Inhaltsübersicht: DB2

6.3 RDB-Generierung aus XML
6.3.1 XML Parser (Übersicht)
XML-Parser sind Programme, um die Wohlgeformtheit und die Validität von XML-Dokumenten zu prüfen. Es gibt zwei Modelle von XML-Parsen:

a) ereignisorientiertes Parsen SAX(**S**imple **A**PI for **X**ML-Parsing)
b) strukturorientiertes Parsen DOM (Document Objet Model) | JDOM)

zu a) SAX: Beim SAX-Parsen werden Standard-Ereignisse durch Call-Back-Methoden verarbeitet, die durch Eintreten der Standard-Ereignisse ausgelöst werden.

Standard-Ereignisse sind:

(e1) Dokumentanfang,
(e2) Elementanfang,
(e3) Nutzdatenfolge,
(e4) Elementende,
(e5) Dokumentende.

Ein Sax-Parser arbeitet sequentiell, dh er kann nicht auf verflossene Zustände zurückgreifen. D.h. die Verwaltung von Informationen, die über mehrere Zustände geleistet werden soll, muss vom Anwendungsprogramm sichergestellt werden.

zu b) DOM: Der DOM-Parser baut im RAM eine Baumstruktur auf , die dem XML-Dokument entspricht. Der DOM-Parser hat gegenüber dem SAX-Parser einen wesentlich höheren RAM-Speicherbedarf.

Beispiel-Baumstruktur:

DOM stellt für den Dokumentenbaum Navigationsmethoden zur Verfügung:

getParentNode():	Navigation zum Parentknoten des aktuellen Knotens.
getFirstChild():	Navigation zum ersten Kinderknoten des aktuellen Knotens (im Sinne der Inorder-Verarbeitung)
getSibling():	Navigation zum nächsten Geschwisterknoten des aktuellen Knotens (in Richtung der Inorder-Verarbeitung)

6.3.2 Programmierung der SAX-Parser Schnittstelle:
A) Pakete: A1: javax.xml.parsers : SAXParser, SAXParserFactory
A2: org.xml.sax: Schnittstellen(Interfaces): ContentHandler, ErrorHandler

zu A2)
Die ContentHandler-Schnittstelle enthält 11 zu implementierende Methoden, von denen 5 Methoden für die oben genannten Zustände wichtig sind:

(A.2.1) public void startDokument(): Hier werden die Aktivitäten programmiert die am Anfang des Dokuments auszuführen sind.

(A.2.2) public void startElement (String vri, String localname, String qName, Attribute attr) throws SAXException

Attribute attr = Attributliste des öffnenden Tags: Vererbung:

V1) Eine Implementation der Attributliste anlegen:

```
AttributesImpl ax = new AttrivutesImpl(attr);
//AttributesImpl Element org.xml.sax.helpers
```

V2) Anzahl der XML-Attribute ermitteln:

```
int k = ax.getlength();
```

V3) Schleife zum Abfrage der Attributnamen, der Attributwerte und der XML-Datentypen der Attribute:

```
for (int i = 0, i < k; i++){
    String an = ax.getQName(i); //Attributname ("qvalified name")
    String aV = ax.getValue(i); //Attributwert
    String aDt = ax.getType(i); //XML-Datentyp des Attributs
}
```

String qName = Name des öffnenden Tags ("qualified name": [uri-Präfix:] Tagname). in der Variable qName enthalten:

z.B. if(qName.equals("Zeile")){...}

uri-Präfix: Kann null sein wenn keine uri-Referenz im Tag steht.

Localname (Bestandteil der uri): kann null sein, wenn keine uri-Referenz im Tag steht
[Hinweis zur Definition einer uri: siehe wjc-Definition (www.w3.org)]

(A.2.4)
public void endElement (String uri, String localname, String qName) throws SAXException

Aktionen, die am Ende eines Elements auszuführen sind, werden hier programmiert.

```
if(qName.equals(Zeile)){
    //insert String abschließen; Aufruf einer JDBC-Insert-Methode
}
```

(A.2.5) public void characters (char [] ch, int start, int length)

Diese Methode gibt die Reihenfolge der Nutzdaten eines Elements zurück, die zwischen dem öffnenden und schließenden Tags des XML-Parsers stehen:

String nutzdaten = new String (ch, start, length);

(A.2.6) public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException

(A.2.7) public void startPrefixMapping(String prefix, String nri) throws SAXException

(A.2.8) public void endPrefixMapping(String prefix) throws SAXException

(A.2.9) public void skippedEntity(String name) throws SAXException

(A.2.10) public void processingInstruction(String target, String data) throws SAXException

(A.2.11) public void setDocumentLocator(Locator locator) throws SAXException

B) Aufruf eines SAX-Parsers

B1) Anlegen einer SAXParserFactory Instanz. Damit wird gesteuert, ob der Parser nur die Wohlgeformtheit oder auch die Validität prüft:
- SAXParserFactory factory = SAXParserFactory.newInstance();

B2) Setzten der Validitätsanforderung: factory.setValidating(true);

B3) Anlegen der SaXParser Intanz: SAXParser saxparser1 = factory.newSAXParser();

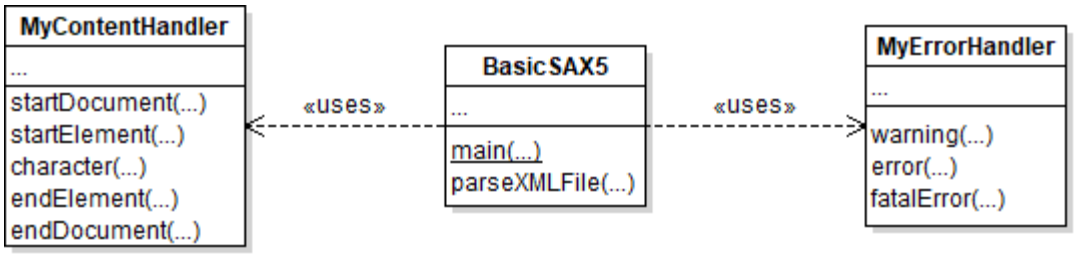
B4) Zuordnung einer XMLReader Instanz:
- XMLReader read1 = saxparser1.getXMLReader();

B5) Verbinden einer ContentHandler Instanz mit der XMLReader Instanz:
- MyContentHandler handler = new MyContentHandler();
- reader1.setContentHandler(handler);

B6) Verbinden einer ErrorHandlerInstanz mit der XMLReader Instanz:
- MyErrorHandler ehandler = new MyErrorHandler();
- reader1.setErrorHandler(ehandler);

B7) Aufruf des Parsers mit Angabe der zu parsenden XML_Datei (Dateiname in URI-Notation):
- String h1 = new File(filename).toURL().toString();
- reader1.parse(h1);

Beispiel Klassendiagramm:



6.3.3 Generierung von SQL-Insert Befehlen aus zeilenähnlichen XML-Elementen:

Allg. Insert-Befehl: Insert INTO tabname (sp1, sp2, ..., spk) VALUES (w1,w2,....,wk)
Konstante Bestandteile: (INSERT INTO)

Variable Bestandteile: (tabname)
Quelle: XML-Element tabname
(bei Methode startElement () gegeben <-> qName.equals("tabname"))

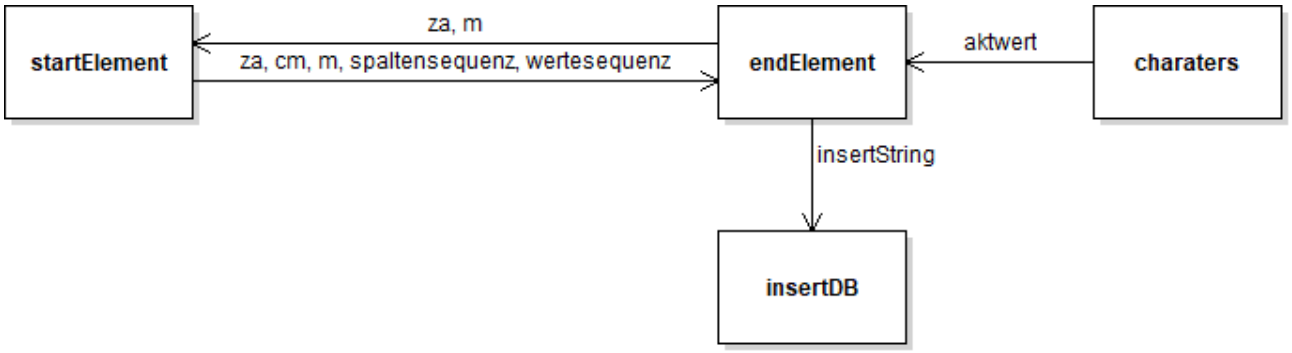
Spaltensequenz: (sp1, sp2, ..., spk)
Quelle: XML-Elemente die innerhalb des XML-Elements Zeile liegen.
- Isolieren des Tag-Namens, um den Spaltennamen spj (1<= j >=k) zu bekommen
- SQL-Datentyp der Spalte spj aus DT einschließen, u, später den zugehörigen Wert wj korrekt aufzubauen

3 Fälle zu unterscheiden:
- 1: num. Datentyp(DT = int, decimal, float,...) <preis...>2.65</preis> => wj, 2.65
Nutzdateneintrag (cm = 0)

-2: Zeichenkette Datentyp (DT = char, varchar,...)<a.bez>Parfüm</Artbez> => wj 'Parfüm'
Klammerung der Nutzdaten mit '...' beachten (cm = 1)

