der Vererbungsbeziehung zwischen Klassen.

zu (2.6): Überladen / Überschreiben von Methoden ist im Data-Dictionary eines OODBS genauso wie in einer objektorientierten Programmiersprachen (OOP) möglich.

Bsp.:

Integritätsprüfung des gekapselten Attributs plz der persistenten Klasse Anschrift mittels der set-Methode setPlz ( ).



02.02.06

6.2. Abstrakte Datentypen (ADT)

Abstrakte Datentypen beschreiben einen Formalismus, mit dem aus einer Menge einfacher Datentypen und einer Menge von Konstruktoren höhere Datentypen definiert werden können. Dieser Formalismus ist programmiersprachenunabhängig.
Das Konzept der ADT gehört innerhalb des Software Engineerings zu den Design-Methoden. ADT gestatten Design-Überlegungen durchzuführen, ohne an konkrete Implementationen in einer konkreten Programmiersprache verbunden zu sein. Weiterhin besteht die Möglichkeit (1 : n)-Beziehungstypen direkt in ADT zu übersetzen. Dadurch kann der Normalisierungsprozess (1NF,...) ersetzt werden.

Bsp.: Eine Entitätenmenge von Zeichenketten z soll verwaltet werden.
Man definiert hierzu einen programmiersprachenunabhängigen Datentyp string.
Dieser Datentyp string kann in Programmiersprachen unterschiedlich implementiert werden.
(1) in Java: String z;
(2) in Java: StringBuffer z;
(3) in C: char z[n]; mit: n konstante Länge und: z[n-1]= ' \0 '

Die Theorie der abstrakten Datentypen besteht aus folgenden Elementen:

I) Für das ADT-Modell wird eine Menge DT von einfachen Datentypen festgelegt:
DT = {int, float, string, char, decimal, boolean }
int: Datentyp für ganze Zahlen $z \in \mathbb{Z}$.
float: Datentyp für rationale Zahlen $q \in \mathbb{Q}$, die als dezimale Gleitpunktzahlen dargestellt werden
string: Datentyp für Zeichenketten.
char: Datentyp für einzelne Zeichen.
decimal: Datentyp für rationale Zahlen $s \in \mathbb{Q}$, die als dezimale Festpunktzahlen dargestellt werden.
boolean: Datentyp für logische Werte

II) Jedem einfachen Datentyp aus DT ist ein maximaler Wertebereich W ("Domäne") zugeordnet.
z.B. für den Datentyp int ist $W = \mathbb{Z}$.

III) Zur Erzeugung von höheren Datentypen stehen die folgenden vier Konstruktoren zur

Verfügung:

a) der Tupel-Konstruktor zur Erzeugung von Tupel-Datentypen: TUPLE OF( )
b) der Set-Konstruktor zur Erzeugung von mengenwertigen Datentypen: SET OF ( )
c) der Listen-Konstruktor zur Erzeugung von listenwertigen Datentypen: LIST OF ( )
d) der Sammlungs-Konstruktor zur Erzeugung von „Bag"-wertigen Datentypen: BAG OF( ) (bag = (engl.) Tasche)

Bsp. 1: Eine Artikeltabelle TA hat folgendes Relationenschema:

RS(TA)

A_i	artnr	artbez	preis
dt_i	int	char(50)	decimal(10,2)
w_i	PRIK		

Dann kann jede Zeile Z der Tabelle TA durch einen Tupel-Datentyp dZ verwaltet werden.

Allgemeine Syntax von TUPLE OF ( ): Ein Tupeldatentyp TU wird in folgender Form definiert:
TU := TUPLE OF(A_1: d_1 ; A_2 : d_2 ; ... ; A_n : d_n). Hierbei sind $A_1,A_2,...,A_n$ Attributnamen und $d_1,d_2,...,d_n$ sind definierte abstrakte Datentypen.

Anm. zu BSP1: dZ kann folgendermaßen definiert werden, da d_1=int , d_2=string , d_3= decimal als einfache Datentypen aus DT bereits definiert sind:
dZ := TUPLE OF (artnr: int ; artbez: string; preis: decimal)

Eigenschaft von ADT-Konstruktoren: Die ADT-Konstruktoren können rekursiv verwendet werden (Hintereinanderschaltung).

Bsp. 2: Die folgendermaßen spezifizierte Klasse Person soll durch einen ADT person beschrieben werden:

a) ADT Datum: Datum:=
   TUPLE OF(tag: int; monat: int; jahr: int)
b) ADT person:
   person:= TUPLE OF (name: string; vnam: string;
   gebdat: Datum; plz: int; ort: string)

IV) Konstruktion von Mengen

Def1: Eine endliche Menge M ist eine Sammlung von Objekten $a_1,...,a_m$, die untereinander verschieden sind.
Notation: Eine solche Menge M wird notiert als M = { $a_1,...,a_m$}

Bsp.1 : Die Klasse Person' entsteht aus der Klasse Person, indem das Attribut vnam durch das Attribut vseq (Vornamen-Sequenz) ersetzt wird, womit alle Personen beschrieben werden können, die mehrere Vornamen haben.

Definition des Datentyps dvseq für das Attribut vseq als ADT:
dvseq := SET OF (vnam: string)
Damit ist der ADT zur Klasse Person' definiert durch: person1 := TUPLE OF(name:string;

vseq: dvseq; gebdat: Datum; plz: int; ort: string)

Bsp.2 : Definition eines ADT zur Verwaltung von Messreihen:
Jede Messreihe besteht aus Messpunkten. Ein Messpunkt hat einen Zeitpunkt t und einen (z.B. ganzzahligen) Messwert m.

a) ADT für Messpunkt dMp := TUPLE OF (t: float; m: int)
b) ADT für eine Messreihe dMr := SET OF (Mp: TUPLE OF (t: float ; m: int))

Übung: Gegeben ist eine Tabelle T mit RS(T) = { $(A_1,d_1,W_1),...,(A_m,d_m,W_m)$ } mit $d_1,...,d_m \in DT$, mit W_1=PRIK.
Bestimmen Sie einen ADT für T. Tip: Eine Tabelle ist eine Sammlung von Zeilen

V) Wertebereich von Tupel-Datentypen

Gegeben: Attribte $a_1,...a_m$ mit Datentypen $d_1,...,d_m$ (alle $d_i \in DT, 1 \le i \le m$) haben die Domänen $W_1,...,W_m$

Gesucht: Die Domäne W_{TU} des ADT TU := TUPLE OF (a_1:d_1 ; a_2 : d_2 ; ... ; a_m: d_m)

Es gilt: Die Domäne W_{TU} ist das karthesische Produkt der Mengen W_1, ...,W_m:
$W_{TU} = W_1 \times W_2 \times ... \times W_m$

Übung: Verifizieren Sie diese Feststellung für den ADT dMp, wenn die Domänen
W_t = { 1.5 , 1.7 , 1.9, 2.0, 2.1, 2.5 }
W_m = {-2, -1, 0, 1, 2 }
gegeben sind.

VI) Wertebereich von Mengen-Datentypen:
Gegeben: Ein Attribut A_1 mit Datentyp d_1, dessen Domäne W_1 ist.
Gesucht: Die Domäne des Datentyps dS := SET OF (A_1: d_1)

Bsp.: Das Attribut vseq mit Datentyp dvseq := SET OF (vnam: string). Das Attribut vnam hat die Beispiel-Domäne W_{vnam} = { "A", "B", "C", "D"}

Gesucht: Die Domäne W_{dvseq}. Welche Teilmengen aus W_{vnam} können insgesamt gebildet werden?

W_{dvseq} = Menge aller Teilmengen von W_{vnam}

Behauptung: Für die Mächtigkeit von W_{dvseq} gilt: $|W_{dvseq}| = 2^n$, wenn $n = |W_{vnam}|$ ist.

0-elementige Teilmenge: {} : leere Menge	: 1
1-elementige Teilmengen: {"A"}, {"B"}, {"C"}, {"D"}	: 4
2-elementige Teilmengen:	: 6
3-elementige Teilmengen:	: 4
4-elementige Teilmengen: W_{vnam}	: 1
	$16 = 2^4$

Def. : Gegeben ist eine Menge M. Dann heißt die Menge P(M) aller Teilmengen von M die Potenzmenge von M.

Satz : Wenn |M| = n , $n \in \mathbb{N} \cup \{0\}$, dann gilt $|P(M)| = 2^n$

Bew.: a) Die Anzahl, um aus einer Menge von n Elementen k Elemente zu ziehen, ist $\binom{n}{k} = \frac{(n!)}{(k! \cdot (n-k)!)}$

Insbesondere gilt:

Anzahl der 0-elementigen Teilmenge : 1 = $\binom{n}{0}$

Anzahl der 1-elementigen Teilmenge : n = $\binom{n}{1}$

Anzahl der 2-elementigen Teilmenge : $\binom{n}{2}$

Anzahl der k-elementigen Teilmenge : $\binom{n}{k}$

...

Anzahl der (n-1)-elementigen Teilmenge : n = $\binom{n}{n-1}$

Anzahl der n-elementigen Teilmenge : 1 = $\binom{n}{n}$

$$P(M) = 1 + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} = \sum_{k=0}^{n} \binom{n}{k} \cdot 1^{(n-k)} \cdot 1^{k}$$

$= (1+1)^n = 2^n$

=> Bin. Lehrsatz

Bin. Lehrsatz:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot a^{(n-k)} \cdot b^{k}$$

Ergebnis: Die Domäne des Set-Datentyps dS ist $W_{ds} = P(W_1)$ und es gilt:

$|P(M)| = 2^A$, wenn $A = |W_1|$ ist.

Übung: Gegeben: A_1 mit d_1 = int und $W_1 = \{1,2\}$
A_2 mit d_2 = char und $W_2 = \{a,b,c\}$
A_3 mit d_3 := TUPLE OF ($A_1 : d_1$; $A_2 : d_2$)
A_4 mit d_4 := SET OF ($A_3 : d_3$)

Bestimmen Sie W_3 (Domäne von A_3) und $|W_3|$
Bestimmen Sie W_4 (Domäne von A_4) und $|W_4|$

VII) Der Konstruktor: LIST OF ( )

Mit dem LIST OF ( ) - Konstruktor wird ein allgemeiner Listen-Datentyp erzeugt, der unabhängig von möglichen Implementationen definiert wird.

Bsp.: Listen-Datentyp-Implementationen:

a) in Java: List l_1; l_1 = new LinkedList ( ); (Implementation als dynamisch verkettete Liste)
b) in C: struct dvkett
{ char wort [z_1]; /* Knotenfärbung: */
int wz; /* wort (String) und Zahl */
dvkett *vor; /* Zeiger auf Vorgänger */
dvkett *nach; /* Zeiger auf Nachfolger */
} v_1, v_2, v_3;

Def.: Ist ein Attribut a_1 mit bereits definiertem Datentyp d_1 gegeben, dann wird eine Liste, die aus Knoten besteht, die alle durch das Paar (a_1, d_1) gefärbt sind, durch folgenden ADT verwaltet:
l := LIST OF (a_1 : d_1)

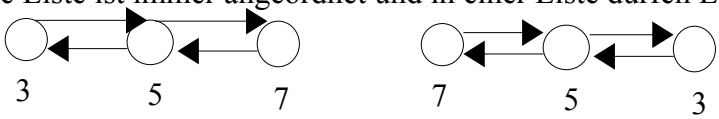
Anm.: Bei der Java-Implementation des ADT LIST OF ( ) ist in jedem Listenaufbau-Modul (= überall dort, wo mit add( ) neue Knoten der Liste hinzugeführt werden) darauf zu achten, daß nur Knoten gleichen Datentyps in die Liste eingekettet werden.

Bsp.1: Liste von Messwerten x, die vom Typ double sind:
ADT: l_1 := LIST OF (x, double)

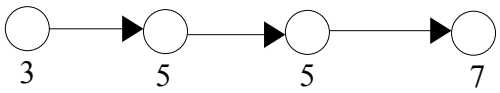
Bsp.2: Liste l_2 von Tupelwerten (wort, wz), wobei wort vom Typ String und wz vom Typ int ist:
ADT: l_2 := LIST OF (t: TUPEL OF (wort: String; wz: int))

Anm.: Unterschiede zwischen dem Mengen- und Listen-Datentyp:
a) Eine Menge hat keine Anordnung und in einer Menge dürfen keine Elemente mehrfach vorkommen. (Bsp.: (1) Es gibt: {3, 5, 7} = {7, 5, 3}; (2) Die Sammlung <3, 5, 5, 7> ist keine Menge)
b) Eine Liste ist immer angeordnet und in einer Liste dürfen Elemente mehrfach vorkommen

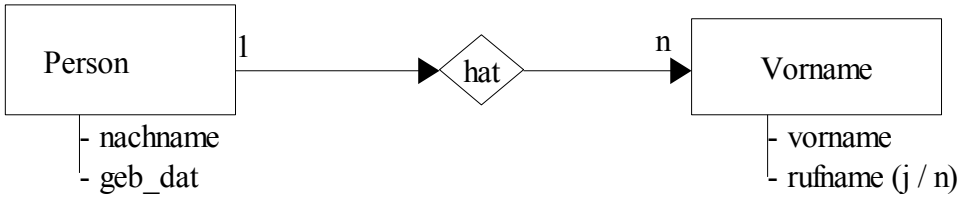
(Bsp.: (1)



Bsp.: (2)

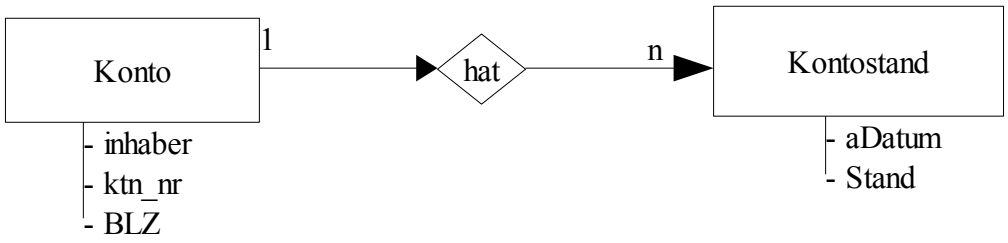


Bsp.3:



2 Möglichkeiten für die Abbildung der (1:n)- Beziehung, als SET und als List. SET ist möglich, da es keine Person gibt, die einen gleichen Datentyp Z, mit dem die gesamte (1:n)-Beziehung abgebildet wird:
Z := TUPLE OF (nachname: String; geb_dat: date; vnamliste: LIST OF (vk: X) mit X := TUPLE OF (vorname: String; rufname: boolean) ∟> Vornamensknoten

Bsp.4:



Übung: Setzen Sie das ERD in einen ADT um !!!

VIII Der Konstruktor BAG OF ( )

Der abstrakte Datentyp BAG dient zur Verwaltung von Datensammlungen, die durch die Eigenschaften angeordnungslos (im Gegensatz zu Listen) und Mehrdeutigkeit der Elemente (im Gegensatz zu Mengen) gekennzeichnet sind. BAGs sind Datensammlungen, die weder Listen noch Mengen sind.

Bsp.: Die Datensammlung <3, 5, 5, 7>, die ohne Anordnung betrachtet wird ist ein BAG: Man notiert BAGs in der Form $[a_1, a_2, ..., a_n]$. Es gilt: [3, 5, 5, 7] = [5, 3, 5, 7] = [7, 5, 3, 5] = ...

Anm.: Notation von BAGs: Wenn Elemente a_i in einem BAG mit Vielfachheit m_i vorkommen, kann diese Sammlung gleicher Elemente auch durch den Tupel (a_i, m_i) notiert werden.

Bsp.: a) Gegeben: BAG : z = [a, c, b, c, a] = [a, a, b, c, c] = [(a, 2), (b, 1), (c, 2)] ={ (a, 5), (b, 2), (c, 2) }
b) w = [(a, 2), (b, 2), (c, 2), (a, 3)] = [(a, 5), (b, 2), (c, 2)] ={ (a, 5), (b, 2), (c, 2) }
↑——————↑
unvollständig zusammengefasst

Anm.: Vollständig zusammengefasste BAGs in Tupelnotation können als Mengen dargestellt werden:
$w \subseteq M \times \mathbb{N}$ M = {a, b, c} Menge der einfachen Werte, aus denen der BAG gebildet wird.

Def.: Gegeben ist ein Attribut a_1 mit dem Datentyp d_1, dann wird die BAG-Datensammlung, deren Elemente durch das Paar $(a_1 : d_1)$ bestimmt sind, durch den folgenden ADT b verwaltet:
b := BAG OF $(a_1 : d_1)$

Bsp.1: Die Datensammlung [3, 5, 5, 7] ist ein Wert von folgendem Datentyp:
b := BAG OF (zahl : int)

Bsp.2: a) Die Datensammlung [(a, 5), (b, 2), (c, 2)] ist vom Typ:
c := BAG OF (x : char)
b) Wenn diese Datensammlung als Menge { (a, 5), (b, 2), (c, 2) } repräsentiert wird ("Multimenge"), dann kann sie auch durch folgenden SET-ADT dargestellt werden:
f := SET OF (tu : TUPEL OF (x : char; m : int))

6.3. Implementation persistenter Klassen

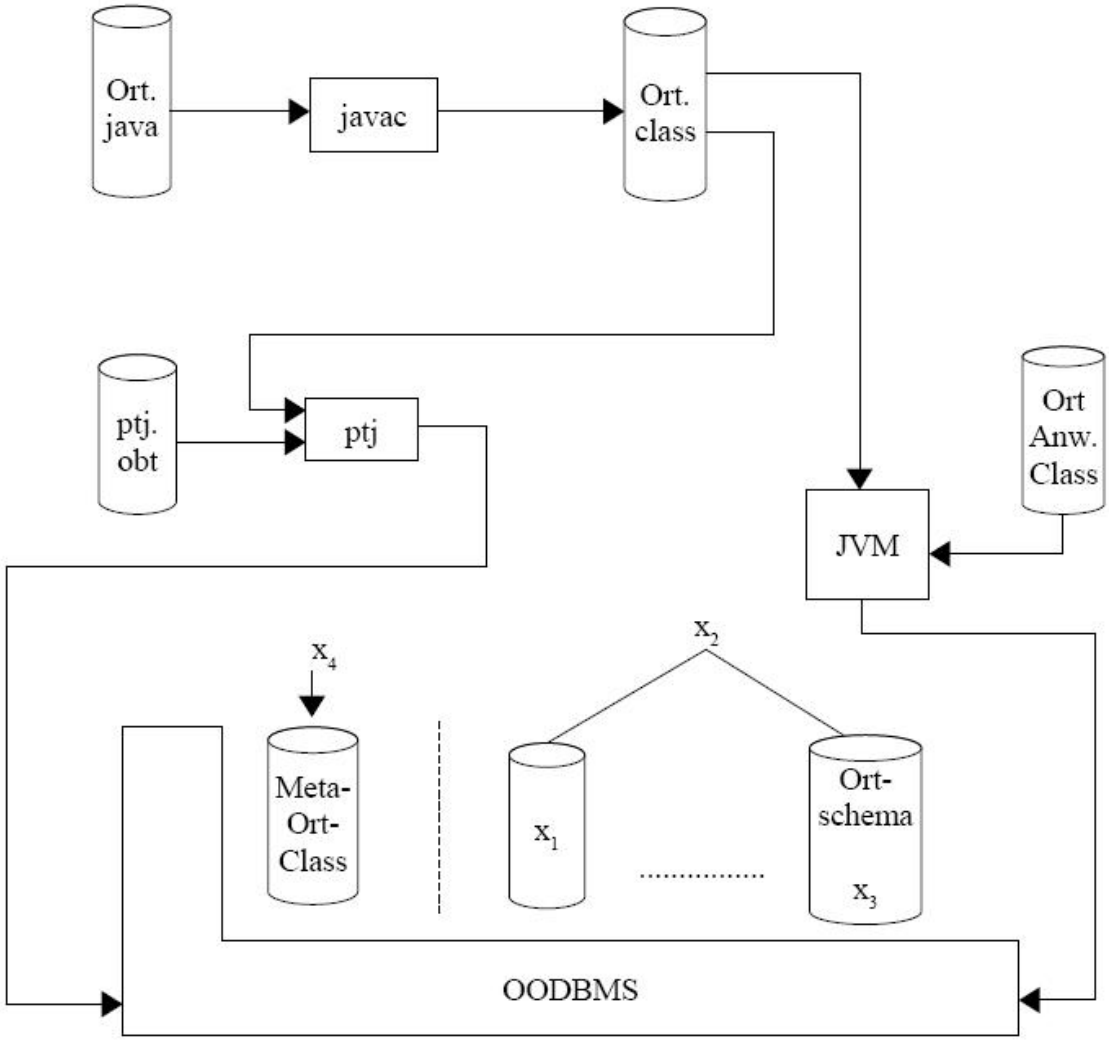
1) Anlegen eines DB-Schemas und eintragen einer persistenten Klasse in das DB-Schema

Bsp.: Persistente Klasse Ort:

```
class Ort {
        String ort_name;
        int plz;
        int anz_einwohner;
    /* Konstrukt. Methoden */
}
```

a) Kompilieren einer persistenten Klasse
b) ptj-Programm zum Anlegen des DB-Schemas und zum Eintragen einer persistenten Klasse (hier: Klasse Ort) in das DB-Schema

Abbildung zu a) & b)



x_1 => Menge der OODB-Nutzdaten: File = DB-Name
(→ public static final String dbname = "poet//LOCAL/../Ortp"
→ Eintrag in Label [databases \ Ortp])
*) in Java-Anwendungsprogramm, das auf die DB zugreift

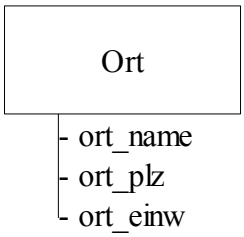
x_2 => Nutzdaten und bestimmte Metadaten sind abgelegt im DB-Verzeichnis
a) bestimmt in ptj.obt: defaultDatabasePath = /home/DBPRAKxx
b) schema-Name in ptj.obt: Label [schemata\Ortschema]

x_3 => Das Label [classes\Ort] bestimmt, daß die Klasse Ort persistent gespeichert wird
→ persistent = true
(Für jede persistente Klasse muß ein solcher Labeleintrag erstellt werden (in ptj.obt)).

x_4 => Diese Kopie der Metadaten befindet sich im User-Verzeichnis: /home/DBPRAKxx

2) Definition einer persisenten Klasse:

Geg.: Ein ERD, es enthält mindestens eine Entitätenmenge mit Attributen.
Bsp.: Entitätenmenge



Entitätenmengen werde in einem OODBS durch persistente Klassen implementiert.
In Poet werden persistente Klassen wie gewöhnliche Klassen definiert. Zur Klassendefinition gehören die Attribute, die gekapselt werden, ihre get( )- und set( )-Methoden, sowie ein Konstruktor, wenn man nicht mit dem Standardkonstruktor arbeiten möchte.

Bsp.: Definition der persistenten Klasse Ort durch:

```
class Ort {     private String ort_name;
        private int ort_plz;
        private int ort_einw;
        ...// zugehörige get( )- und set( )-Methoden
        ...// [Konstruktor]
}
```

Anm.: Eine Klasse bekommt in dem POET-OODB ihre Persistenzeigenschaft durch Eintragung ins Data-Dictionary (Schema) der OODB. Hierzu dient das Programm ptj (Aufruf des ptj Programm, siehe Arbeitsblatt Nr. 5). Notwendig ist, dass vor Start von ptj ein entsprechender Label-Eintrag in derDatei ptj.opt vorgenommen wird.
(Zum Aufbau derptj.opt-Datei vgl. Praktikumsanleitung Teil 1).
Bsp.: Labeleintrag für die Klasse Ort in ptj.opt: [classes\Ort]
persistent=true
[Die nachfolgenden Punkte sind Bestandteile es Java-Anwendungsprogramms, mit dem lesend und schreibend auf die OODB zugegriffen wird. Bsp.: (s. Bild Vorlesung letzte Woche):
Anwendungsprogram: Ortanw.class]

3) Das Öffnen und Schließen einer Datenbank

(1) Der Datenbankname, der im ptj.opt-Label [databases\...] festgelegt wurde und durch das ptj-Programm in das DB-Schemaeingetragen wird, muss im Anwendungsprogramm als Konstante definiert sein.
Bsp.: Label-Eintrag: [databases\Ortp]
DB-Name als globale Konstante des Anwendungsprogramms:
public static final String dbname = "poet://LOCAL/Ortp";

(2) Das Öffnen:
Eine Instanz der Klasse Database anlegen:
Database db1;
db1=new Database();
Öffnen: db1.open(dbname, Database.OPEN_READ_WRITE);
^^Modus des Öffnens (hier: lesend&schreibend).

(3) Das Schließen: db1.close( );
Anm.:
1) Alle Transaktionen, die auf die DB zugreifen, müssen abgeschlossen sein, bevor man schließen kann.
2) Nur geöffnete DB können geschlossen werden: if (db1.isOpen( )) db1.close( );

4) Transaktionen
Transaktionen sind Folgen von Datenbankzugriffen, die entweder alle gemeinsam ausgeführt werden, oder es wird kein Datenbankzugriff ausgeführt.
(1) Transaktion anlegen: Transaction tx1=new Transaction( );
(2) Transaktion starten: tx1.begin( ); (Transaktionsstart i.d.R. vor try-catch-Anweisung zum DB-Zugriff
(3) Transaktion regulär beenden (<=> alle Änderungen werden in der DB wirksam):
tx1.commit( );
(Das commit( ) wird im try-Block nach Ausführung des DB Zugriffs programmiert).
(4) Transaktions-Abbruch (<=> DB bleibt in ihrem alten Zustand) : tx1.abort();
(abort( ) wird im catch( )-Block programmiert).

5) Das einfache Einfügen von persistenten Instanzen in die DB (= INSERT in SQL)
In OODB gibt es zwei Arten des Einfügens:
i) einfaches Einfügen.
ii) Einfügen von Objekten mit Objektnamen(Objektnamen sind Surrogate für OIDs).
Das einfache Einfügen wird mit der Mehthode makePersistent( ) ausgeführt:
db1.makePersistent(x); wobei x eine Instanz einer persistenten Klasse ist.

Bsp.: Einfügen einer Instanz der Klasse Ort:

```
Ort ox = new Ort(eo_name, eo_plz, eo_einw);
tx1.begin( );
try
{ db1.makePersistent(ox);
tx1.commit( );
}
catch (POETRuntimeException e)
{ System.out.println("INSERT-Fehler für:"+ox.getOname( )+"Fehlerart:"+e.toString( ) );
tx1.abort( );
}
```

6) Lesen von Instanzen einer persistenten Klasse:
Der „Select"-Befehl zum Lesen in einer OODB gehört der Object-Query-Laguage (OQL)

an.
OQL ist mit dem SQL verwandt. Es kennt über SQL hinausgehend Ausdrücke, die mit Methoden der persistenten Klasse gebildet werden können.

„Allg." OQL-SELECT-Syntax:
SELECT zlI FROM klassennameExtent AS zlI WHERE whereklausel ORDER BY orderbyKlausel

Hierbei ist:
a) zlI: die zu lesende Instanz.
b) klassennamenExtent: die Menge aller persistenten Objekte einer persistenten Klasse (z.B. OrtExtent).
c) whereKlausel: enthält logische Verknüpfungen von Vergleichsausdrücken. Vergleichsausdrücke enthalten Vergleichsoperatoren:

i) für numerische Datentypen: =, <, >, >=, <=
ii) für Zeichenketten-Datentypen: =, LIKE
d) orderbyKlausel: (wie beim SQL-Select): mit ASC (aufsteigend) bzw. mit DESC (absteigend).

Anm.: Im Unterschied zu SQL ist in OQL die WHERE-Klausel notwendig.