-3 Datus Datentyp (DT_ date,time, timestamp,...) <edat>24.10.2017</edat>
=> wj = TO_DATE('24.10.2017','dd.MM.yyyy') (cm = 3)

Diagramm zum Nachrichtenaustausch zwischen den Call-Back-Methoden:



Aktivitäten in den Call-Back-Methoden:
a) in startElement():
- a.1) Zeilenanfangsverarbeitung: Auslöser: qName="Zeile":
Aktionen:
i) setze za=1;
ii) Initialisierung: spaltenseq "(" ; wertseq = "("
- a.2) Verarbeitung von XML-Elementen in einer Zeile: Auslöser: za=1
Aktionen:
i)Bestimmung von cm: DT = "char" bzw DT = "varchar" => cm=1
DT = "date" bzw DT = "timestamp" => cm3
sonst cm = 3
ii) Pflegen der Spaltensequenz: spaltenseq = spalten[+","]+qName;

b) in characters(): String h = new String (ch, start, length);

aktwert = h;

c) in endElement:
- c.1) Tabellennamen isolieren: Auslöser: qName="tabname": Aktionen:
    (i) insertString = "INSERT INTO " + aktwert;
- c.2) Wertesequenz pflegen: Auslöser: za = 1: Aktionen
    (i) Wertaufbau: cm = 0 => hoch=""; hoch1="";
    cm = 1 => hoch="'"; hoch1="'"; cm=0;

    cm = 2 => hoch="TO_DATE('"; hoch1="', 'dd.MM.yyyy')"; cm=0;

    (ii) Wertesequenz:
        (1) m==0: wertseq=wertseq+hoch+aktwert+hoch1; m=1;
        (2) m>=1: wertseq=wertseq+", "+hoch+aktwert+hoch1; m=m+1;

- c.3) Zeilenendeverarbeitung: Auslöser: qName="Zeile": Aktionen:
    (i): Spalten- und Wertesequenz abschließen: spaltseq=spaltseq+")";
    wertseq= wertseq+")";
    (ii): za=0;
    (iii): INSERT-String fertigstellen: insertSql=insertSql+spaltseq+" VALUES "+wertseq;
    (iv): Aufruf der Methode insertDB mit Übergabe insertSql.

12.04.2018 VL 3 UDATE/DELETE - Kommandos

Donnerstag, 12. April 2018
08:01

6.3.4 Erzeugen von UPDATE- und DELETE Kommandos aus XML-Dateien.

Geg.: Eine XML-Datei zum Generieren eines UPDATE-Kommandos :
Allgemeiner Aufbau eines dafür geeigneten XML-Elements:

<UPDATE TABXX PRIK="w0" dt1="w1" a1="wa1" dt2="w2" a2="wa2" ..... dtM="wM" aM="waM" />

Ziel : Generieren des folgenden UPDATE-Kommandos:

UPDATE TABXX SET a1=v1, a2=v2,...., aM=vM WHERE PRIK=w0

(Vorraussetzung: dt(PRIK)=int), wobei gilt : vi= $\begin{cases} wai <=> dt(ai) ist\ numerisch \\ 'wai' <=> dt(ai) ist\ ein\ Zeichenkettendatentyp \\ TO\_DATE('wai', FORMATSTRING <=> dt(ai) = date \end{cases}$

Struktogramm für den Algorithmus der UPDATE-Generierung (Implementierung in der StartElement()-Methode) :

Anlegen einer AttributesImplementation Instanz : b1 = new AttributesImpl(attributes);

Anzahl der XML-Attribute ermitteln: L=b1.getLength();

Bestimmen des PRIK-Werts und -Namens: w0=b1.getValue(0); PRIK=b1.getName(0);

Setze: i=1, k=1, setKlausel="SET ", m=0;

Solange i<= L-2

- Hole SQL-Datentyp wk der Spalte ak : wk=b1.getValue(i);
- Hole den Wert wak für das Update des zu ändernden Spaltenwerts: wak = b1.getValue(i+1);
- Bilde in Abhängigkeit vom Datentyp wk den korrekten UPDATE-Wert vk:
  - Wk numerischer Datentyp => vk=wak
  - Wk Zeichenkettendatentyp => vk = 'wak'
  - Wk Datumsdatentyp => vk= TO_DATE('wak', FORMATSTRING)
- m=0?
  - J:
    - setKlausel + ak + "=" + vk;
    - m=1;
  - N:
    - setKlausel+","+ak+"="+vk;
- K++
- i=i+2

setKlausel = setKlausel + " ";

Aufbau der WHERE-Klausel : whereK=" WHERE " + PRIK+ "=" + w0;

Zusammenbau des UPDATE-Kommandos: updateK="UPDATE TABXX" + setKlausel+whereK;

Aufruf einer JDBC-Updatemethode: z=jdbcXXupdate(updateK);

6.3.4
Erzeugen von UPDATE und DELETE-Kommandos aus XML-Datei

Geg: Eine XML-Datei zum generieren eines UPDATE-Kommandos

Allgemeiner Aufbau einer dafür geeigneten XML-Elements:
<UPDATETABXX PRIK =
"w0" dt1 = "w1" a1="wa1" dt1 = "w2" a2="wa2" ....dtn = "wn" an="wan"/>

Ziel: Generieren des folgenden UPDATE-Kommandos:

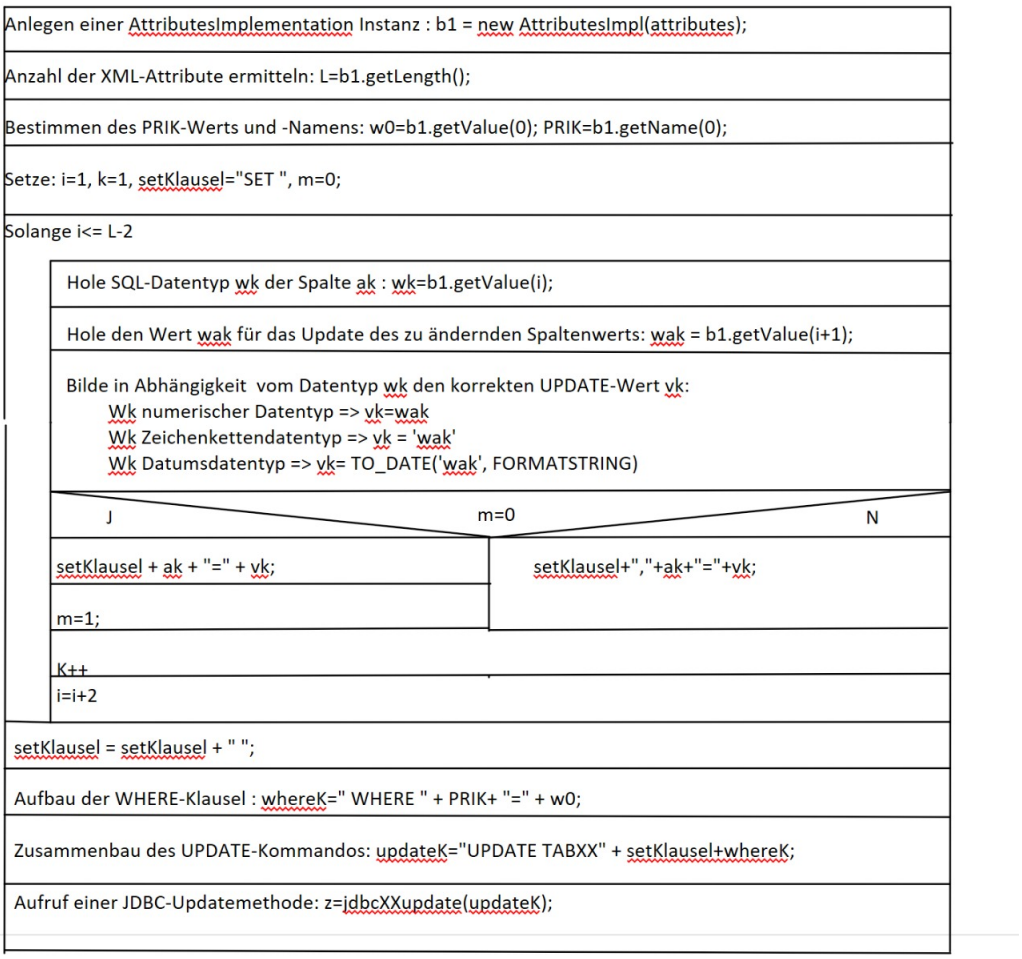
UPDATE TABXX SET a1=v1, a2=v2, an=vn WHERE PRIK =w0
(von: dt(PRIK)=int), wobei gilt :

vi = wai <=> dt(ai) ist nummerisch
'wai' <=> dt(ai) ist ein Zeichenkettendatentyp
TO_DATE('wai',FORMATSTRING <=> dt(ai) ist ein Datum

für alle i mit 1<= i <= N

Anm.
Die

Struktogramm für den Algorithmus der UPDATE-Generierung (Implementierung in der StartElement()-Methode) :



Generierung des DELETE-Kommandos ist ein einfacher Spezialfall des obrigen Algorithmus:
Das gegebene XML-Element hat die Form : <DELTabXX PRIK="w0"/>
Der Delete-Befehl hat die WHERE-Klausel whereK = "WHERE "+PRIK+"="+w0;

6.4 XML-Schema

Bisher haben wir die Grammatik einer Menge strukturgleicher XML-Elemente durch eine DTD beschrieben.
Nachteil einer DTD:
Für die Nutzdaten eines XML-Blattelements gibt es nur den unspezifischen Datentyp #PCDATA. Insebesondere können vom Datentyp ganze und rationale Zahlen nicht von Zeichenketten unterschieden werden.