Bsp.1: SELECT ox FROM OrtExtent AS ox WHERE ox.getOrtplz( ) >= 50000 AND ox.getOrtplz( ) < 60000 ORDER BY ox.getOrtsname( ) ASC
a)OQL-Select-String definieren:
String a1=new String("SELECT ox from OrtExtent as ox WHERE ox.getOrtsname like 'A%' ");
b) OQL-Query-Objekt erstellen: OQLQuery q1=new OQLQuery(a1);
c) SELECT ausführen: Object r1=q1.execute( );
d) Iterierende Verarbeitung des Ergebnisses r1 ist nur dann möglich, wenn r1 eine tatsächlich Ergebnismenge (= Menge von Instanzen der angefragten Klasse) ist. Dieses wird mit folgender Prüfung festgestellt:
if (r1.instanceOf CollectionOfObject) {...}

```
d1) Iterator erstellen: Iterator iter = ( ( CollectionOfObject)r1).select(a1);
d2) Navigieren mit Hilfe des Iterators:
while(iter.hasNext( ) )
{Ort o1=(Ort) iter.next( );
// Weitere Verarbeitung von o1, z.B.
// Zugriff mit den entsprechenden get( )-Metoden auf die Attribute von o1, z.B.
System.out.println("Ort ="+o1.getOrtname( ));
...
}
```

7) Löschen von persistenten Instanzen (= SQL-DELETE)

Löschen: db1.deletePersistent(x);
^die zu löschende Instanz.
Man benötigt eine Instanz x, die tatsächlich in der DB vorliegen muss, um sie löschen zu können. Dies wird durch eine vorhergehende SELECT-Anfrage sichergestellt:

Bsp.:
```
Ort x = (Ort)iter.next( );
db1.deletePersistent(x);
```

Anm.: POET kennt kein "SQL-UPDATE". Um ein Objekt x zu überschreiben, sind folgende Aktionen nötig:
(1) x wird in eine Instanzvariable eingelesen.
(2) x wird gelöscht
(3) z wird mit neuen Attributwerten überschrieben
(4) z wird in die DB eingefügt.

8) Einfügen benannter Objekte

Gegeben ist eine persistente Klasse P mit persistenten Objekten (Instanzen) $o_1, o_2, ..., o_n$

$[o_1, o_2, o_r, ..., o_n]$ = Extent von P

$I_1\ I_2 ... I_r\ I_n$: I_j : Objektidentität von O_j $(1 \le j \le N)$

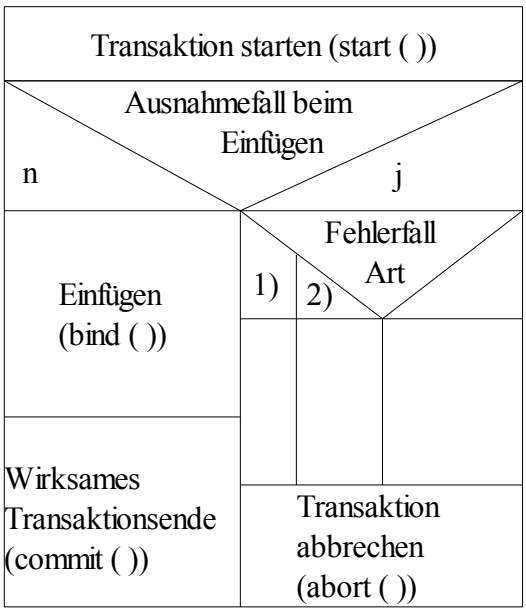
$N_1\ N_2 .. N_r$: N_j : Objektnamen von O_j $(1 \le j \le N)$. Objektnamen sind Surrogate für Objektidentitäten, die vom System hermetisch gekapselt sind.

Im POET-OODBMS sind die Objektnamen Zeichenketten, die als Primärschlüsselwerte umkehrbar eindeutig den persistenten Instanzen beim Erfassen zugeordnet werden. Das hat zur Folge, daß beim Löschen nicht nur die persistente Instanz, sondern auch ihr Objektname gelöscht werden muss.

Einfüge-Anweisung:

```
db.bind(x, N);      /* x = Instanz einer persistenten Klasse (z.B.: bhf als Instanz der
                       Klasse Bahnhof */
                    /* N = String, der den Objektnamen enthält (Der Name muss im
                       Extent der Klasse P eindeutig sein). */
```

Das bind ( ) wird innerhalb einer Transaktion ausgeführt, die mit einer try-catch Anweisung implementiert wird:



Fehlerfall – Arten:

1) RUNTIME-Exception. Diese Fehlerfallart kann bei jedem DB-Zugriff entstehen (z.B.: bei INSERT = makepersistent ( ), bei bind ( ), etc.)

2) ObjectNameNotUniqueException: Dieses ist der Fehlerfall, wenn gegen die Bedingung, daß der Objektname N eindeutig sein soll, verstoßen wird. (Kann nur bei bind ( ) auftreten.)

9) Löschen von benannten Objekten:

a) Löschen der persistenten Instanz x : db.deletePersistent(x);
b) Löschen des Objektnames N : db.unbind(N)

10) Lesen von benannten Objekten:

Der Direktzugriff auf benannte Objekte mit dem Objektnamen wird mit einer besonderen Lesemethode (lookup( )) ausgeführt, die den Objektnamen als Argument hat (ODMG-Standart für Objektorientierte Datenbanken).

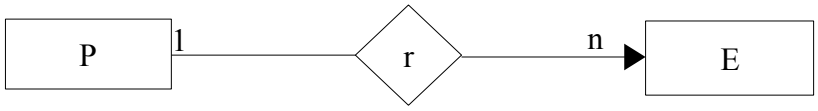
Allgem. Syntax:

x = (P)db.lookup(N); mit: x ist Instanz der persistenten Klasse P.
N ist String mit Objektnamenskandidat.

Bsp.:
```
try{
Bahnhof bhf = (Bahnhof)db.lookup(estr);
}
catch (ObjectNameNotFoundException exn)
{ .... }
```

6.4. Implementation von (1:n)-Beziehungen mit ADT-Konstrukten

Gegeben ist eine (1:n)-Beziehung in einem ERD.



Je nach Anwendungsfall soll die (1:n)-Beziehung r als SET, LISTE oder BAG implementiert werden. Jeder dieser ADT-Konstrukte wird durch einen besonderen POET-ODMG-Kollektionsdatentyp verwaltet: SetOfObject, ListOfObject, BagOfObject.
Die (1:n)-Beziehung wird durch ein Attribut v, das einen dieser Kollektionsdatentypen als Typ hat, bestimmt. Dieser Typ bestimmt dann das Methodeninventar zum Verwalten der Instanzen e von E in der (1:n)-Beziehung (Art des Einkettens, Duplikatkontrolle, Anordnung).

Bsp.: Deklaration von v in E:
(1:n)-Beziehung r ist vom Typ SET => private SetOfObject vs;
(1:n)-Beziehung r ist vom Typ LISTE => private ListOfObject vl;
(1:n)-Beziehung r ist vom Typ BAG => private BagOfObject vb;

Bsp.: P = Auftrag, E = Artikel, (1:n)-Beziehung "enthält" soll als SET implementiert werden
=> Die persistente Klasse Auftrag hat folgende Definition:

```
class Auftrag{
        private int auf_nr;             // einfaches Attribut
        private Date auf_dat;           // einfaches Attribut
```

```
        private String kde_name;      // einfaches Attribut
        private SetOfObject art_set;  /* Attribut zur Verwaltung der (1:n)-Beziehung, in der
                                         die Artikelpositionsinstanzen eingekettet werden */
        ....                          // Konstruktoren und Methoden
    }
```

a) Anlegen einer ADT-Instanz:

Bsp.:

```
art_set = new SetOfObject( );
```

Diese Aktion ist nicht erforderlich, wenn art_set in einer Instanz auf der Klasse Auftrag enthalten ist, die mit dem Standardkonstruktor angelegt wird: auf = new Auftrag( );

b) Get-Methode zum Zugriff auf die ADT-Referenzen:

```
public ADTTyp getVx( )
{ return vx; }
```

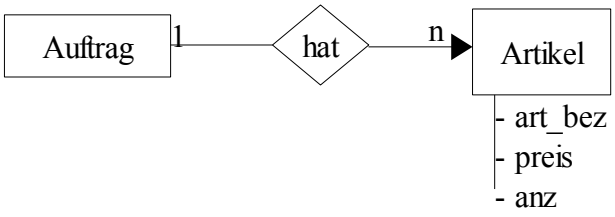
Bsp.:

```
public SetOfObject getArt_set( )
{ return art_set; }
```

c) Ein Element in eine ADT-Datensammlung einfügen:

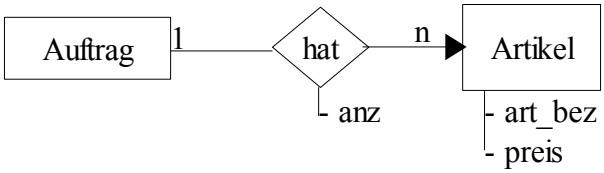
c_1) Datentyp E der Elemente e bestimmen, die in die (1:n)-Beziehung eingekettet werden sollen:

(v1)
```
class Artikel{                // Variante 1
        String art_bez;
        double preis;
        int anz;
}
```



(v2)
```
class Artikel{
        String art_bez;
        double preis;
}

class Artikelposition{        // Variante 2
        Artikel a1;
        int anz;
}
```



c_2) Eine einzukettende Instanz aufbauen:

i) bei (v2): Eine Instanz art_1 vom Typ Artikelposition aus der DB lesen.
ii) Eine Instanz ap_1 vom Typ Artikelposition, die eingekettet werden soll,

erfassen:
ap_1 = new Artikelposition( );
ap_1.setArtikel(art_1);
ap_1.setAnz(eanz);

Anm.: bei: (v1) ist nur die Aktion (i) auszuführen.

c_3) Das Einketten der Instanz:

Das Einketten wird mit der Methode add( ) ausgeführt, die es für jeden ADT-Kollektionsdatentyp gibt:

void add(Object ox)

Bsp.1: Einketten einer Artikelposition innerhalb einer statische Methode:

```
public static void neuArtpos(Auftrag au1, Artikelposition ap1){
        an1.getArtSet( ).add(ap1);
}                    ↑              ↑
                     |              einzukettende Instanz
                     die SetOfObject-Instanz
        ——————————
                ↑
                die der Auftragsinstanz an1 gehört
```

Vor.: Die Instanz an_1 muss vorher mit einem Konstruktor erzeugt worden sein.

Bsp.2: Wie Bsp.1, aber als Methode der Klasse Auftrag:

```
public static void neuArtpos(Artikelposition ap1){
        this.getArtSet( ).add(ap1);
}
```

d) Das Entfernen von Elementen aus einer ADT-Kollektion

Das Entfernen wird mit der ADT-Methode remove( ) ausgeführt:

Signatur: Object remove(Object or)

Bsp.: Löschen einer Artikelposition:

```
Artikelposition apex = au1.getArtSet( ).remove(ap1);
                       ↑                       ↑
                       |                       zu löschende Position
                       gegebene Auftragsinstanz
```

e) Das Anzeigen aller Objekte einer ADT-Kollektion:

e_1) Einen Iterator für eine gegebene Kollektion mit der ADT-Kollektionstyp-Methode iterator( ) anlegen:

```
Iterator it1 =  au1.getArtSet( ).iterator( );
                ↑
```

gegebene Auftragsinstanz

e_2) Navigation durch die Kollektion:

Bsp.:
```
while(it1.hasNext( )){
        Artikelposition ap_neu = (Artikelposition)it1.next( );
        /* Attribute von ap_neu ausgeben */
        .....
}
```

f) Anmerkung zum Insert von P-Instanzen p:

Einfache Attribute von p erfassen
Alle Kollektionselemente e_1, ..., e_n von p einketten (vgl. c) mit add( ) jeweils
Instanz für p (mit makepersistent( ) z.B.)

g) Methoden für ADT-Attribute (für alle ADT-Arten) :

g1) Angabe der Elemente einer Datensammlung:
Methode: int size( );
Bsp.: ListOfObjects x; int m; m = x.size( );
g2) Prüfung: Ist ein Objekt p in einer Sammlung x enthalten?
Methode: boolean contains(Object p);
Bsp.: BagOfObjects x; boolean a; a = x.contains(p);
g3) Prüfung: Ist eine Sammlung leer?
Methode: boolean isEmpty( );
Bsp.: BagOfObjects x; boolean a; a = x.isEmpty( );

h) BAG-Methode: Abfrage, wie oft ein Element p in einem BAG x auftritt:
Methode: int occurrences (Object p);
Bsp.: int m; m = x.occurences(p);
i) LIST-Methoden für das Feststellen der Listenposition eines Elementes (i1) und für den Zugriff auf ein i-tes Listenelement (i2):
(i1) int indexOf(Object p);
(i2) Object get(int i);
Bsp.:
```
ListOfObjects artikelliste;
Artikel a1;      a1 = (Artikel)artikelliste.get(i);
//(für alle i mit   0 ≤ i ≤ artikelliste.size( );   )
```

6.5. Lesen im Extent

Neben dem Lesen mitdem OQL-Select und dem Lesen von benannten Ojekten mit lookup( ) ist das

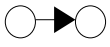
Lesen im Extent die dritte wichtige Methodik zum Lesen von persistenten Instanzen. Das Leseverfahren im Extent besteht aus den folgenden Schritten:

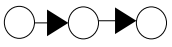
(1) Konstruktion eines Extents: Konstruktoraufruf mit Angabe einer persistenten Klasse:
Extent ex1 = new Extent("Artikel");
(2) Prüfen, ob ein erstes bzw. nächstes Element im Extent vorliegt:
boolean hasNext( );
(3) Positionieren auf das nächste Element im Extent: void advance( );
(4) Lesen des aktuellen Elements aus dem Extent: Object current( );

Bsp.:
```
while ( ex1.hasNext( ) )
{
Artikel a1 = (Artikel)ex1.current( );
//Verarbeitung der Instanz
...
//Positionieren auf die nächste Instanz
ex1.advance( );
}
```

6.6. Wertebereiche des LIST OF(...)- und des BAG OF(...)-ADT

6.6.1 Wertebereich des ADT LIST OF(...)

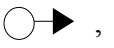
Gegeben:
a) Ein Attribut x mit Datentyp A und Wertebereich W
b) Ein LIST OF(...)-ADT, der folgendermaßen konstruiert wird:
Z = LIST OF(x: A)
Gesucht: Wertebereich W_Z, wenn z ein Attribut vom Typ Z ist.
Wesentlich für den Aufbau einer Liste ist der Verkettungsoperator ◦: x ◦ y bedeutet, dass der Knoten y Nachfolgerknoten von x ist. x ◦ y <=> 
x y

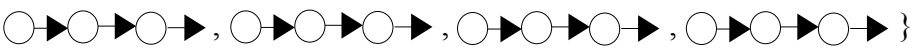
Bsp.1: x ◦ y ◦ z <=> 
x y z

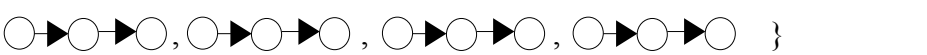
Anm.: Der Verkettungsoperator ist nicht kommutativ:

Ist der Verkettungsoperator ◦ gegeben, dann kann man die Verkettungsmenge von zwei Mengen von Listen definieren:
Def.: Sind L_1 und L_2 zwei Mengen von Listen, dann ist die Verkettungsmenge $L_1 \circ L_2$ definiert als:

$L_1 \circ L_2 = \{ u \circ v \mid u \in L_1 \text{ und } v \in L_2 \}$

Bsp.: L_1 = {  } L_2 = {  }
x y a b w v

A) $L_1 \circ L_2$ = {  }
x y w w y v a b w a b v

B) $L_2 \circ L_1$ = {  }
w x y v x y w a b v a b

Anm.: $L_1 \circ L_2 \neq L_2 \circ L_1$
Def.: ε ist die leere Liste
Def.: Ist eine W eine Menge von Elementen, die als Knoten in einer Liste vom Typ Z auftreten können, dann ist der Wertebereich W_Z aller dieser Listen folgendermaßen induktiv definiert:

a) $L^0 := \{\varepsilon\}$, b) $L^1 = L := \{O \rightarrow | x \in W\}$ (Menge der einelementigen Listen)

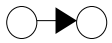
c) $L^i := L \circ L^{(i-1)}$, dann ist $W_Z = \bigcup_{i=0}^{\infty} L^i$ W_Z heisst Kleenesche Hülle von L.

Anm.1: Es gilt L^1 und W sind als Mengen gleichmächtig: $|L^1| = |W|$
(man kann für endliche Mengen W und L^1 zeigen, dass $L^1 \subseteq W$ und $W \subseteq L^1$ ist)

Bsp.: $L^2 = L \circ L = \{ | \; x \in W \text{ und } y \in W \; \}$

$|L^2| = |W|^2$

denn: für x hat man |W| Auswahlmöglichkeiten, für y hat man |W| Auswahlmöglichkeiten, also für

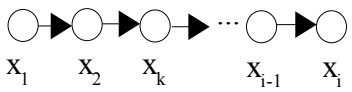


hat man |W| * |W| Auswahlmöglichkeiten.

Ergebnis: Jeder Attributwert z vom Typ Z := LIST OF (a: A), wobei a den Wertebereich W hat, ist aus dem Wertebereich W_Z. D.h. W_Z (die Kleenesche Hülle von L^1 mit $|L^1| = |W|$) ist der Wertebereich von W_Z .
Frage: Wenn die Listenproduktion auf Listen maximaler Länge $m \in \mathbb{N}$ beschränkt wird, wie mächtig ist dann $\bigcup_{i=0}^{m} L^i$ (Beschränkung der Verkettung auf maximal m-elementige Listen)?

Bsp.: $L_3 = L^0 \cup L^1 \cup L^2$ $L^0 = \{ \varepsilon \}$, $|L_1| = |W|$, $|L^2| = |W|^2$
$|L_3| = |L^0| + |L^1| + |L^2|$ (denn $L^{(i-1)} \cap L^i = \{\}$ für alle $1 \leq i \leq m$) = (*)
$|L_3| = 1 + |W| + |W|^2$ ($1 = |W|^0$)

Antwort: $|L_m| = |L^0| + |L^1| + |L^2| + ... + |L^{m-1}| + |L^m| = |W|^0 + |W| + |W|^2 + ... + |W|^{m-1} + |W|^m$
^ (*) ^ $|L^i| = |W|^i$ (<- für jeden Knoten x_k in



hat man |W| Auswahlmöglichkeiten).

$$|L_m| = 1 + |W| + |W|^2 + ... + |W|^m = \sum_{i=0}^{m} |W|^i = \frac{(|W|^{(m+1)} - 1)}{(|W| - 1)} \quad (|W| \neq 1)$$

Bsp.: W = {a,b,c} Ges: $|L_3|$
$|L_3| = 1+3+3^2+3^3 = 40$
$|L_3| = \frac{(3^4 - 1)}{(3-1)} = \frac{80}{2} = 40$

Bsp.: $W = A_{26} = \{a,b,c,...,x,y,z\}$

L_3= Menge der Wörter mit maximal 3 Buchstaben

$$|L_3| = \frac{(26^4 - 1)}{25} = \frac{(676^2 - 1)}{25} = 18279$$

6.6.2 Wertebereich des ADT BAG OF (...)

Gegeben: Eine Variable s vom Datentyp Z mit Wertebereich W_s und $|W_s| = k$

Gesucht: Für jede Variable b vom Datentyp B := BAG OF (s: Z) ist der Wertebereich W_b gesucht.

Wb ist die Menge aller Bags (Sammlungen vom Typ BAG) B_r, die aus Elementen von W_s besteht. Ein BAG B_r kann Elemente aus Wz mehrfach enthalten, aber in B_r gibt es keine Anordnung.

Bsp.: $W_s = W_x \times W_y$ Beispiele für Bags B_r, wo die Häufigkeit μ, mit der gleiche Elemente in B_r auftreten können, nicht größer als 2 ist ($\mu \leq 2$).

B_1 = [((1,a), 2), ((1,b),1), ((3,a),2), ((3,b),1)]
B_2 = [((1,b),2), ((2,a),1), ((2,b),2)]
B_3 = [((1,a),1), ((1,b)1), ((3,a),2), ((3,b),1)]
.....

W_b = { [ ], ..., B_1, B_2, B_3, ... }
$W_b = \overline{B_0} \cup \overline{B_1} \cup \overline{B_2} \cup ... \cup \overline{B_6}$ => $6 = |W_s| = k$

$\overline{B_i}$ = Menge der Bags, die aus i Elementen bestehen

Anm.: Bags heißen auch Multimengen, da sie Elemente mit höherer Vielfachheit enthalten können.
Allgem. gilt: $W_b = \{ B_r \mid B_r = [(a_1, n_1), (a_2, n_2), ..., (a_k, n_k)]$, wobei für alle i mit $1 \leq i \leq k$ gilt:
$a_0 \in W_s$ und $n_i \leq \mu \}$, hierbei gilt $k = |W_s|$

Beh.: Für die Mächtigkeit von W_b gilt: $|W_b| = (\mu + 1)^k$

Bew.: Um B_r zu bilden muss folgende Auswahl getroffen werden:
Elemente aus W_s:

a_1	a_2	a_3	....	a_k
0	0	0	...	0
1	1	1	...	1
2	2	2	....	2
3	3	3	...	3
....	....	....	....	....
μ	μ	μ	μ	μ

=> (μ + 1) Möglichkeiten eine Auswahl in Bezug auf a_1 zu treffen
=> (μ + 1) Möglichkeiten eine Auswahl in Bezug auf a_2 zu treffen

.....
=> (μ + 1) Möglichkeiten eine Auswahl in Bezug auf a_k zu treffen

=> insgesamt hat man (μ + 1) * (μ + 1) * ... * (μ + 1) = (μ + 1)^k Möglichkeiten B_r zu bilden
=> $|W_b| = (\mu + 1)^k$

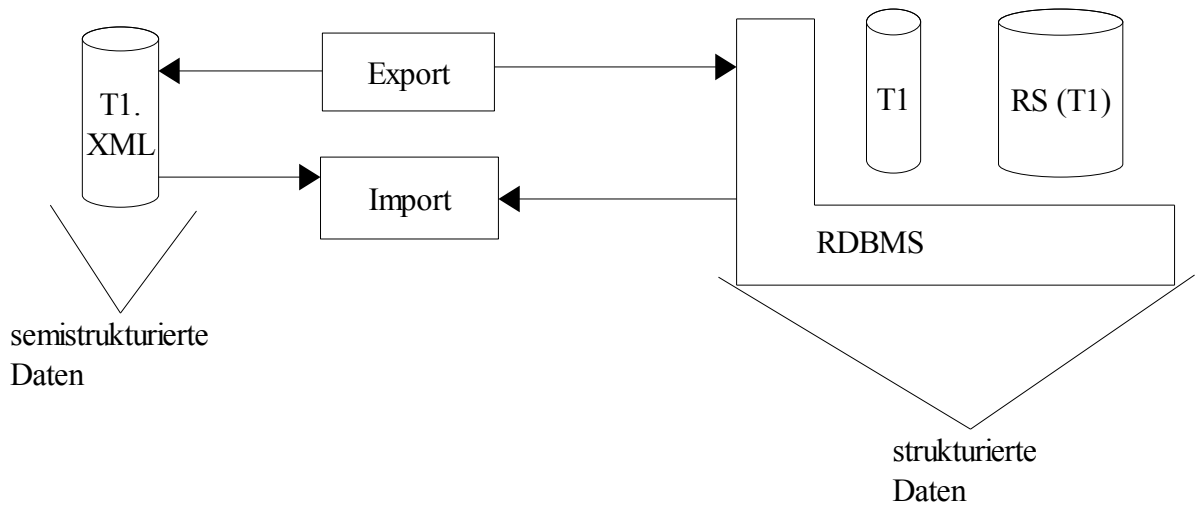
Bsp.: $W_s = W_x$ x W_y = { (1,a), (2,a), (3,a), (1,b), (2,b), (3,b) } => K = 6
Geg.: μ = 2
Ges.: $|W_b| = (\mu + 1)^k = 3^6 = 81 * 9 = 729$

Übung: Gegeben: Variable s mit Wertebereich W_s = {a, b, c}
U := BAG OF (s: char), μ = 2
Gesucht:-Wertebereich Wb alles Bags b vom Typ U.
- $|W_b|$

7. Datenbanken und XML

Überblick:

XML: Extended Markup Language
XML wird im WEB als Standardaustauschformat für Dateien verwendet.



Anmerkung:
a) Alle Nutzdaten in einer Datenbank sind strukturierte Daten. Für jeden Wert w_{ij} einer Tabelle T1 gilt: Er liegt in einer Zeile i und unterliegt einem Attribut A_j. Für das Attribut A_j ist im Relationenschema RS(T1) der Eintrag (A_j, dt(A_j), W_j) hinterlegt. D.h. für den Wert w_{ij}

ist durch A_j die Semantik und durch $dt(A_j)$ der Datentyp bestimmt. Daten heißen strukturiert genau dann, wenn ihre Semantik und ihr Datentyp bestimmt ist. D. h. Daten in einer DB sind strukturierte Daten.

b) XML ist eine Markup-Sprache in deren Dokumente die Nutzdaten durch Markup-Paare (jeweils ein öffnendes und ein schließendes Tag) eingeschlossen sind.
Bsp.: <WERT_ATT$_j$> w_{ij} </WERT_ATT$_j$>
^Tag-Name

^öffnendes Tag ^schließendes Tag

Der Tag-Name bestimmt die Semantik des Werts. Werte haben in XML keine besonderen Datentypen. Werte sind Zeichenfolgen. D. h. Sie sind alle, grob betrachtet, vom Typ String. Daten heißen semistrukturiert genau dann, wenn sie eine Semantik, aber keinen Datentyp haben. D. h. XML-Daten sind semistrukturierte Daten. [Dieses Manko von XML veranlasste das W3C das Konzept von XML-Schema zu entwickeln. In XML-Schema werden Datentypen-Bestimmungen (Schemadaten) für zugeordnete XML-Dokumente hinterlegt.].

Lit.: - Th. Rottach / S. Groß: „XML Kompakt“, Heidelberg, Berlin (Spektrum), 2002, ISBN: 3-8274-1229-7
- [XML-Standard]: http://www.w3c.org/TR/2004/REC-xml-20040204/

Bem.: Markup-Sprachen: neben XML gibt es HTML und SGML.

SGML: Standardized General Markup Language (Vorläufer von XML), ISO-normiert, Nutzdaten stehen wie bei XML in Paaren von Markups (a) , Markups geben den von ihnen eingeschlossenen Zeichenketten eine Bedeutung (Semantik) (b) ; die Struktur von SGML Dokumenten wird in DTD (=: Document Type Definition) beschrieben (c).
((a), (b), (c) wie in XML.
(c) in XML: „Kann“-Kriterium
in SGML „muß“-Kriterium
Für XML gibt es standardisierte Parser.

HTML: Hypertext Markup Language, keine Semantik durch Markups. Die Mehrheit der Markups dient der grafischen Gestaltung des Dokuments. Wenige strukturgliedernde Markups (z.B. <Head>, <Body>). Markupsequenzen dürfen sich überlappen, Markupsequenzen müssen nicht abgeschlossen werden.
Browser enthalten eine HTML-Parser-Komponente. Den HTML-Parsern liegt eine HTML-Norm zugrunde, die als eine implizite DTD angesehen werden kann.

Lit: Ralph Steyer „XML und Java“, entwickler.press (Frankfurt), 2003, ISBN 3-935042-78-7

7.1. XML-Grundlagen

7.1.1 XML-Syntax, Hierarchiemodell, Wohlgeformtheit

Ein XML-Dokument heisst wohlgeformt <=> es erfüllt die Regeln der XML-Syntax.