Ein DTD Eintrag für ein XML-Blattelement UUU hat die Form

<!ELEMENT UUU (#PCDATA)>

Ziel ist die Beschreibung von XML-Elementen in XML-Format selber zu geben.

6.4.1 Datentypen in XML-Schema
XML-Schema bietet elementare Datentypen für Nutzdaten in XML-Elementen an:
a) nummerische Datentypen: int, long, integer, (für ganze Zahlen),
decimal, (für rationale Festpunktzahlen),
float, double (für Gleitpunktzahlen)

b) Zeichenkettentyp : string
c) Datumsdatentypen: date, dateTime

6.3.4
Erzeugen von UPDATE und DELETE-Kommandos aus XML-Datei

Geg: Eine XML-Datei zum generieren eines UPDATE-Kommandos

Allgemeiner Aufbau einer dafür geeigneten XML-Elements:
<UPDATETABXX PRIK =
"w0" dt1 = "w1" a1="wa1" dt1 = "w2" a2="wa2" ....dtn = "wn" an="wan"/>

Ziel: Generieren des folgenden UPDATE-Kommandos:

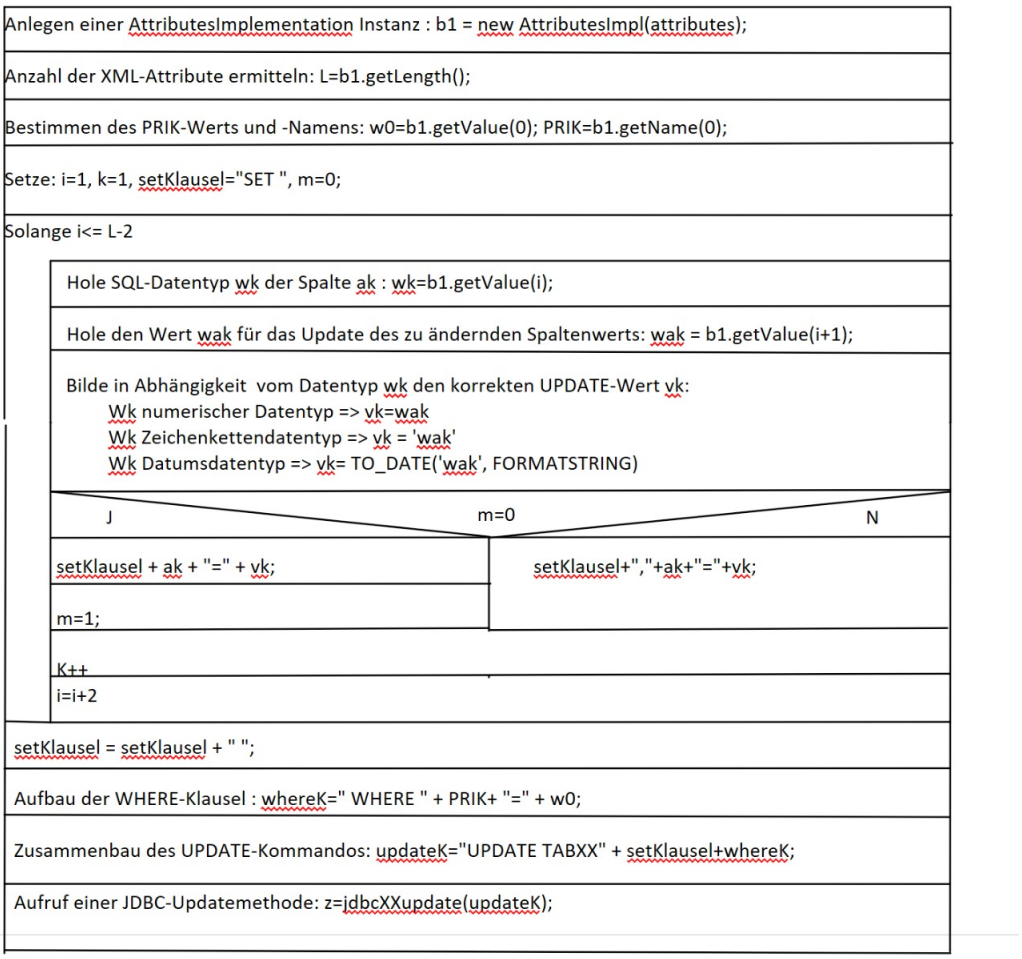
UPDATE TABXX SET a1=v1, a2=v2, an=vn WHERE PRIK =w0
(von: dt(PRIK)=int), wobei gilt :

vi = wai <=> dt(ai) ist nummerisch
'wai' <=> dt(ai) ist ein Zeichenkettendatentyp
TO_DATE('wai',FORMATSTRING <=> dt(ai) ist ein Datum

für alle i mit 1<= i <= N

Anm.
Die

Struktogramm für den Algorithmus der UPDATE-Generierung (Implementierung in der StartElement()-Methode) :



Generierung des DELETE-Kommandos ist ein einfacher Spezialfall des obrigen Algorithmus:
Das gegebene XML-Element hat die Form : <DELTabXX PRIK="w0"/>
Der Delete-Befehl hat die WHERE-Klausel whereK = "WHERE "+PRIK+"="+w0;

6.4 XML-Schema

Bisher haben wir die Grammatik einer Menge strukturgleicher XML-Elemente durch eine DTD beschrieben.
Nachteil einer DTD:
Für die Nutzdaten eines XML-Blattelements gibt es nur den unspezifischen Datentyp #PCDATA. Insebesondere können vom Datentyp ganze und rationale Zahlen nicht von Zeichenketten unterschieden werden.

Ein DTD Eintrag für ein XML-Blattelement UUU hat die Form

<!ELEMENT UUU (#PCDATA)>

Ziel ist die Beschreibung von XML-Elementen in XML-Format selber zu geben.

6.4.1 Datentypen in XML-Schema
XML-Schema bietet elementare Datentypen für Nutzdaten in XML-Elementen an:
a) nummerische Datentypen: int, long, integer, (für ganze Zahlen),
decimal, (für rationale Festpunktzahlen),
float, double (für Gleitpunktzahlen)

b) Zeichenkettentyp : string
c) Datumsdatentypen: date, dateTime

VL 3 XML-Schema

Donnerstag, 12. April 2018
08:53

6.4 XML-Schema

Bisher haben wir die Grammatik einer Menge strukturgleicher XML-Elemente durch eine DTD beschrieben.
Nachteil einer DTD: für die Nutzdaten eines XML-Blattelements gibt es nur den unspezifischen Datentyp #PCDATA.
Insbesondere können vom Datentyp ganze und rationale Zahlen nicht von Zeichenketten unterschieden werden.

Ein DTD-Eintrag für ein XML-Blattelement UUU hat die Form <!ELEMENT UUU(#PCDATA)>

Ziel ist die Beschreibung von XML-Elementen in XML-Format selber zu geben.

6.4.1 Datentypen in XML-Schema:

XML-Schema bietet elementare Datentypen für Nutzdaten in XML-Elementen an:

a. Numerische Datentypen : int, long, integer => für ganze Zahlen
   decimal => rationale Festpunktzahlen
   float, double => für Gleitpunktzahlen

b. Zeichenkettendatentyp : string
c. Datumsdatentyp: date, datetime

A. Definition von XML-Elemente, die von einem einfachen XML-Datentyp sind (z.B. XML-Blattelement) :
   <xsd:element name="TAGNAME" + type="xsd:DTYP"/> [xsd:= XML-Schema-Definition]

BSP.: <xsd:element name="artnr" type ="xsd:integer"/>
<xsd:element name="artbez" type="xsd:string"/>
<xsd:element name="preis" type="xsd:decimal"/>

B. Definition von XML-Elementen mit komplexen Datentypen: Hier gibt es zwei Arten komplexe Datentypen zu bilden:
(i) die Sequenz von Elementen mit bereits definierten Datentypen (xsd:sequence) [entspricht in der DTD dem Tupel-
element Aufbau: <!ELEMENT xxx(A1 c1, A2 c2, ... , AN CN)>]
(ii) die XOR-Auswahl von Elementen mit bereits definierten Datentypen (xsd:choice) [DTD-Entsprechung :
<!ELEMENT xxx(A1 c1 | A2 c2 | .... | AM cM)]

BSP1: Definition des komplexen XML-Elements zeile:

```
<xsd:element name="zeile">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="artnr"/>
            <xsd:element ref="artbez"/>
            <xsd:element ref="preis"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

BSP2: XML-Element Anschrift mit choice auf Postfach oder Straße :

```
<xsd:element name="Anschrift">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Ort" type="xsd:string"/>
            <xsd:element name="PLZ" type="xsd:int"/>
                <xsd:choice>
                    <xsd:element name="Postfach" type="xsd:string"/>
                    <xsd:element name="Strasse" type="xsd:string"/>
                </xsd:choice>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Kap.X: Datenbanken und XML: XML-Schema Datei mit Schema zur Artikeltabelle

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <!-- definition of simple elements -->
<xsd:element type="xsd:string" name="tabname"/>
<xsd:element type="xsd:integer" name="artnr"/>
<xsd:element type="xsd:string" name="artbez"/>
<xsd:element type="xsd:decimal" name="preis"/>
    <!-- definition of complex elements -->
<xsd:element name="tabelleMa">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="tabname"/>
<xsd:element ref="zeile" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="zeile">
<xsd:complexType>
<xsd:sequence>
<xsd:element ref="artnr"/>
<xsd:element ref="artbez"/>
<xsd:element ref="preis"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Kap.X: Datenbanken und XML: XML-Datei (Artikeltabelle) mit XML-Schema-Tags

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<tabelleMa xsi:noNamespaceSchemaLocation="http://www.nt.fh-koeln.de/fachgebiete/inf/buechel/tabelleMa.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<tabname>Artikel</tabname>
<zeile>
<artnr>4711</artnr>
<artbez>Parfüm</artbez>
<preis>10.25</preis>
</zeile>
<zeile>
<artnr>4850</artnr>
<artbez>Seife</artbez>
<preis>0.45</preis>
</zeile>
<zeile>
<artnr>4850</artnr>
<artbez>Kamm</artbez>
<preis>2.65</preis>
</zeile>
</tabelleMa>
```

Festlegung der Stelleanzahlen (p,q) beim Datentyp decimal(p,q)

```
<xsd:element name="preis">
    <xsd:simpleType>
        <xsd:restriction base="xsd.decimal">
            <xsy:totalDigits value="7"/>
                <xsy:furctionDigits value="2"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

BSP: (p,q) = (7,2)

6.4.2 Attribute von XML-Elementen
Allg. Attribut-Definition: Für jedes einzelne Attribut:
<xsd:Attribute name "ATTRIBUTENAME" type "DTYP" use ="....."/>

Anm:
a) Keine use-Angebe des Attribut notwendig (REQUIRED)
b) Angabe : use="optional" => das Attribut ist optional (IMPLIED)

BSP: Ein XML-Element MOTOR mit mehreren Attributen
z.B.:

<MOTOR	HUB="1600"	KW"59.9"	DREH="5900">
//	required	required	implied

Otto-Motor
</MOTOR>

XML-Schema Definition von Motor

```
<xsd:element name="MOTOR" type="xsd:string">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:attribute name="HUB"  type="xsd:int"                    />
            <xsd:attribute name="KW"   type="xsd:float"                  />
            <xsd:attribute name="DREH" type="xsd:int"   use = "optional"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Kap 7 Objektrelationale und Objektorientierte Datenbanken/ abstrakte Datentypen

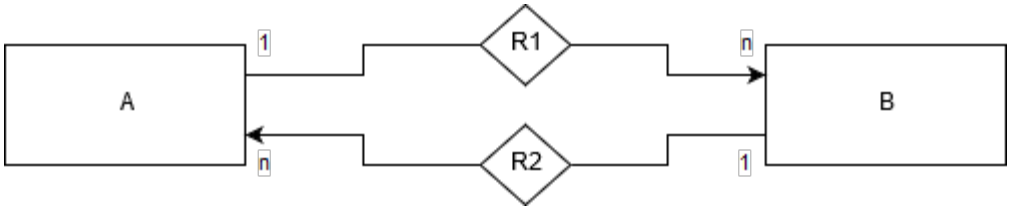
Ausgangspunkt: Mismatch zwischen "flachen" Tabellenstrukturen bei RDB und beliebig komplexe Datentypen in OOP

Anforderungsprofil an ein Objektorientiertes DBMS, um diesen Mismatch aufzuheben (OOPDMS-Manifesto[1990])

1. Das OODBMS soll die Standardanforderungen an ein allgemeines DBMS erfüllen (vgl Kap. 1 der Vorlesung WS)
-1.1 Peristenz
-1.2 Sekundärspeicherverwaltung
-1.3 Transaktionskonzept
-1.4 Datensicherung / Recovery
-1.5 Datensicherheit (Rechteverwaltung)

2. Das OODMMS verfügt über ein objektorientiertes Datenmodell
-2.1 Klassen (komplexe Datentypen, insbes. auch Listen oder Mengen)
-2.2 Objektidentität (OID)
-2.3 Kapselung
-2.4 Vererbung
-2.5 Überladen /Überschreiben von Methoden

7.1 Abstrakte Datentypen:
Wenn man in einem ERD die folgende Situation hatte



musste man zur Modellierung als RDB eine Referenztabelle AtoB definieren mit
RS(AtoB) = {(pa2bid,int,PRIK), (pa,int,FKEY(A.aid)), (pb,int,FKEY(b.bid)),.....}
(Notwendiger Schritt, um das ERD in ANF zu überführen)

Idee: Kann unter Verwendung von Komplexen Datentypen (z.B. Listen) auf die Normalisierung verzichtet werden ?
Antwort: JA!
Weg: Konzept der abstrakten Datentypen (=: ADT)

Das Konzept der ADT besteht aus zwei Teilen
-1 Für das Konzept der ADT wird eine Menge von einfachen Datentypen DT festgelegt:
DT= { int, float, decimal, char, string , boolean}

int: Datentyp für beliebig große ganze Zahlen z Element aus Z
float: Datentyp für beliebig große rationale Zahlen q Element aus Q
decimal: Datentyp für beliebig große rationale Festpunktzahlen
char: Datentyp für Zeichen x eines beliebigen Alphabets
sting: Datentyp für beliebig lange Zeichenketten
boolean: Datentyp für die logischen Werte true, false

-2 Konstruktoren für komplexe Datentypen:
Es gibt vier Arten von Konstruktoren.
TUPLE OF()
SET OF()
LIST OF()
BAG OF()

-2.1 TUPLE OF() Konstruiert einen Tupeldatentyp, der aus bereits definierten komplexen Datentypen oder aus einfachen Datentypen besteht:
Allg. Syntax: sind a1,.....,an Attribute mit bereits definierten Datentypen dt1, dt2,..., dtn
dann ist ein Tupel-Datentyp TDT definiert durch TDT:=TUPLE OF(a1:dt1, as:dt2,...,an:dtn)

BSP.1 Abstrakte Datentyp ARTIKEL:

ARTIKEL:=TUPLE OF(artnr: int, artbez:string, preis: decimal)

- 1.1 Typ ARTIKEL als Schema einer RDB-Tabelle Artikel:
RS(ARTIKEL) := {(artnr, int, PRIK), (artbez, varchar(50), NOT NULL), (preis, decimal(7,2), preis>0)}
- 1.2: Klasse Artikel: `class Artikel { int artnr; String artbez; BigDecimal preis; }`
- 1.3:

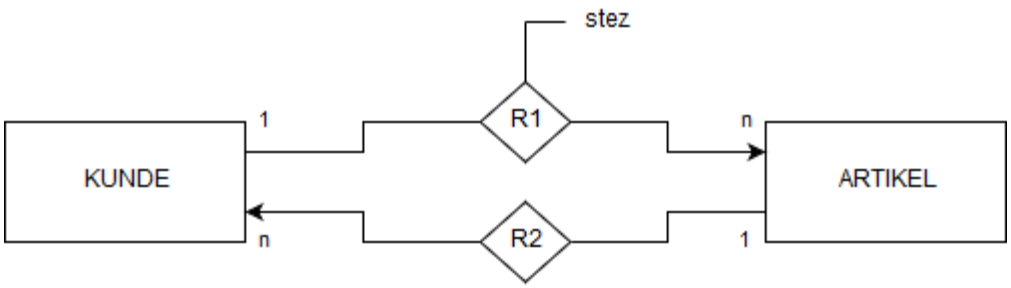
```
<xsd:element name="Artikel">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="artnr"  type="xsd:int"     />
            <xsd:element name="artbez" type="xsd:string"  />
            <xsd:element name="preis"  type="xsd:decimal" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

II. 2) SET OF(): konstruiert einen mengenwertigen Datentype: Die Menge, die durch diesen Datentyp bestimmt wird, enthält keine doppelten Elemente und hat keine Anordnung.
Allg. Syntax: Ist a_1 ein Attribut für ein Element der Menge und ist dt_1 sein Datentyp, dann ist der Mengen-Datentyp SDT definiert als SDT:=SET OF(a_1.dt_1)
BSP.: Eine Klasse SmartRoom für Räume mit Sensoren:

```
class SmartRoom {
    String raumbez;
    Set<Sensor> senset;
}
class Sensor {
    String senbez;
    String phydim;
}
```

Als ADT-Definition:
a) ADT Sensor:
   Sensor := TUPLE OF(senbez : string, phydim : string)
b) ADT SmartRoom:
   SmartRoom := TUPLE OF(raumbez : string, senset : SET OF (x : Sensor))

II. 3) LIST OF(): Konstruiert einen Listen-Datentype: Eine List ist eine angeordnete Sammlung von Knoten, die mehrfach vorkommen können.
Allg. Syntax: Ist a_1 ein Attribut für einen Knoten der Liste und ist dt_1 sein Datentyp, dann ist der Listendatentyp LDT definiert als: LDT:=LIST OF(a_1 : dt_1)
BSP.: ERD:



Legende:

R1: kauft

R2: wird verkauft an

skz: Srückzahl

a) ADT für die Knoten der Artikelliste: ArtKnoten := TUPLE OF(a: ARTIKEL, stez : int)
b) ADT für eine Artikelliste: ArtList := LIST OF(x : ArtKnoten)
c) ADT für Kunden: KUNDE := TUPLE OF(knr : int, kname : string, plz : int, y : ArtList)