Die XML-Syntax besteht aus folgenden Regeln:
(R1) Das XML-Dokument besteht aus einem Prolog und einem Dokument-Element
(R2) Das Dokument-Element ist die Wurzel des XML-Dokuments (d.h. das XML-Dokument enthält eine Baumstruktur mit genau einem Wurzelelement und Nutzdaten, die sich an den Blättern des Dokuments befinden. Der Baum kann eine beliebige Verschachtelungstiefe haben).
(R3) Das Dokument-Element ist ein Element.
(R4) Jedes Element kann Kinderelemente sowie Nutzdaten beinhalten (Nutzdaten = Zeichenfolgen).
(R5) Jedes Element enthält ein öffnendes und ein schließendes Tag:
<Elementname> ... </Elementname>
Innerhalb eines solchen Tag-Paares steht ein Teilbaum oder eine Zeichenkette.
(R6) Tags verschiedener Elemente dürfen sich in ihrem Gültigkeitsbereich nicht überlappen.
(R7) Öffnende Tags können (dürfen) beliebig viele oder keine Attribute haben, deren Namen innerhalb des Tags eindeutig sein müssen. Ein Tag mit Attributen hat folgenden Aufbau:
<Elementname att1="..." att2="..." ... attN="...">
(R8) Der Prolog enthält genau eine XML-Deklaration. Er kann darüberhinaus noch eine Dokumenttyp-Deklaration, Kommentare und Processing Instructions beinhalten.
XML-Deklaration: <?xml version="1.0" encoding="ISO-8859-1" ?>

<?: Syntax für Processing Instruction. Jedes XML-Dokument soll durch einen XML-Parser verarbeitet werden können. D.h. die XML-Deklaration ist eine Processing Instruction an den XML-Parser.
encoding: Zeichensatzangabe. Hier Standardzeichensatz mit deutschen Umlauten

BSP1: Ein minimales XML-Dokument:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wurz1> Ein karges Dokument</Wurz1>
```

BSP2: Ein Dokument mit Teilbäumen:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Freundeskreis>
  <Freund bk="gut">
    <NName> Fritz </NName>
          <Adr>
      <Name>Müller</Name>
      <Vname>Friederich</Vname>
           <Plz>50678</Plz>
    </Adr>
  </Freund>
  <Freund bk="ferner>
    <NName>Otto</NName>
     <Tel>474747</Tel>
      <Adr>
       <Name>Schmitz</Name>
       <Email>otto_schmitz@gmx.de</Email>
      </Adr>
  </Freund>
  <Freund bk="Bier">
    <Nname>Kalle</NName>
    <Kneipe> Früh </Kneipe>
  </Freund>
</Freundeskreis>
```

BSP3: Ein XML-Dokument, dass möglichst strukturtreu alle Daten und Metadaten einer Tabelle T aus einem RDBS abbildet. Hier T=Artikel (Artikeltabelle) mit dem Tabellenschama RS(T): RS(T) = { (artnr,int,PRIK) , (artbez,cahr(30),0) , (preis,decimal(7,2), [preis >=0]) }

XML-Dokument:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tabelleMa SYSTEM "tabelleMa.dtd">
<tabelleMa>
<tabname>Artikel</tabname>
<Zeile>
        <artnr DT="int">4711</artnr>
        <artbez DT="char(30)">Parfüm</artbez>
        <preis DT="decimal(7,2)">5.95</preis>
</Zeile>
<Zeile>
        <artnr DT="int">4810</artnr>
        <artbez DT="char(30)">Seife</artbez>
        <preis DT="decimal(7,2)">0.45</preis>
</Zeile>
...
</tabelleMa>
```

: Dokumenttyp-Deklaration
: Name des Dokument-Element (Wurzelelement)
: Datei (+Pfad-) Name der DTD-Datei (DTD := Dokumenttyp-Deklaration)

7.1.2 DTD, Validität

Gegeben ist ein XML-Dokument und eine zugehörige DTD. Ein XML-Parser kann prüfen, ob ein XML-Dokument sich an die in der DTD vereinbarte Struktur hält.
Ein XML-Dokument ist valide <=> ihm ist eine DTD zugeordnet und es entspricht der in der DTD festgelegten Struktur.
A) Die DTD wird dem XML-Dokument im Prolog zugeordnet (s. 7.1.1)
B) Eine DTD besteht aus Deklarationen
B1) Elemente, B2) Attribute, B3) Entities (=ein Zeichen oder ein XML-Teildokument), B4) Processing Instructions, B5) Notationen

Anm.: Zu B3)Entity-Deklarationen notwendig, falls Sonderzeichen '<', '>', ' " ',... benötigt werden, falls ein XML-Dokument auf mehrere Teildokumente verteilt werden soll.
Zu B4)z.B. erforderlich, wenn man einem XML-Dokument ein XSLT-Stylesheet für die grafische Gestaltung zuordnen will.
Zu B5)Notationen dienen dafür, um Dateien mit Nicht-XML-Dateien in ein XML Dokument zu integrieren.

B1) DTD-Deklaration für Elemente: Bei dieser Deklaration wird unterschieden,a) ob ein Element deklariert werden soll, das nur Nutzdaten enthält oder b) ob ein Element deklariert werdensoll, das einen XML-Teilbaum enthält.
a) <!ELEMENT tagname (#PCDATA)> PCDATA: Parsed Character Data
Bsp.: <!ELEMENT preis(#PCDATA)>
b) Wenn ein Element andere Elemente enthält, werden diese mit Quantitäten in einer Klammer notiert (≈Tupelnotation):
<!ELEMENT tagname (ce1 q1, ce2 q2, ... ceN qN) qa>

mit cei : i-tes Kindelement
qi : Quantität des i-ten Kindelements
qa : Quantität des Tupels

Quantitäten:

q = nichts	<=>	genau 1
q= ?	<=>	höchstens 1 (0 oder 1)
q=+	<=>	mindestens 1 (1...n)
q=*	<=>	0 oder mindestens 1 (0...n)

BSP1: <!ELEMENT zeile(artnrm artbez,preis)+>
BSP2: <!ELEMENT buch(autor+,titel,schlagwortfolge?)>
<!ELEMENT schlagwortfolge(nr, schlagwort+)+>

Ü1: Betrachten sie das XML-Dokument von Bsp.:3 ohne XML-Attribute. Stellen Sie dafür eine DTD auf!

zu B2) Attributdeklarationen

Ein Element kann mehrere Attribute haben. In der DTD werden für jedes Attribut folgende drei Angaben hinterlegt:

a) Attributname
b) sog. XML-Datentyp
c) Voreinstellung

zu b) CDATA := Characterdata
zu c) Muß-Attribut: #REQUIRED
Kann-Attribut: #IMPLIED
"Defaultwert": voreingestellter Wert für das Attribut (mit einem implizierten #IMPLIED)

Allgem. Syntax:

```
<!ATTLIST elementname
        att1    DT1    v1
        ...     ...    ...
        attn    DTn    vn
>
```

mit: att_i : Name des i-ten Attributs ($1 \leq i \leq N$)
DT_i : XML-Datentyp des i-ten Attributs
v_i : Vereinstellung des i-ten Attributs

Anm.: Mit der DTD zu Bsp.3 hat man im Prinzip schon eine Norm-DTD für XML-Dokumente gegeben, die durch Tabellenexport aus RDBn entstehen:

```
<!ELEMENT tabelleNdtd (allgAngTab, Zeile*)>
<!ELEMENT allgAngTab ( *1 )>
<!ELEMENT Zeile (sp1, sp2, ..., spM)>
<!ELEMENT sp1 (#PCDATA)>
  ......
<!ELEMENT spM (#PCDATA)>
<!ATTLIST sp1
        DT CDATA #REQUIRED
        IB1 CDATA "NULLS"        => IB = Integritätsbedingung
```

```
        IB2 CDATA #IMPLIED
        BEM CDATA #IMPLIED    => BEM = Bemerkung
>
 .....
<!ATTLIST spM
        DT CDATA #REQUIRED
        IB1 CDATA "NULLS"
        IB2 CDATA #IMPLIED
        BEM CDATA #IMPLIED
>
```

Anm.: zur Norm-DTD:

a) IB1: Integritätsbedingung: NULLS / NOT NULLS
   IB2: " PRIK / FKEY
   BEM: Anmerkungen über weitere Integritätsbedingungen (z.B.: Werteintegrität, o. ä.)
b) zu (*1): Hier ist eine Projektentscheidung zutreffen, ob das XML-Element allgAngTab durch einen Teilbaum (i) oder durch ein einfaches XML-Element (ii) aufgelöst wird.

Bsp.(i):

```
<!ELEMENT allgAngTab (tabname, verfasser, projekt)>
<!ELEMENT tabname (#PCDATA)>
<!ELEMENT verfasser (#PCDATA)>
<!ELEMENT projekt (#PCDATA)>
```

Bsp.(ii):

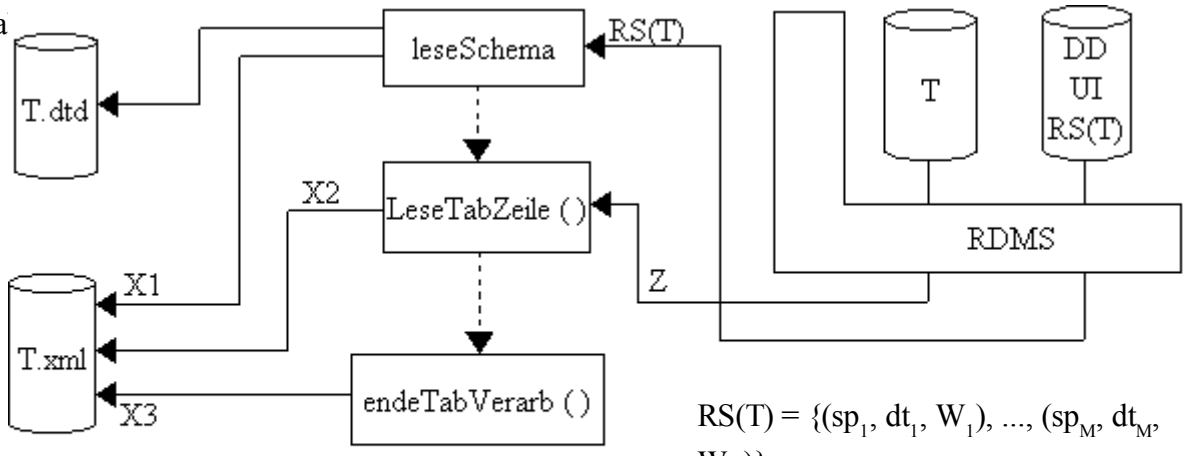
```
<!ELEMENT allgAngTab (#PCDATA)>
<!ATTLIST allgAngTab
        verfasser CDATA #REQUIRED
        projekt CDATA #REQUIRED
>
```

Hierbei wird dann der Tabellenname in die #PCDATA-Sequenz des Elements allgAngTab geschrieben.

7.2. Export von RDB-Tabellen in XML-Dateien

T.dtd stellt einen Ausschnitt aus der Norm-DTD für Ta

Modularer Aufbau eines Export-Programms



RS(T) = {(sp_1, dt_1, W_1), ..., (sp_M, dt_M, W_M)}
RS(T) wird erzeugt durch die JDBC-Metadatenabfrage für die Tabelle T, durch die man die Felder [sp_1, ..., sp_M] und [dt_1, ..., dt_M] erhält.

Anm.1: leseTabZeile ( ): Liest eine Tabellenzeile Z aus dem ResultSet der SQL-Abfrage "SELECT * FROM T"

Anm.2: X1 besteht aus den XML Zeilen:
```
<? xml version="1.0" ...>
<!DOCTYPE tabelle SYSTEM "T.DTD">
< tabelle>
<tabname> T </tabname>
```

Anm.3: Mit X2 werden für jede Zeile Z folgende XML-Zeilen in T.XML eingefügt:
```
<zeile>
<sp1> w1 </sp1>
<sp2> w2 </sp2>
.....
<spM> wM </spM>
</zeile>
```

Anm.4: Mit X3 schreibt man folgendes Abschlusstag nach T.XML:
```
</tabelle>
```

7.3. XML-Parser

XML-Parser sind Programme, um die Wohlgeformtheit und die Validität von XML-Dokumenten zu prüfen. Sie bieten die Möglichkeit ereignisorientiert oder XML-strukturorientiert XML-Dokumente zu verarbeiten. Es gibt zwei Parser-Modelle für XML:

a) Das ereignisorientierte Parser-Modell: SAX: Simple API for XML-Parsing:
Standardereignisse sind: (e1) Dokumentanfang

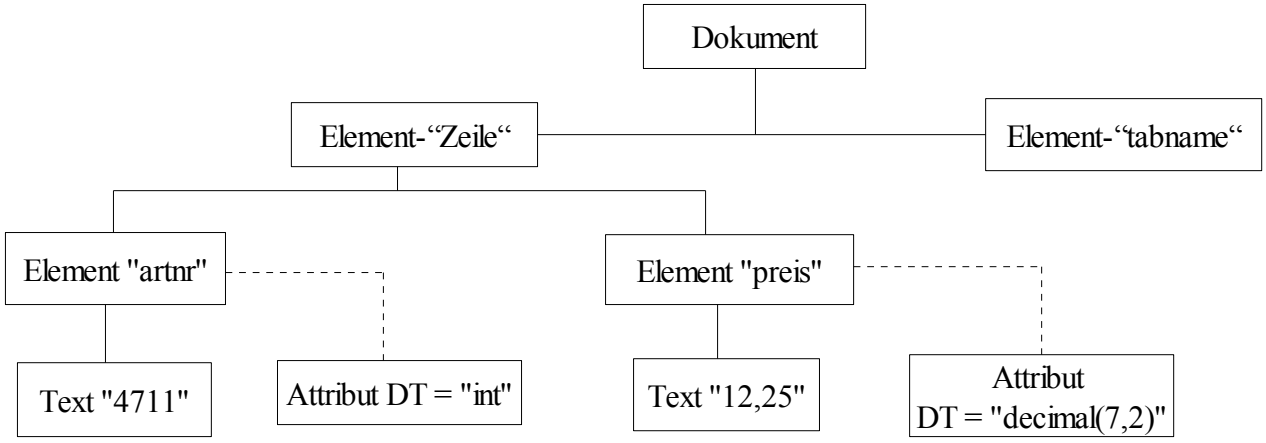
(e2) Dokumentende
(e3) Elementanfang
(e4) Elementende
(e5) Nutzdatenende

Ein SAX-Parser arbeitet das XML-Dokument sequentiell ab, d.h. er kann nicht auf verflossene Zustände zurückgesetzt werden. Auch die SAX-Validitätsprüfung, die gegen die DTD des Dokuments prüft, gibt ihre Ereignisse ereignisbezogen aus.

b) Das strukturoreintierte Parser-Modell: DOM: Document Object Model

Der DOM-Parser baut im RAM eine Baumstruktur auf, die der XML-Dokumentstruktur entspricht. (=> der DOM-Parser hat gegenüber dem SAX-Parser einen höheren RAM-Speicherbedarf).

Bsp. Baumstruktur:



DOM stellt für den Dokumentenbaum Navigationsmethoden zur Verfügung, z.B.:

```
getParentNode( )
getFirstChild( )
getSibling( )          /* Zugriff auf einen Geschwisterknoten */
....
```

Anm.: Java bietet sowohl DOM- als auch SAX-Unterstützung.

XML-Programmierung der SAX-Parser-Schnittstelle

A) Pakete:

A1) javax.xml.parsers: SAX-Parser, SAXParserFactory
A2) org.xml.sax: SAX-Parser-Interfaces
A3) org.xml.sax.helpers: Klassen für A2)

B) Allgemeine Funktionsweise von XML-Parsing-Programmen zur Prüfung der Wohlgeformtheit und der Validität

B1) Aufruf eines SAX-Parsers (Historie: J1.4: Crimson; J1.5: Xerxes)
a1)Anlegen einer SAX-Parser-Factory:

```
SAXParserFactory factory = SAXParserFactory.newInstance( );
```

a2)Setzen der Validierungsanforderung:
factory.setValidating(true);

a3)Anlegen der SAX-Parser-Instanz:
SAXParser saxparser1 = factory.newSAXParser( );

a4)Mit der SAX-Parser-Instanz eine XMLReader-Instanz anlegen:
XMLReader read1 = saxparser1.getXMLReader( );

a5)Zuordnung des Content-Handlers (Der Content-Handler enthält Methoden, die durch den XML-Parser angestoßen weden, wenn bestimmte Ereignisse auftreten, z.B.:
START_Element, END_Element, Alle_PCDATA-Einträge-gelesen,.... Sie programmieren eine ContentHandler-Klasse, die die ContentHandler-Schnittstelle aus org.xml.sax implementiert. Dabei werden die Methoden, die in dieser Schnittstelle stehen, für die zu programmierende Anwendung überschrieben.):
read1.setContentHandler(handler);
mit: MyContentHandler handler = new MyContentHandler( );

a6)Zuodrnung Error-Handler: (Der Error-Handler enthält Methoden, die Warnungen und Fehlermeldungen produzieren, wenn Syntax oder Validitätsverletzungen auftreten):
read1.setErrorHandler(ehandler); mit MyErrorHandler ehandler = new MyErrorHandler( );

a7)Aufruf des SAX-Parsers durch Verwendung der Methode parse( ) der Schnittstelle XMLReader:
SIG: void parse(String systemId)
systemId: Angabe des Namens der zu parsenden XML-Datei in URI-Notation
(URI := Uniform Resource Identifier)
(URI = URL- oder URN-Dateiname, URN = Uniform-Resource-Name (URN = „file“+pfad+dateiname))

Bsp.: Datei: tabelleMa.xml Pfad: c:\gbjavat\
=> systemId=“file:/c:/gbjavat/tabelleMa.xml“.
Tip: String h1 = new File(filename).toURL( ).toString( );

Aufruf des SAX-Parsers: read1.parse(h1);

Anm.: Der SAX-Parser-Aufruf sollte in einen try-catch-Anweisung eingekleidet sei, in der auf folgende Ausnahmefälle reagiert wird: IOException, ParserConfigurationException, SAXException.

B.2) Programmierung der Content-Handler-Klasse:
b.1) Implementation der ContentHandler-Schnittstelle:

```
public static class MyContentHandler implements ContentHandler {
        //attribute
//Methoden}
```

b.2) Die MyContentHandler-Klasse enthält 11 zu überschreibende Methoden:

(1) public void startDocument( )
    Hier werden die Aktivitäten programmiert, die am Anfang eines XML-Dokuments auszuführen sind.
(2) public void endDocument( )

Aktionen, die nach dem Ende des XML-Dokuments auszuführen sind.
(=EOF-Verarbeitung der XML-Datei)

(3) public void startElement
(String uri, String localName, String qName, Attributes attributes) throws SAXException
^a ^b ^c ^d
(a) uri = uri Name des Tags, kann null sein.
(b) localName = TagName(wird zurückgegben, wenn (a) ≠ null ist), kann null sein.
(c) Tagname in der Form: [uri-Präfix:]Tagname (wird immer zurückgegeben!).
(d) attributes = Attributliste des Tags

Verarbeitung der Übergabeparameter:
v1) String tagname; tagname = qName;
v2) Verarbeitung der Attributliste:
i) AttributesImpl a1 = new AttributesImpl(attributes);
AttributesImpl ist eine Klasse aus org.xml.sax.helpers mit der die Schnittstelle Attributes implementiert wird.
ii) Feststellen der Attributlistenlänge: int L1 = a1.getLength( );
iii) Schleife zum Zugriff auf alle Werte und Typen der in dem Tag vorkommenden Attribute:

```
for (i=0; i < L1 ; i++) {
        String gV1 = a1.getValue(i);
        String gVT = a1.getType(i);
        String gNam = a1.getQName(i);
        //Verarbeitung
}
```

(4) public void endElement
(String uri, String localName, String qName) throws SAXException
Aktionen, die am Ende eines Elements auszuführen sind, werden hier programmiert.

Bsp.: if (qName.compareTo("Zeile") = = 0) //ausführen eines insert-befehls
(in der Import-Verarbeitung: XML-Datei => RDB-Tabelle)

(5) public void characters
(char[] ch, int start, int length) throws SAXException
Diese Methode gibt die Zeichenfolge der Nutzdaten eines Elements zurück, die zwischen dem öffnenden und schließenden Tag stehen.
ch: Zeichenfolge, start: OFFSET von ch im XML-Dokument,
length: Länge der Nutzdatenfolge
Verarbeitung: String nh = new String (ch, start, length);

Weitere ContentHandler-Methoden:

(6) public void ignorableWhitespace
(char[] ch, int start, int length) throws SAXException
(7) public void startPrefixMapping(String prefix, String nri) throws SAXException
(8) public void endPrefixMapping(String prefix) throws SAXException
(9) public void skippedEntity(String name) throws SAXException
(10)public void processingInstruction(String target, String data) throws SAXException
(11)public void setDocumentLocator(Locator locator)

Bsp.: System.out.println("Locator: "+locator.toString( ));
=> Angabe über den benutzten XML-Parser wird ausgegeben.

B.3) Programmierung der Error-Handler-Klasse

C.1)Klasse soll ErrorHandler-Schnittstelle implementieren:

```
public static class MyErrorHandler implements ErrorHandler
    {...}
```

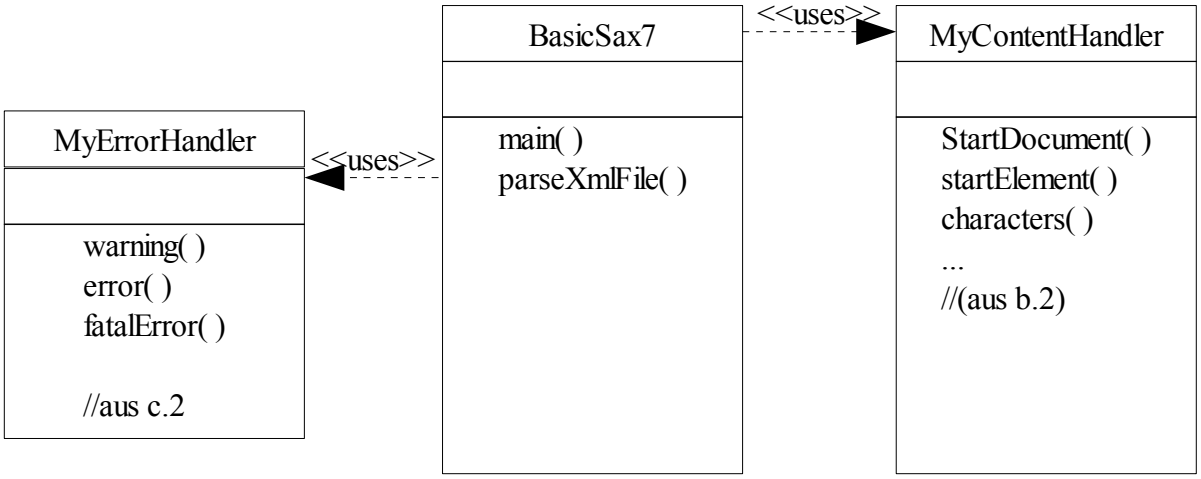
C.2) Methoden:

(1)
```
public void warning(SAXParseException ep) throws SAXException
    {System.out.println("Warnung: "+ep.toString( ));
     System.out.println("betr. Entity: "+ep.getPublicId( ));
     System.out.println("Zeile: "+ep.getLineNumber( ));
     System.out.println("Spaltenpos.: "+ep.getColumnNumber( ));
    }
```

(2)
```
public void error(SAXParseException ep) throws SAXException
   { //Aktionen wie in (1)
    [//Programmabbruch dann, wenn es die Anwendung erfordert (z.B. bei Insert)
    System.exit(2);]
    }
```

(3)
```
public void fatalError(SAXParseException ep) throws SAXException
    { //Aktionen wie in (1)
      System.exit(3); //Abbruch
     }
```

C) Beispiel Programm-System:



main( ):

- XML-Dateiname einlesen
- Instanz von MyContentHandler anlegen
- Instanz von MyErrorHandler anlegen
- Aufruf on parseXMLFile( ) [validierend (j/n)]

SIG: public static void parseXMLFile
(String filename, MyContentHandler c1, MyErrorHandler e1, boolean val)
val = = true <=> der Parser prüft doe Validität
parseXmlFile( ) führt die Aktionen a.1) bis a.7) aus.

7.4. Aufbau von RDB-Tabellen aus XML-Dateien

Geg.: Eine XML-Datei mit Nutzdaten, die in eine RDB-Tabelle T einzufügen sind. Die XML-Datei wird durch eine Norm-DTD beschrieben, die den Zeilenaufbau von T als XML-Elementdefinition enthält. Die Norm-DTD erfüllt folgende Bedingungen:

(1) Sie enthält ein Element <zeile>, das aus Spaltenattributen sp_1, ..., sp_M der aufzubauenden Tabelle T besteht.
(2) Jedes Element <sp_i> ($1 \leq i \leq M$) hat ein Attribut mit Namen "DT" und einem Attributwert, der ein SQL-Datentyp ist.
(3) Sie enthält ein Element mit dem Tabellennamen (z.B.: <tabname>).

A) Attribute der MyContent-Handler-Klasse

Um die Kommunikation zwischen den Call-Back-Methoden der Content-HandlerKlasse zu gewahrleisten, werden folgende Attribute benötigt, um pro <zeile>-Subdokument der XML-Datei ein INSERT-Kommando zu erzeugen.

a.1) Attribute zum Aufbau eines INSERT:

Allgem. Syntax: INSERT INTO tabelle (sp_1, ..., sp_M) VALUES (w_1, ..., w_M)
^^ insertAuf. ^^tabelle ^^spaltseq ^^values ^^wertseq

```
String insertAuf = new String ("INSERT INTO");
String tabelle = new String ( );
String spaltseq = new String ( );
String values = new String ("VALUES" );
String wertseq = new String ( );
```

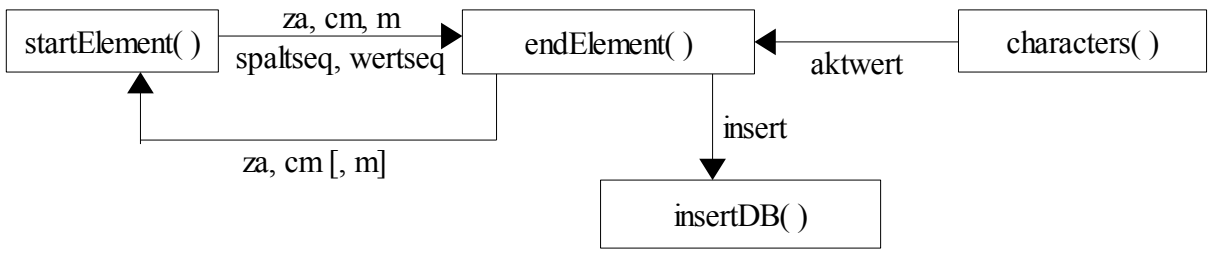
a.2) Zustandsattribute für die Kommunikation zwischen den Call-Back-Methoden startElement( ) und endElement( ).

```
int za;        => Zeile aktiv <=> za = 1
int cm;        => dtyp(spi) = "char" <=> cm=1
[int m;        => lfd. Nr. der Zeile ]
```

a.3) Ein Attribut zur Verwaltung des Nutzdatenstrings, der von der Call-Back-Methode characters( ) erzeugt wird und in der Methode endElement( ) weiterverarbeitet werden kann:

```
String aktwert;
```

Diagramm zum Nachrichtenaustausch zwischen den Call-Back-Methoden:



B) Aktivitäten in startElement( ):
B.1) Zeilenanfangsverarbeitung:
Auslöser: qName = "zeile"
Aktionen: i) za = 1
ii)spaltseq, wertseq mit "(" initialisieren.

B.2) Verarbeitung von Elementen in einer Zeile:
Auslöser: za = 1
Aktionen: i) Falls dtyp(sp_i) = "char" => setze cm = 1
ii) spaltseq = spaltseq[+","] + qName

C) Aktivitäten in characters( )
Aktionen: String h = new String (ch, start, length);
aktwert = h;

D) Aktivitäten in endElement( )
D1) Tabellenname isolieren:
Auslöser: qName = "tabname"
Aktionen: tabelle = aktwert;

D2) Wertsequenz pflegen:
Auslöser: za = 1
Aktionen: 1) wertseq = wertseq [+","][+" ' "]+aktwert[+ " ' "]
2) cm = 1 => cm = 0

D.3) Zeilenendeverarbeitung
Auslöser: qName = "zeile"
Aktionen: 1) spaltseq, wertseq mit ")" abschließen
2) za = 0
3) INSERT-String aufbauen und insertDB( ) aufrufen.

8. Algorithmen der Sekundärspeicherverwaltung (Bayer-Bäume)

8.1. Schlüssel und Offsets

Ziel von Algorithmen der Sekundärspeicherverwaltung ist, mit wenigen Sekundärspeicherzugriffen (auf Festplatten o. ä.) zu entscheiden, ob für einen Benutzerwert BKEY ein Datensatz DS mit PRIK-Wert PRIK(DS) = BKEY existiert oder nicht.
Datensätze DS einer DB seien auf einem Sekundärspeicher in folgender Form gespeichert:

o_1| p_1| |ds_1	o_i = Offset, wo der Datensatz ds_i auf dem Sekundärspeicher beginnt (Position des Startbytes von ds_i) $1 \le i \le N$
o_2| p_2| |ds_2	p_i = Primary-Key-Wert von ds_i
...	
...	
...	
o_n| p_n| |ds_n	

Anm.1: Der jeweilige Offset o_i wird beim Einfügen (INSERT) des Datensatzes ds_i vom DBMS festgestellt.
Anm.2: Ist der Offset o_i zu einem PRIK-Wert p_i bekannt, d.h. das Paar (p_i, o_i) ist für alle $1 \le i \le N$ gegeben, dann kann mit einem Sekundärspeicherzugriff ds_i gelesen werden.

Folgerung 1: Während der Laufzeit des DBMS werden alle Paare (p_i, o_i) ($1 \le i \le N$) in einer langen

RAM-Tabelle gesammelt und bei Ende der DBMS-Laufzeit auf den Sekundärspeicher geschrieben. (= Index-Tabelle IDX, die auf dem Sekundärspeicher geschrieben wird.)

Aufwand bei diesem System der Indexverwaltung:

a) N Lernoperationen bei Start der DBMS-Sitzung.
b) Pro Leseanfrage ≈ N/2 Durchsuchungsoperationen in der IDX-RAM-Tabelle
c) N Schreiboperationen bei Ende der DBMS-Sitzung.

Ziel der Algorithmen der Sekundärspeicherverwaltung:

Die Aufwände a), b), c) sollen minimiert werden.

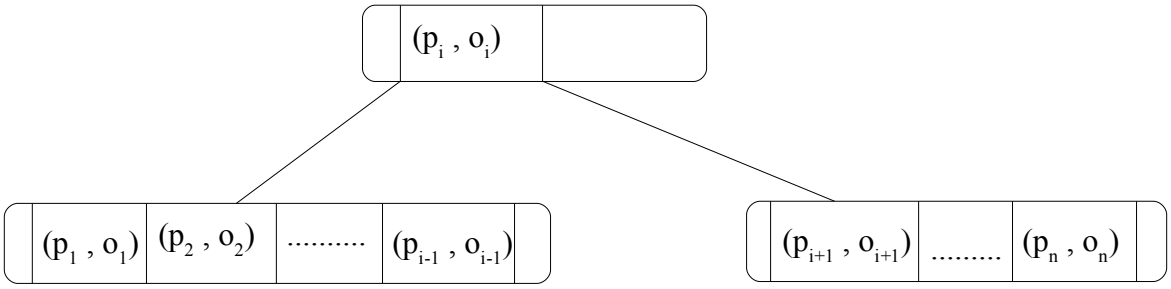
Bsp.: Eine Strategie der Minimierung:

Binäre Suche in der IDX-RAM-Tabelle: Resultat: Aufwand(b)) ≈ ld(N)
Preis: Für die Vor. der binären Suche, daß die IDX-RAM-Tabelle sortiert sein muss, hat man folgenden Zusatzaufwand:

d) Pro INSERT von einem Datensatz ds_{N+1} sind folgende Operationen notwendig:
   1. ld(N) Suchopertionen im RAM
   2. ≈ N/2 Schreiboperationen im RAM

Bsp.: Minimierung von a), c) durch Lesen bzw. Schreiben der Folge der Indexpaare (p_i, o_i) in Rekords. (Indexdatensätze die aus M Paaren bestehen => Aufwand (a)) = Aufwand (c)) = N/M)

Anm.: Man kann beide Strategien in einer Index-Verwaltung mit einem ausgeglichenem Binärbaum kombinieren:



R. Bayer, E.M. McCreight: „Organisation and Maintenance of Large Ordered Indexes“, (1972), Acta Informatica 1, S. 173 – 189

=> Ausweitung der obrigen Strategie auf ausgeglichene Bäume mit X Nachfolgerknoten
$M+1 \leq X \leq 2M+1$ $(M \geq 2)$

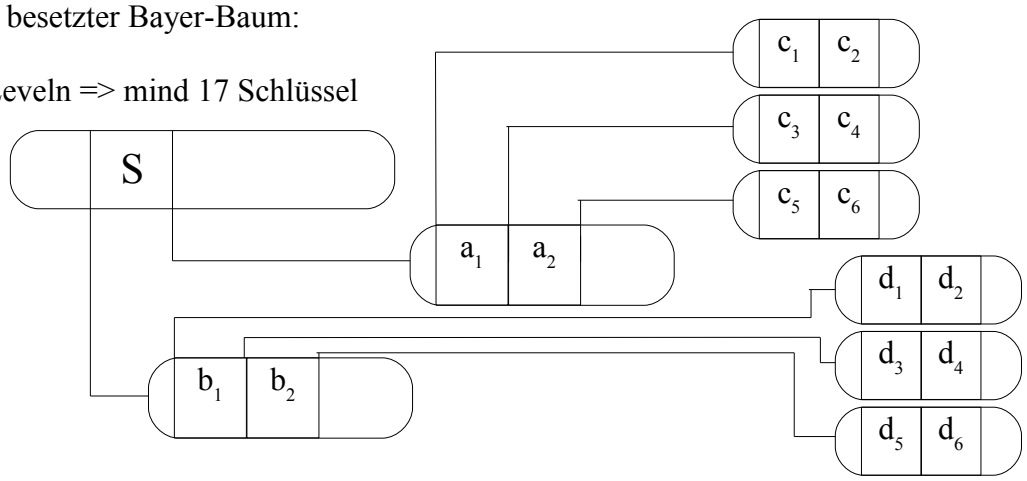
8.2. Aufbau von Bayer-Bäumen

Def.1: (Bayer-Baum) Ein Bayer-Baum der Ordnung M ($M \in \mathbb{N}, M \geq 2$) ist ein Baum, der folgende Eigenschaften hat:

(1) Jeder Knoten enthält als Färbung höchstens 2M Schlüssel.
(2) Jeder Knoten außer der Wurzel enthält mindestens M Schlüssel. Die Wurzel enthält mindestens 1 Schlüssel.
(3) Jeder Knoten außer den Blatt-Knoten hat (K+1) Nachfolger, wenn er K Schlüssel hat.
(4) Alle Blätter liegen auf der gleichen Baum-Ebene.

Anm.: 1) Ein Knoten, der nicht Wurzel und Blatt ist, hat k Nachfolger mit $M+1 \le k \le 2M+1$.
2) Man kann die Hierarchieebenen eines Bayer-Baumes numerieren: Von der Wurzel (Level 0) bis zu den Blättern (Level x). Alle Blätter haben den gleichen Level x.

Bsp.: Minimal besetzter Bayer-Baum:
M=2
mit 3 Leveln => mind 17 Schlüssel



Bem.: Bei einem ausgeglichenen Binärbaum mit 3 Leveln hat man nur 7 Schlüssel untergebracht.

Algorithmus des Einfügens von Schlüsseln in einen Bayer-Baum:
Gegeben:
1) Bayer-Baum der Ordnung M.
2) Ein einzufügender Schlüssel m_x.
3) Ein Knoten K_0 auf Hierarchiestufe i (mit $0 \le i \le h-1$, wenn Baum h Level hat), in den der Schlüssel m_x nach Sortierordnung des Baumes einzufügen ist.

Verfahren: Fallunterscheidung: (einfügenBB (K_0,m_x))
Fall I) K_0 hat a<2M Schlüssel => m_x wird gemäß Sortierordnung in K_0 eingefügt.
Fall II) K_0 hat a=2M Schlüssel => Folgende Schritte sind auszuführen:
II.1 Bilde aus dem Schlüssel von K_0 und dem Schlüssel m_x eine sortierte Schlüsselmenge:
$S(K_0,m_x) = \{ s'_1, s'_2, ..., s'_M, ..., S'_{2M+1} \}$, wobei $m_x = S'_i$ für ein i mit $1 \le i \le 2M+1$.
II.2 Bestimme in $S(K_0,m_x)$ den mittleren Schlüssel s'_M .
II.3 Bilde den Knoten K_{01} mit den Schlüsseln $\{ S'_{1,...}, S'_{M-1} \}$ und
K_{02} mit den Schlüsseln $\{ S'_{M+1}, ..., S'_{2M+1} \}$.

Die Zerlegung des Knotens K_0 in die neuen Knoten K_{01} und K_{02} heisst SPLIT( )-Operation

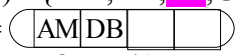
II.4 Fallunterscheidung in Bezug auf die Hierarchiestufe i:
a: i>0 => Füge S'_M in den Knoten K_p (Parentknoten von K_0 auf Hierarchiestufe (i-1) ein). (<=> Rekursiver Aufruf: einfügenBB(K_p, S'_M)).
b: i=0 => Bilde neuen Wurzelknoten mit S'_M als einzigen Schlüssel.

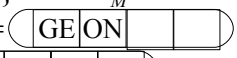
Anm.: Der Bayer-Baum wächst von den Blättern zur Wurzel.
Bsp.: Aufbau eines Bayer-Baums für folgende Schlüsselmenge:

SM =
{AM,EL,GE,DB,ON,TI,EKS,ST,PH,PI,FB,MA,ASS,DSS,TK,GT,SP,DN,SI,SE,BM,AZ,MM,IB}

Bayer-Baum mit M=2

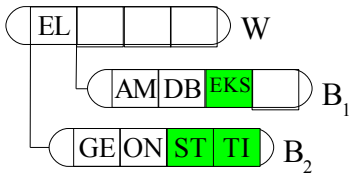
1) Fülle die Wurzel:
   W = 

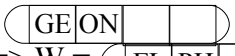
2) Einfügen: ON => SPLIT (): S(W,{ON})= {AM,DB,EL,GE,ON} => $S'_M = EL$
   => Bilde die Knoten B_1= 

, B_2= 

   => Bilde neue Wurzel (gemäß II.4.b): W = 

   Zwischenergebnis:
   Bayer-Baum BB1

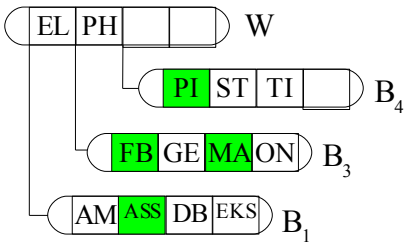


3) In BB1 wird gemäß Sortierordnung eingefügt: TI, EKS, ST (hier gekennzeichnet)
4) Einfügen PH => SPLIT(B2) : S(B2,{PH}) = {GE,ON,PH,ST,TI} => $S'_M = PH$
   => Bilde neue Knoten B_3 = 

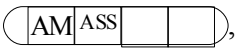
, B_4 = 

   => EinfügenBB(W,{PH}) => W = 

   Zwischenergebnis:
   Bayer-Baum BB2



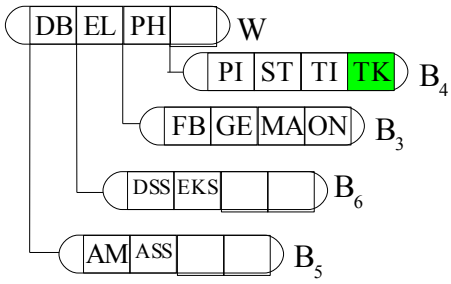
5) Einfügen in BB2: PI,FB,MA,ASS.
6) Einfügen: DS => SPLIT (B1) : S(B1,{DSS}) = {AM,ASS,DB,DSS,EKS}

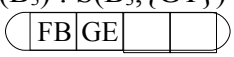
   => B_5= 

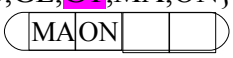
, B_6 = 

, W= 

   Zwischenergebnis:
   Bayer-Baum BB3

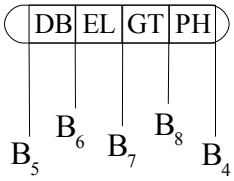


7) Einfügen in BB3: TK
8) Einfügen: GT => Split(B_3) : S(B_3,{GT}) = {FB,GE,GT,MA,ON}
   => B_7 = 

, B_8 = 

   => W = 

   Zwischenergebnis:
   Bayer-Baum BB4

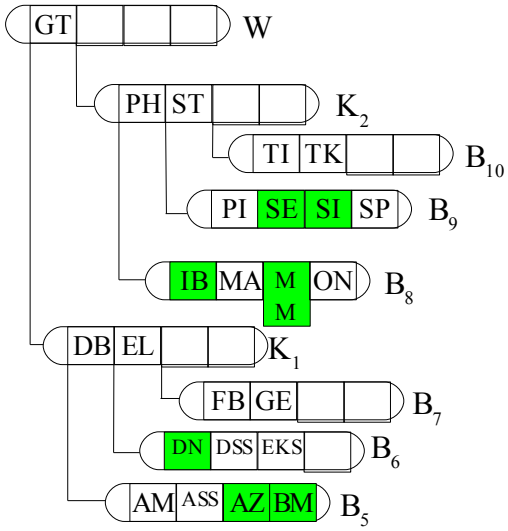


9)

10) Einfügen: SP => Split(B4): S(B4,{SP}) = {PI,SP,ST,TI,TK} => $S'_M = ST$
=> B_9 = PI | SP, B_{10} = TI | TK

=> einfügenBB(W,{ST})
=> SPLIT(W) : S(W,{ST}) = {DB,EL,GT,PH,ST} => $S'_M = GT$
=> K_1 = DB | EL, K_2 = PH | ST, W = GT

Zwischenergebnis:
Bayer-Baum BB5



Einfügen in BB5: DN,SI,SE,BM,AZ,MM,IB

8.3. Löschen von Schlüsseln in Bayer-Bäumen: (LÖSCHE(k,m_x))

Gegeben: Bayer-Baum der Ordnung M. Ein Schlüssel m_x soll gelöscht werden.
Fallunterscheidung:

I) m_x liegt in einem Blatt B:
I.1) B hat a>M Schlüssel => lösche m_x in B.
I.2) B hat a=M Schlüssel => BALANCE(B, B') [B' ist hier der lexikographisch benachbarte Knoten]:

Bilde die Schlüsselmenge S(B,B') = { $\underbrace{S'_{1,\dots}, S'_{M-1}}_{\text{Schlüssel aus B ohne } m_x}$, $\underbrace{S'_M, \dots, S'_R}_{\text{Schlüssel aus B}}$ }

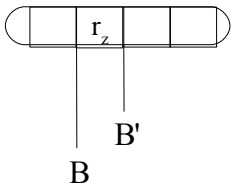
Schlüsselanzahl in B' k mit:

$$\begin{array}{r} M \le k \le 2M \\ +\,M-1 \le \quad \le M-1 \\ \hline 2M-1 \le k \le 3M-1 \end{array}$$

Schlüsselanzahl in S(B, B'):

Fallunterscheidung:
I.2.a) k=2M-1 => Knoten B wird gelöscht => Rekursion: LÖSCHE (K_p, r_z)
=> alle Schlüssel aus S(B,B') werden in B' eingefügt.

I.2.b) $k \geq 2M$ => Betrachte den Parentknoten K_p von B und B':



=> Bilde Schlüsselmenge S(B,B',{r_z}), bestimme deren mittleren Schlüssel $S_n^{||}$. Füge $S_n^{||}$ in K_p ein.
Füge alle Schlüssel S mit s< $S_n^{||}$ in B ein.
Füge alle Schlüssel s mit s> $S_n^{||}$ in B' ein.

II) M_x liegt in einem Nicht-Blatt-Knoten K: Dann sind folgende Schritte auszuführen:
a) Suche den zu m_x lexikographisch nächsten Schlüssel s_0 in einem Blatt B_1.
b) Überschreibe m_x mit dem Wert s_0 in K.
c) LÖSCHE(B_1, s_0)

8.4. Abschätzung des Suchaufwandes in einem Bayer-Baum

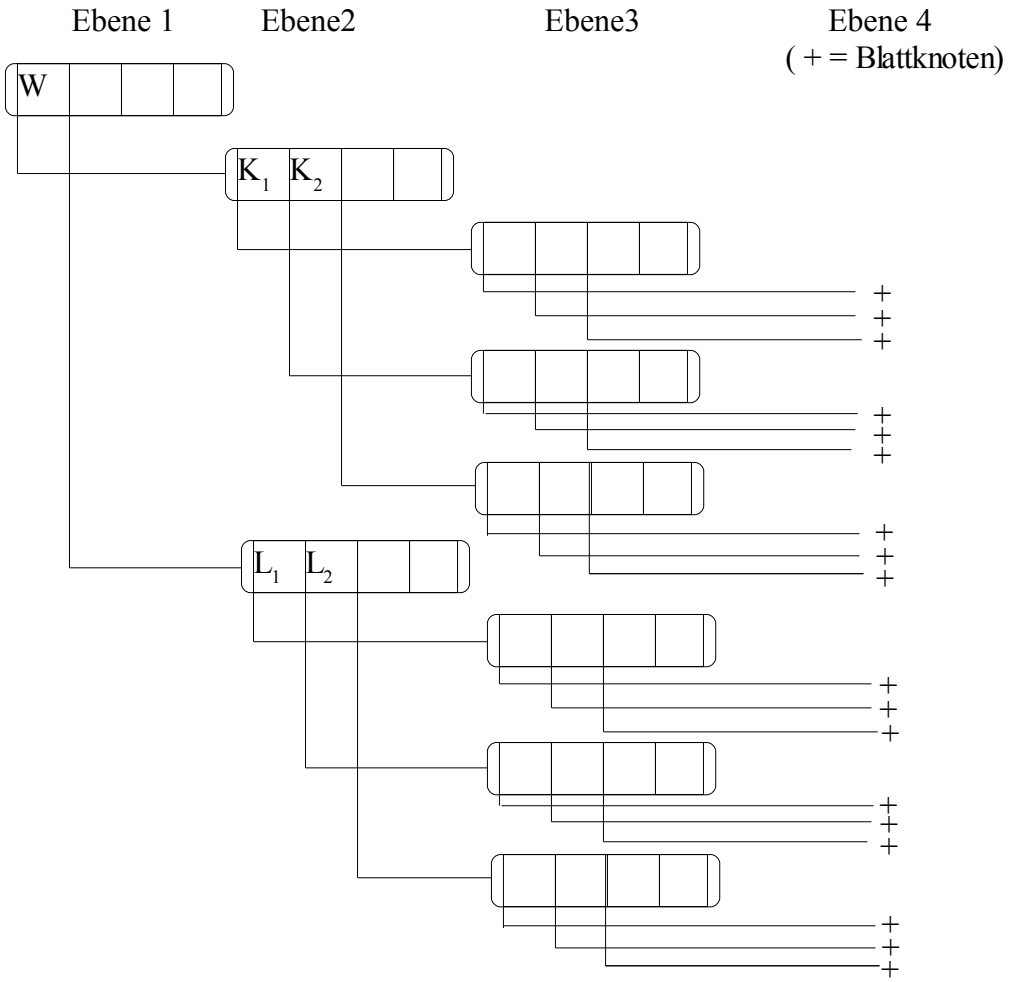
Bsp.1) 3 * N + 17 => O(N)
Bsp.2) 5 * N² + 3 * N + 9 => O(N²)

Gegeben: Ein Algorithmus A zur Berechnung einer Lösung X. Hierfür sind f(N) Operationen erforderlich. $N \in \mathbb{N}$. f(N) ist eine analytische Funktion. Z.B.: f = Polynom, Logarithmus, Exponentialfunktion.

Die Aufwandsnotation O(N) betrachtet in f(N) nur den am schnellsten wachsenden Term.
Bsp.3) f(N) = 5 * N + lg(N) => O(N)
Bsp.4) f(N) = e^{3N} + 7 * N^5 => O(e^N)

Anm.: a) Algorithmen mit O(N), O(N²), O(N^r). $r \in \mathbb{N}$ sind Probleme mit polynominalen Rechenaufwand
b) Algorithmen mit O(e^N) haben nicht polynominalen Aufwand.

Gegeben: Ein Bayer-Baum der Ordnung M, der minimal mit Schlüsseln gefüllt ist.
Gesucht: Der Aufwand, um zu entscheiden, ob ein Benutzerschlüssel S_0 im Bayer-Baum enthalten ist oder nicht.
Ansatz: Aufwand ≈ Anzahl h der Knoten, die von der Wurzel bis zu einem Blatt zu durchlaufen sind. Betrachte den allg. Aufbau des Bayer-Baums, um die Zahl h abzuschätzen.



Ebene	Anzahl Knoten	Anzahl Schlüssel
0	1	1
1	2	2 * M
2	2 * (M +1)	2 * M * (M+1)
3	2 * (M + 1)²	2 * M * (M + 1)²
...	...	...
L	2 * (M + 1)$^{L-1}$	2 * M * (M + 1)$^{L-1}$

Summe SK aller Schlüssel im Bayer-Baum:

SK = 1 + 2 * M + 2 * M * (M + 1) + ... + 2 * M * (M + 1)$^{L-1}$

= 1 + 2 * M * (1 + (M + 1) + ... + (M + 1)$^{L-1}$)

$$= 1+2\cdot M\cdot\sum_{k=0}^{L-1}(M+1)^k = 1+2\cdot M\cdot\left(\frac{(M+1)^L-1}{M+1-1}\right)$$

^^®

$$= 1+2\cdot M\frac{(M+1)^L-1}{M} = 1+2\cdot(M+1)^L-2 = -1+2\cdot(M+1)^L$$

zur Gleichung (R): $\sum_{k=0}^{n} q^N = \frac{q^{n+1}-1}{q-1}$

Berechne L in Abhängigkeit von SK:

$SK = -1 + 2 * (M+1)^L$

$\frac{SK+1}{2} = (M+1)^L$ | $\log_{(M+1)}$

$\log_{(M+1)}\left(\frac{SK+1}{2}\right) = L$

=> Aufwand h = L-1

h = $\log_{(M+1)}\left(\frac{SK+1}{2}\right) - 1$

=> Hat der Bayer-Baum N Schlüssel

=> Aufwand h = $\log_{(M+1)}\left(\frac{N+1}{2}\right) - 1$

=> $O\left(\log_{(M+1)}\left(\frac{N+1}{2}\right) - 1\right)$

Vergleich: a) Suche in einer linearen Liste (mit N Knoten): O(N)
b) Binäre Suche in einer linearen, sortierten Liste (N Knoten): O(ld(N))

Tabelle zum Vergleich des Suchaufwandes in a), b) und in c) Bayer-Bäumen:

N	a) = N/2	b) = ld(N)	c) Bayer-Baum M	$\log_{(M+1)}\left(\frac{N+1}{2}\right)$
100	50	7	2	4
100.000	50.000	17	10	5
1.000.000.000	500.000.000	30	300	4

29.6.2006

9. (Ausblick) Datenbanken und Wissensrepräsentation

Schlagwort: normdatei

Anlass:Semantic Web (W3C) : Grundidee: Im WWW soll nicht nur mit Operationen des Zeichenkettenvergleichs (wie bei Google o.ä. Suchmaschinen) gesucht werden, sondern mit Operationen, die auf den Bedeutungen von Zeichenketten operieren.
(Ähnlichkeitssuche)

Ein Beispiel für eine Ähnlichkeitssuche, die eine Charakterisierung von Büchern, die einen ähnlichen Inhalt behandeln, der durch eine Schlagwortkette beschrieben wird, benutzen, ist die Büchersuche in Bibliotheken mittels Schlagwortrecherche.
Diese Art der Recherche ist lange schon vor der Entwicklung des WWW ausgeführt worden.
Wichtiges Beschreibungsmittel für den normierten Aufbau von Schlagwortketten ist die sogenannte Schlagwortnormdatei, die von Bibliothekswissenschaftlern gepflegt wird.

Allgemein: Ein Wort W (bzw. eine Zeichenkette W) hat für eine Gruppe von WWW-Benutzern eine oder mehrere Bedeutungen BW1, ..., BWm.

BSP1:
a) Zeichenkette „Strom“
- ➔ BW1 := „bewegte elektrische Ladung“ (Elektrotechnische Entität)
- ➔ BW2 := „großes fließendes Gewässer“ (Geographische Entität)

b) Ähnliche Begriffe: BW1 :Ampére
Ähnliche Begriffe: BW2 :Fluß

BSP2:
a) Zeichenkette „Datenbanksystem“
b) Ähnliche Begriffe: „DBMS“(Kürzel), „strukturierte Datenhaltung“, „Informationssystem“

Ein Ziel: Bedeutungsarten (Bedeutungsmengen) sollen so strukturiert werden, dass sie für Such-Operationen auf WEB-Seiten mit Wörtern ähnlicher Bedeutungen genutzt werden können.

Probleme:
1) Klassische Bedeutungsdefinitionen sind in natürlicher Sprache gegeben
   a) natürliche Sprache ist maschinell nicht unmittelbar verarbeitbar
   b) zirkulärer Prozess (I.d.R. sind in einer Bedeutungsdefinition Wörter enthalten, die wiederum einer Definition bedürfen usw.)
2) Das Zusprechen von Bedeutungen zu Zeichenketten ist stark abhängig von der Benutzergruppe. (z.B. das Verb „ablangen“ hat im 18.Jh. eine Bedeutung, die uns heute hier nicht bekannt ist)

Erste Ansätze zur Festlegung definierter Vokabularien (Wörterverzeichnissen)

W3C-Stack

Ontologien	Ontologien können in einer auf RDF aufbauenden formalen Sprache bestimmt werden: OWL (Web Ontology Language)
RDF	Resource Description File (basiert auf XML)
XML	

Eine Ontologie O ist eine Menge von Wörtern, die in ihren Bedeutungen und in ihrem Zusammenhang von einer Benutzergruppe festgelegt sind.

O = (V, R, A, H)
V = Wörterverzeichnis , R = Relationen zwischen Wörtern aus V
A = Axiome für Wörter aus V , H = Begriffshierarchie in V

Übung

I) Klausur vom 17.7.03 – Aufgabe 6

Gegeben: $W_1 = \{3,5,7\}$, $W_2 = \{-1.5,4.1\}$ $A_1 \in W_1, A_2 \in W_2$
d_1=int , d_2=float

a) A3 := TUPLE OF(A_1:d_1, A_2:d_2)
Gesucht ist der Wertebereich W_3 von A_3 und seine Mächtigkeit $|W_3|$.

$W_3 = W_1 \times W_2 = \{ (3,-1.5) , (3,4.1) , ...\}$ $|W_3| = 6 = 3 \cdot 2$

b) Gesucht ist W_4 und $|W_4|$.
A_4 := SET OF (A_1:d_1)
Ansatz: W_4 ist die Potenzmenge.
Menge aller Teilmengen von W_1 = Potenzmenge(W_1) = $W_4 = P(W_1)$ =
{ {3} , {5} , {7} , {3,5} , {3,7} , {5,7} , {3,5,7} , { } }
$|W_4| = 8 = 2^{|W_1|} = 2^3$

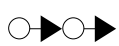
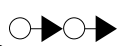
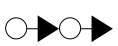
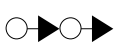
b') A_5= LIST OF(A_1:d_1) mit max. Listenlänge: 2

Gesucht ist W_5 und $|W_5|$.

$W_5 = \bigcup_{i=0}^{2} L^i(W_1)$ $L^i(W_1)$:= Menge der Listen mit i Knoten aus W_1

$W_5 = L^0(W_1) \cup L^1(W_1) \cup L^2(W_2) =$

$\{\} \cup$ {  ,  ,  } $\cup$ {  ,  ,  ,  ,  ,

 ,  ,  ,  }

$|W_5| = |L^0(W_1)| + |L^1(W_1)| + |L^2(W_1)| = 1 + |W_1| + |W_2|^2 = 1+3+9 = 13$

$|L^n(W_1)| = |W_1|^n$

c) Gegeben: S = [(3, -1.5, 3) , (5,-1.5, 7) , (3,-1.5,3) , (7,4.1,5)]
Gesucht ist ein ADT mit dem alle Elemente von S verwaltet werden können.

BAG OF (x: Z) mit Z := TUPLE OF(a:int, b:float, z:int)

II) Gegeben ist die Beschreibung der Informationselemente SENDUNG und T_BEITRAG in DataDictionary-Syntax:

SENDUNG ::= S-TITEL + S_DATUM + S_DAUER
T_BEITRAG ::= T_TITEL + REDAKTEUR + T_DAUER

<u>Aufgabe:</u>
ADT definieren für eine Liste der Textbeiträge einer Sendung (Sendungsattribute mit zugehörigem Textbeitrag)

<u>Lösung:</u>
DTSEN := TUPLE OF(S_nr: int, S_name: String, S_datum: int, S_dauer: int)
DTBEI := TUPLE OF(T_titel: String; T_redakt: String; T_dauer: int)
LTis := TUPLE (x: DTSEN, y: LIST OF(K:DTBEI))

Abkürzungsverzeichnis