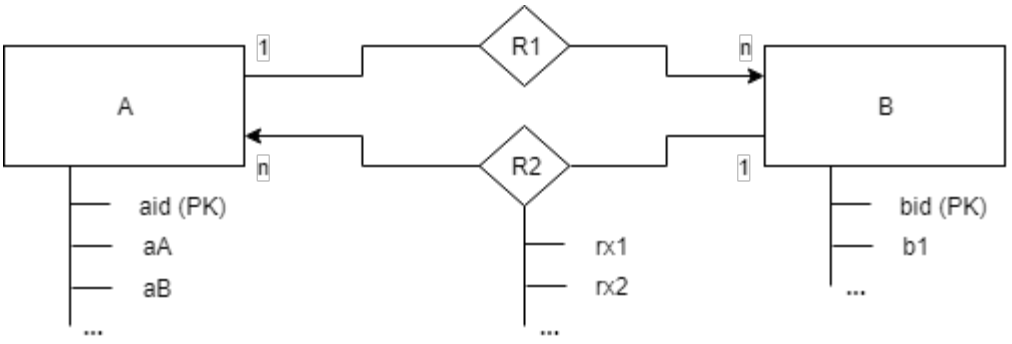
Bem.:

Anordnug vorhanden	**Doppelte Elemente erlaubt**	**Sammlungsdatentyp**
+	+	LIST OF() : Liste
-	-	SET OF() : Menge
-	+	BAG OF() : ("Tasche/Beutel")
+	-	[kein ADT-Konstruktor] : Geordnete Menge (Java: TreeSet)

II. 4) BAG OF(): Konstruiert Datensammlung vom Typ BAG. In einem BAG können mehrere gleiche Objekte vorhanden sein, ein BAG ist ungeordnet.
Allg. Syntax: Ist a_1 ein Attribut für ein BAG-Objekt und ist dt_1 sein Datentyp, dann ist ein BAG-Datentyp BDT definiert als: BDT:=BAG OF(a_1 : dt_1)
BSP.: EINKAUFSTASCHE:= BAG OF(x : ARTIKEL)

7.2 Objektrationale Datenbanken (ORDB)

ORDB sind Erweiterungen von RDB. Sprachnorm SQL2003: Erweiterung von SQL mit Konstrukten, um komplexe Datentypen abzubilden.



a) im Fall einer RDB muss dieses ERD mittels einer Referenztabelle A2B auf eine NF abgebildet werden.

b) mittels ADT kann dieses ERD durch folgende Datentypen modelliert werden:
DTB:= TUPLE OF(bid: int, b1: dtb1;....) (Datentyp für B)
DTB:= TUPLE OF(bid: int, rx1: dtb1; rx2: dt2;....)
(Datentyp für die Knoten der Liste, die als Attribute zu A gehört, um die Realtionships R1/R2 abzubilden)

DTB:= TUPLE OF(Kx: DTK) (ADT für diese Liste)
DTB:= TUPLE OF(aid: int; a1: dta1;....; :lblist:DTL)

1. Definition eines Komplexen Datentyps in einer ORDB:
CREATE TYPE dtypnamen AS OBJECT
(a1 dt1 hierbei sind dti entweder atomare SQL-Datentypen oder
a2 dt2 bereits schon definierte komplexe Datentypen
) am dtm

BSP: DTK für eine Artikelliste:

CREATE TYPE DTKA AS OBJECT (artnr integer, strkanz integer)

// DTKA = Datentypname für Knoten einer Artikelliste

BSP2: Datentyp ANSCHR für Anschriften in einer Mitarbeitertabelle
a) Deinition von ANSCHR :
CREATE TYPE ANSCHR AS OBJECT (plt integer, Ort char (20), strasse varchar(20))
b) CREATE TABLE MITARB (pnr integer PRIMARY KEY, mnam varchar(20), gehalt decimal(7,2) adv1 ANSCHR)

2. Definition eines Datentyps für eine LIST OF (....) Implementation
CREATE TYPE DTL AS TABLE OF DTK

DTL : Datentypname ("Tabellenname") des ORDN-Speichermodell einer Datensammlung (wie LIST OF(...)).

DTK wie in 1. definierter Datentyp der Knoten der Liste

BSP1:

```
CREATE TYPE DARTLI AS TABLE OF DTKA
```

3. Einbau des Speichermodells von 2. als NESTED TABLE in eine Tabellendefinition:

CREATE TABLE A (aid integer primary key, a1 dt,....., lx DTL)NESTED TABLE lx STORE AS phlxTab

Anm;
phlxTab = Physikalische Name der Nested Table.

BSP1: Definition einer Kundentabelle mit einer Artikelliste als NESTED TABLE:
CREATE TABLE KUNDE (knr integer, PRIMARY KEY, kname char (20), kadr ANSCHR, artl1 DARTLI) NESTED TABLE artl STORE AS phNTArt

Anwendung der komplexen Datentypen in der DML:

4. Konstruktoren für komplexe Typen:

Für alle komplexe Typen STX, die in 1. definiert worden sind, gibt es einen Konstruktor:

DTX(w1, w2, w3,...., wk)

DTK = ADT-Name aus 1.
w1 = Wat gemäß Datentyp des 1. Attributs von DTX
wk = Wat gemäß Datentyp des k. Attributs von DTX

Anwendung eines Konstruktors in INSERT-Kommando(in der VALUES-Klausel)
zu BSP 2b)
INSERT INTO MITARB (pns, mnam, gehalt, adr1) VALUES(102, 'Franz Meyer', 4500.50, ANSCHR (50678, 'Koeln','Betzdorferstraße 2'))

zu BSP2:

INSERT INTO KUNDE VALUES (1025, 'SCHMITZ KG' ,ANSCHR(50555, 'KOELN-X', 'AAchnerstraße 999'), DARTLI(DTKA (4711,50), DTKA(4717, 130), DTA(4812, 270))

b) Einfügen weiterer Artikelknoten in die Artikelliste mittels eines geschachtelten SELECTs
INSERT INTO KUNDE (SELECT artl1 FROM KUNDE WHERE KNR = 1025) VALUES (DTKA(4850, 320))

// der 2.SELECT steht in der Spaltenklausel des INSERT

UPDATE:
5. Zugriff auf Attribute eines komplexen Datentyps mit der Punktnotation

zu BSP2 b):

UPDATE MITARBEITER M1 SET M1.Kadr.Strasse='Gremberstraße 17', M1.Kadr.plz = 50677 WHERE M1.pnr = 107

zu BSP 1:
UPDATE einer stckanz (Attribute eines Knotens in einer NESTED TABLE)

UPDATE TABLE (SELECT artl1 FROM KUNDE WHERE knr= 1025)A
SET A.stckanz=290 WHERE A.artnr=4850

zu BSP 2b) Verwendung des Konstruktors als Wertgeber in der SET-Klausel:

UPDATE MITARBEITER M1 SET M1.Kadr= ANSCHR(5311,'Bonn', 'Koelnstraße') WHERE M1.Knr = 107

DELETE: Löschen von Positionen einer NESTED TABLE

DELETE FROM TABLE (SELECT artl1 FROM KUNDE WHERE Knr = 1025)A
WHERE A.artnr = 4850

SELECT: - Nutzung der Punktnotation in der Spaltenauswahl und in der WHERE-Klausel
- Zugriff auf Positionen einer NESTED TABLE mit einem symbolischen Tabellenname

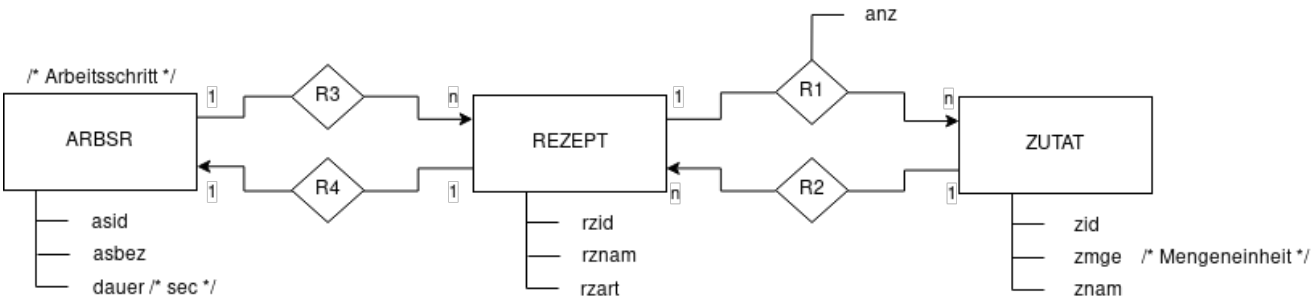
BSP Anzeige aller Kunden mit plz und stckanz, die einen bestimmenten Artikel gekauft haben.

SELECT knr, kadr.plz, B.stckanz FROM Kunde, TABLE(artl1)B WHERE B.artnr = 4711

BSP: Verwendung von Aggregatfunktionen für NESTED TABLE Attribute

SELECT sum(B.stckanz) AS X, B.artnr FROM Kunde, TABLE (artl1)B GROUP BY B.artnr

Ü1 Umsetyung ERD in ORDB-Datentypen:



Legende

R1: enthält
R2: wird eingesetzt in
R3: benötigt
R4: gehört zu

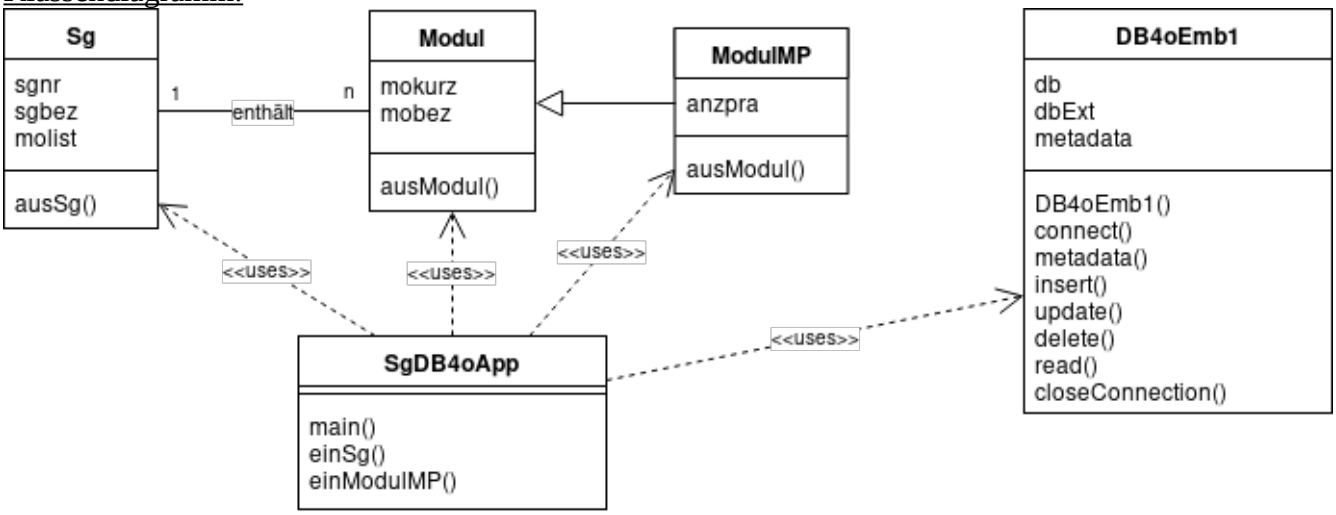
```
01) CREATE TABLE ZUTAT (
      ZID  int PRIMARY KEY,
      ZMGE char(10),
      ZNAM varchar(20)
    )

02) CREATE TYPE ZUPOS AS OBJECT (
      zid INT,
      anz deciaml(7,2)
    )

03) CREATE TYPE ARBSR AS OBJECT (
      asid int,
      asbez varchar(30),
      dauer int
    )

04) CREATE TYPE ZUTAB AS TABLE OF ZUPOS

05) CREATE TYPE ASTAB AS TABLE OF ARBSR

06) CREATE TABLE REZEPT (
      rzid int PRIMARY KEY,
      rznam varchar(30),
      rzart varchar(20),
      zuli ZUTAB,
      arli ASTAB
    )
    NESTED TABLE zuli STORE AS PHZULI,
    NESTED TABLE arli STORE AS PHARLI;

-- ZID: 4711 (Kartoffel), 4712, 4713
07) INSERT INTO REZEPT (rzid, rznam, rzart, zuli, arli)
    VALUES (103, 'Kartoffeln provencalisch', 'vegetarisch',
      ZUTAB (ZUPOS (4711, 2.5), ZUPOS (4712, 3.0)  ),
      ASTAB (ARBSR (300, 'Kartoffel schälen', 15*60)  )
    )
```

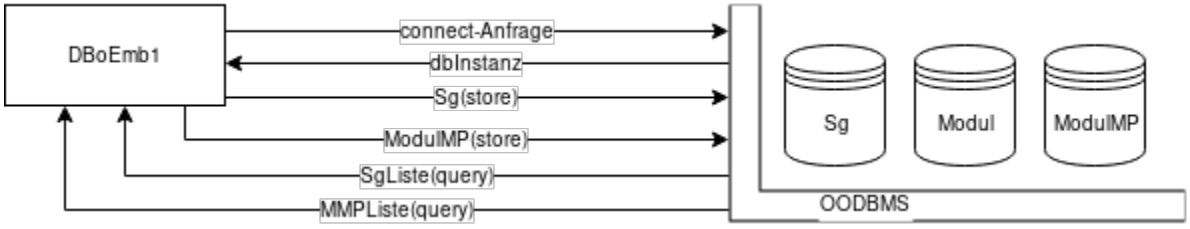
```
08) INSERT INTO REZEPT (SELECT arli FROM REZEPT WHERE rzid=103)
    VALUES ( ARBSR (301, 'Kartoffeln kochen', 20*60)  )

09) -- UPDATE eines Attributwerts in einer NESTED TABLE:
    -- z.B. dauer=1800 von asid=301 in rzid=103
    UPDATE TABLE ( SELECT arli FROM REZEPT WHERE rzid=103) A
    SET A.dauer=1800 WHERE A.asid=301

10) -- Alle Rezepte, die Arbeitsschritte beihalten,
    -- die 'Kartoffeln ...' bearbeiten (asbez), anzeigen
    SELECT rzid, rznam, X.*, B.* FROM REZEPT, TABLE(arli) X, TABLE(zuli) B
    WHERE X.asbez LIKE '%Kartoffel%' AND B.zid=4711
```

Kap. 7.3 Beispiel eines OODBMS (db4Objects)

(OpenSource-Lizenz):
+ Persistente Klassen können als Java-Klassen definiert werden.
+ Vererbung
+Metadatenverwaltung

BSP.: OODB für Studiengänge:
Entitätsklassen: Sg (Studiengänge), Modul (Module), ModulMP (Module mit Praktikum)
Klassendiagramm:





Kapitel 9: NoSQL DBMS

9.1 Überblick

NoSQL := not only SQL

Anforderungen

a) Horizontale Skalierbarkeit: Verteilung der Datensätze der Datenbank auf mehrere Rechnerknoten, um Massenanfragen schnell beantworten zu können.
(Amazon, Google, …)
b) Kein starres Schema wie bei einem RDBMS. Neue Attribute sollen bei INSERT-Aufrufen direkt angelegt werden können.
(“Schemafreiheit” (übertriebenes Postulat). Besser: schwares Schema, da es Minimalanforderungen an ein Schema gibt.)

Google Big Table Konzept

Alle Einträge in einer DB Entität jaben den folgenden Aufbau:

Triple: (KEY, VALUE, TIMESTAMP)

KEY := Entität-/Attributname

VLAUE := Wert gemäß ADT-Defninition des Attributs bzw. Der Entität

TIMESTAMP := Zeit/Datumsangabe

BSP.:

a) KEY = “Ort”, dt(“Ort”) = String
=> Triple(“Ort”, “Köln”, “21.09.2017 12:34”)

b) KEY = “Artikel”, dt(“Artikel”) = TUPLE OF(artnr: int; artbez = String; preis: decimal(7,2))
=> ADT-Triple:
(“Artikel”, 1117-03, ((“artnr”, 4711, T1), (“artbez”, “Brot”, T2), (“preis”, 1.25, T3), T4))
T1-T4 := TimeStamps

Familien von NoSQL-DBMS

I) Reine Key-Value DBMS: Berkeley DB.

II) Wide Column Stores: haben eine Zerlegung der Datenbank in Segmente. Ein Segment heißt Column-Family. Die Datensätze einer Column-Family könnenals komplexe ADT strukturiert sein, insbesondere mit unterschiedlichen Attributen.
BSP.-DBMS: Cassandra, Hadoop/Hbase.

BSP.: Column-Family: ARTIKEL (in Cassandra):

(id: 10; (artbez: “Seife”); (preis: 3.95); (sortiment: “Waschmittel”); T1)
(id: 12; (artbez: “Milch”); T2)
(id: 14; (artbez: “Clorix”); (gefahrgut: 1025); T3)

T1-T3 := TimeStamps

Anm: Der schwache Aufbau des DB-Schemas bei NoSQL-DB erfordert eiin Schema-Monitor, wenn man die unterschiedlichen Attribute, die die Anwender einfügen, kontrollieren möchte.

III) Dokumentorientierte DBMS: z.B. MongoDB (BSON), CouchDB (JSON).

Schema-Anforderungen einer CouchDB:

- Segmentname
- Jeder Datensatz eines Segments ist eine JSON-Datei. Jeder Datensatz hat folgende zwei Pflichtattribute:

  a) id (Eindeutiger Bezeichner des Datensatzes; Kann vom Benutzer vergeben werden; Default: id-Wert wird vom CouchDBMS als URI-Wert vergeben).
  URI := Uniform Resource Identifier (W3C).

  b) rev (Revisionsnummer; wird immer vom DBMS vergeben)

  Dient zwei Zwecken:

  i) Unterscheidung gleicher Datensätze, die als Kopien auf mehreren Rechnerknoten bei horizontaler Skalierung verteilt werden.

  ii) Unterscheidung von Versionen beim UPDATE auf einem Datensatz (d.h. alte Versionen eines Datensatzes gehen nicht verloren)

9.2 Das Dokumentenaustauschformat JSON

JSON := JavaScript Object Notation

JSON ist eine Alternative zu XML. JSON erlaubt eine strukturierte Datenverwaltung, d.h. es git ein JSON-Schema für eine Menge strukturgleicher JSON-Dolkumente.

9.2.1 Sie Syntax von JSON (Wohlgeformtheit)

A) Zeichensatz: UTF-8 (UTF-16, UTF-32).

Geschützte Sonderzeichen:

a1 "…" : Paar von Anführungszeichen (das Paar von Anführungszeichen bestimmt eine Zeichenkette).

a2 : : Doppelpunkt (der Doppelpunkt trennt ein KEY-VALUE-Paar).

a3 , : Komma (das Komma trennt Komponenten innerhalb eines JSON-Arrays).

a4 \ : Backslash (der Backslash dient zur Einleitung einer Escape-Sequenz).

B) Regeln:

b1 Ein JSON-Dokument besteht aus einem Objekt. Dieses Objekt ist die Wurzel des JSON-Dokuments. Dieses Objekt umschließt die Folge der im Dokument gespeicherten Nurzdaten (Memberfolge: <MEMBERF> (BNF)) :
<OBJEKT> := {} | { <MEMBERF> } (Produktionsregel in BNF)

b2 Eine Memberfolge MEMBERF ist eine Folge von Key-Value-Paaren =: KVP (Schlüssel-Wert-Paare), die voneinander durch ein Komma getrennt sind.
<MEMBERF> := <KVP>[{,<KVP>}] (BNF) _..._: BNF-Operatoren

b3 Ein Key-Value-Paar (Property bzw. JSON-Attribut) besteht aus einem Key, der als Zeichenkette den Attributnamen enthält, und aus einem Wert (Value), der vom Key durch einen Doppelpunkt abgetrennt ist: <KVP> := <KEY>:<VALUE>

b4 Eine Zeichenkette <ZKETT> ist eine beliebige Folge nicht geschützter Zeichen $a_1,\ldots,a_n$, die in einem Paar von Anführungszeichen eingeschlossen sind: <ZKETT> := "$a_1a_2...a_n$"

b5 Der Key ist eine Zeichenkette: <KEY> := <ZKETT>

b6 Ein WERT kann eine Zeichenkette, eine ganze Zahl (<GZAHL>), eine rationale Zahl (<GPZ>), ein null-Wert (Schlüsselwort: *null*), ein boolscher Wert (Schlüsselwörter: *true, false*), ein JSON-Array (<JARRAY>) oder ein JSON-Objekt (<OBJEKT>) sein:
<WERT> := <ZKETT> | <GZAHL> | null | true | false | <JARRAY> | <OBJEKT>

b7 Die Zahl werte bestehen aus Dezimalziffern, Vorzeichen und Exponentenzeichen (e, E)
<GZAHL> := [-]$z_1z_2...z_n$ mit $z_i \in \{0,\ldots,9\}$ für $i \le z \le n$, $z_1 \in \{1\ldots..9\}$
<GPZ> := <GZAHL>.$z_2...z_n$[(e|E)[-]<GZAHL>] (BNF) _..._: BNF-Operatoren

b8 Ein JSON Array ist eine beliebige Folge von Werten, leere Arrays sind zulässig:
<JARRAY> := [] | [<WERT>[{,<WERT>}]] (BNF) _..._: BNF-Operatoren

BSP.: JSON-Dokument für einen Skat-Abend:

```
{
	"Skatabend": "23.05.2016 20:00-22:00 Uhr",
	"Ort": {
		"Lokalname": "Parkstübchen",
		"PLZ": 50787,
		"Strasse": "Betzdorfer Str. 3"
	},
	"Spieler": [{
		"Name": "Karl Schmitz",
		"Punkte":  121
	},
	{
		"Name": "Peter Müller",
		"Punkte": -495
	},
	{
		"Name": "Frieda Schulz",
		"Punkte": 374
	}],
	"Schöner Abend": true
}
```

9.2.2 Die Semantik von JSON-Dokumenten (Validität)

Die Semantik einer Klasse strukturleichter JSON-Dokumente wird durch ein JSON-Schema beschrieben. Das JSON-Schema hat selber den Aufbau eines JSON-Dokuments. In einem JSON-Schema werden für Attribute Datentypen festgelegt. Damit sind JSON-Dokumente strukturierte Dateien.

1.) JSON-Attribute mit einem einfachen Datentyp EDTYP:
EDTYP ∈ { integer, number, boolean, string }
- integer für z ∈ ℤ
- number für q ∈ ℚ
- boolean für w ∈ `{ true, false }`
- string für Zeichenketten

Definition eines JSON-Attributs A von eonem eonfachen Datentyp EDTYP:
`"A" : { "type": EDTYP }`

2.) JSON kennt zwei Arten von Komplexen Datentypen:
a) Felder: array
b) Objekte: object

zu a) Definition eines Attributs F vom Typ eines Feldes:
`"F": { "type": "array", "item": KDTYP`*`, "minItems": N, "maxItems": M, "uniqueItems": w`* `}` *optional*
zu b) Definition eines JSON-Attributs Z vom Typ eines Objektes:
`"Z" : { "type": "object", "properties": { DER_OBJEKTATTIBUTE } }`

BSP.:
```
ADT LEBMIT := TUBLE OF(LNAME: string, zus := TUPLE OF(KOMPN: string, PROZ:
decimal(p, q)))
```
Darstellung von LEBMIT als JSON-Objekt:
```
"LEBMIT": { "type": "object", "properties": { "LNAME": { "type": "string"},
"zus": { "type": "object", "properties": { "KOMPN": { "type": "string" },
"PROZ": { "type": "number" } } } } }
```

3.) Für definierte JSON-Attribute kann festgelegt werden, ob die notwendig im Dokument bzw. In Angabe: `"required": [ "A1", "A2", …, "Ak" ]`

4.) Angabe, ob weitere JSON-Attribute, die nicht im Schema des Objekts bzw. Des Dokuments definiert sind, vorkommen dürfen:
```
true  ⇔ sie dürfen vorkommen
false ⇔ sie dürfen nicht vorkommen
"additionalProperties": w
```

5.) Der Rahmen eines JSON-Schemas ist dadurch gegeben, dass jedes JSON-Dokument ein unbenanntes Objekt ist, das aus einer Liste von Attributen besteht LIATT:
`{ "schema": "SCHEMANAME", "description": "…", "properties": { LIATT } }`

9.3 Das kleine 1x1 der CouchDB Anwendungsprogrammierung

1. DB-Verbindung herstellen:

   a) Das Java-Anwendungsprogramm ist ein Http-Client des CouchDBMS, das auf einem Server (IP-Adresse + Port) installiert ist.
   ⇒ Eine Instanz der Klasse HttpClient erzeugen.

   b) Mit einer CouchDbConnector-Instanz die Verbindung zu einem CouchDB-Segment SEGNAME aufbauen.

2. INSERT:

   Einen neuen Datensatz, d.h. ein neues JSON-Dokument, in das in *1b)* ausgewählte CouchDB-Segment einfügen. Der einzufügene Datensatz wird in seinem Aufbau durch eine EKTORP-Java-Klasse bestimmt:

   Notwendige Attribute: **`_id`** (für fen PRIK des Dokuments), **`_rev`** (für die Revisionsnummer des Dokuments).

   Weitere Attribute: je nach DB-Design für die Verwaltung der Nutzdaten (BSP.: Klasse Sensor).

   a) Eine Instanz für die zugehörige EKTORP-Java-Klasse anlegen.

   b) Mit der Methode `db.create()` die Instanz in das DB-Segment einfügen.

3. DELETE:

   a) Direktzugriff auf den zu löschenden Datensatz (unter Angabe des **`_id`**-Werts):
   mit der Methode `db.get(KLASSE, _id)`

   BSP.: `Sensor x = db.get(Sensor.class, idsen1);`

   b) Löschen des Datensatzes unter Verwendung des Rückgabewerts von *3.a)* mit der Methode `db.delete(x);`

4. UPDATE:

   a) Mit `set()`-Methoden EKTORP-Java-Klasse neue Attributwerte setzten.

   b) Mit der Methode `db.update(x)` den UPDATE ausführen.

   Anm.: `x` wurde durch Direktzugriff, wie in *3.a)* beschrieben, erzeugt.

5. Lesen aller Instanzen eines CouchDB-Segments: (≙ `SELECT * FROM SEGNAME`)

   a) Eine ViewQuery-Instanz `vq` für alle Dokumente des zulesenden Segments anlegen.

b) Man benötigt eine Listen-Definition mit Knotentyp = EKTORP-Java-Klasse.

c) Das Lesen aller Instanyen mit der Methode `db.queryView(vq, KLASSE)`

BSP.:
```
List<Sensor> senli = db.queryView(vq, Sensor.class);
```

d) Auswertung / Ausgabe der Liste.

6) Import von JSON-Dokumenten in ein CouchDB-Segment

Vor: Eine CouchDbConnector Instanz zum Zugridd auf ein CouchDB-Segment ist gegeben:
BSP.: CouchDbConnector db = dbInstanze.createConnector("HomeAutomation", true);

(1) Gegeben: JSON-Dokument ohne die Systemattribute_id und _rev.
(2) Arbeitsschnritte:
    (2.1) JSON-Dokument aks InputStream öffnen:
        - File fx = new File(EDSN);
        - InputStream ipst = new FileInputStream(fx);
    (2.2) Erfassen eines _id-Werts (iddoc)
    (2.3) Import ausführen: db.update(iddoc, ipst, fx.length(), null);
        - iddoc ≙ _id-Wert
        - ipst ≙ JSON-Datei als Input-Stream
        - fx.length() ≙ Länge der JSON-Datei in Anzahl Bytes

7) Export von JSON-Dokumente aus der CouchDB

Vor:
a) Eine CouchDbConnector Instanz
b) _id-Wert des zu exportierenden Datensatz (zu exportierenden JSON-Dokument) idex1.

Arbeitsschritte:
(1) Das zu exportierende JSON-Dokuments aus der DB als JSON-Note in den RAM laden:
        JsonNode jx1 = db.get(JsonNode.class, idex1);
(2) JSON-Ausgabedateiname ADSN.JSON festlegen und damit eine BufferedWriter-Instanz öffnen:
        BufferedWriter bjx = new BufferedWriter(new FileWriter("ADSN.JSON"));
(3) Den JSON-Node in die BufferedWriter-Datei schreiben:
        bjx.write(jx1.toString());
(4) Dateiverbindung schließen.

8) SELECT mit MAP-REDUCE (≙ lesen mit einer WHERE-Klausel)
BSP.: SELECT _id, senbez, preis FROM Homeautomation WHERE senbez LIKE SUWORT
MAP: Projektion der gesamten DB-*Segment auf die ausgewählten Attribute
REDUCE: Das Projektionsergebnis wir gemäß einer logischen Bedinngung reduziert.
Verfahren: Für die Abfrage muss ein JavaScript-Dokument (Degindokument) erstellt werden, das eine JavaScript Funktion fpr die Durchführung des MAP-REDUCE-Verfahrens enthält.

ANM: Das DBMS der CouchDB enthält einen JavaSkript Interpreter, der diese JavaSkript Funktion parst und ausführt.

Arbeitsschritte:

(1) Erfassen eines Arguments für die logische Bedingung des REDUCE (hier: SUWORT-Wert)
(2) Anlegen eines Designdokuments mit Namen DDK

```
if (db.contains("_design/DDK")) {
        dd = db.get(DesignDocument.class, "_design/DDK");
        /* Zusammenbau der JAVA-Skript Funktion für die Durchführung des MAP_REDUCE
Verfahrens: */
        String jsuchfn =        "function(doc) {\n" +
                                "       var senbez = doc.senbez;\n" +
                                "       if (senbez.indexOf('" + SUWORT + "') == 0) {\n" +
                                "               emit(doc._id, [doc.senbez, doc.preis]);\n" +
                                "       }\n" + // if
                                "}"; // function
```

function(doc) ≙ Start der Funktionsdefinition
var senbez ≙ Festlegung des Attributs, auf das sich die logische Bedingung des REDUCE bezieht (der WHERE-Klausel)
senbez ≙ Name des zugehörigen JSON Attributs
senbez.indexOf('" + SUWORT + "') == 0 ≙ logische Bedingung des REDUCE (der WHERE-Klausel)
doc._id, [doc.senbez, doc.preis] ≙ Attribute der Spaltenauswahl ([..] JSON-Array, weil eventuell mehrere Attributeinträge möglich sind).

(3) MAP-REDUCE Funktion dem Design-Dokument zuordnen:
    - DesignDokument.View v1 = new DesignDocument.View(jsuchfn);
    - dd.addView(VIEWNAME, v1);
    - // VIEWNAME ≙ symbolischer Viewname (zur Anzeige in der CouchDB)
(4) Fertigstellen des Dokuments: db.create(dd);
(5) Durchführen der Query-Anfrage:
    - Anlegen einer ViewQuery-Instanz:
        - ViewQuery q1 = new ViewQuery().designDocId("design/DDK").viewName(VIEWNAME);
    - Ausführung der Query-Anfrage:
        - ViewResult r1 = db.queryView(q1);
(6) Auswertung der Ergebnisliste:
        - List<ViewResult.Row> erglist = r1.getRow();
    - Durchlaufen der Ergebnisliste:
        - for (int ix = 0; ix < erglist.size(); ix++) {
        - ViewResult.Row y = erglist.get(ix);
        - System.out.println("ID="+y.getKey()+", VALUE="+y.getValue());
        - }
        - // y.getKey(), y.getValue() ≙
        - // Auswertung einer Ergebniszeile nach dem Key-Value-Pinzip
        - // (Key ≙ _id, Value ≙ alle übrigen Attribute der Spaltenauswahl)

9) Schema Monitoring

In einer NoSQL-DB hat man in Unterschied zu einer RDB i.d.R. keine strenge Kontrolle auf die Attribute der Nutzdaten. Es jönnen während der Laufzeir durch die Anwender neue Attribute hinzukommen. Um das Auftreten neuer attribue zu beobachten und ggf. Zu intervenieren braucht man einen Schema Monitor.

Vor: Man benötigt eine Liste aller Datensätze (JSON-Dokumente) eines DB-Segments.
(BSP.: List<Sensor> senlist;)

Arbeitsschritte:

(1) Jeden Knoten dieser Liste als JSON-Note darstellen:
- JsonNode docJN1 = db.get(JsonNode.class, sen1.getId());

(2) Dem JSON-Node einen Iterator zuordnen, der auf akke Attribbute des zugehörigen JSON-Dokuments zugreist:
- Iterator<String> = itJN1 = doc.getFieldNames();

(3) Schleife zur Auswertung der Iteratorergebnisse (d.h. pro Dokument: Liste aller Attributnamen):
- while (itJN1.hasNext()) {
- System.out.println(":::"+itJN1.next());
- }

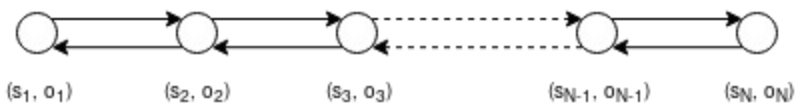
Kapitel 8: Algotithmen der Sekundärspeicherverwaltung / Bayer-Bäume

8.1 Schlüssel und Offsets

Gegeben ist ein DB-Segment mit N Datensätzen d_i. Jeder Datensatz ist auf einem Sekundärspeicher über ein Offset o_i zugreifbar. D. h. in einer Indexverwaltung für einen Direktzugriff ist das Schlüsselpaar (PRIK(d_i), o_i) zu verwalten.

BSP.: (WS) ISAM-Verwaltung der Schlüsselpaare in einer doppelt verketteten List:

s_i:=PRIK(d_i)



Diese Liste ist sortiert nach den Schlüsselwerten: $s_1 < s_2 < \ldots < s_N$

Zugriffsmöglichkeiten auf der Liste, um zu einem Benutzerwert bw einen Schlüsselpaar (s_i, o_i) zu finden:

a) Linieare Suche in der Liste => Aufwand O(N/2)

b) Binäre Suche in der Liste => Aufwand O(ld(N))

Zugriffsaufwand:

A) Laden der Indexliste vr der Arbeitssitzung

B) Suchaufwand (wie z.B. in a), b))

C) 1 Direktzugriff auf den Datensatz d_i über den Offset o_i.

8.2 Bayer-Bäume

[R. Bayer, E. M. McCreight: "Organisation and Maintenance of Lage Ordered Indexes" (1972), AeraInformatica 1, S. 173-189]

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

(1) Jeder Knoten enthält als Färbung höchstens 2M Schlüsselpaare (s_i, o_i).

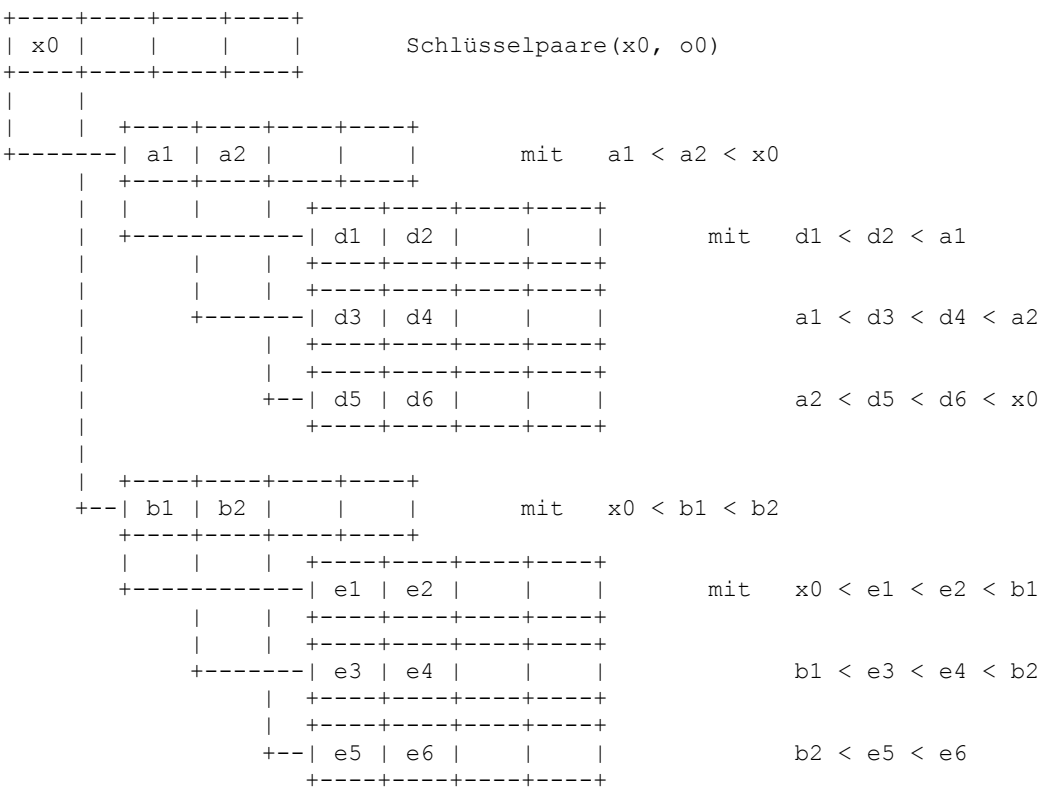
(2) Jeder Knoten außer der Wutzel enthält mindenstens M Schlüsselpaare. Die Wurzel enthält mindestens ein Schlüsselpaar.

(3) Jeder Knoten, wenn der Knoten kein Blatt-Knoten ist, hat (k+1) Nachfolger, wenn er k Schlüsselpaare enthält.

(4) Alle Blätter liegen auf der gleichen Hierachieebene des Baumes.

BSP.: Ein BayerBaum der Ordnung M=2 mit minimal besetzten Knoten:

Wurzel W:



8.3 Einfügen eines Schlüsselpaares in einen Bayer-Baum

Geg.:

(a) Ein Bayer-Baum der Ordnung M.

(b) Ein Schlüsselpaar (m_x,o_x) das eingefügt werden soll.

(c) Ein Knoten k_0 im Bayer-Baum, in den gemäß Sortierordnung der Schlüssel m_x aufzunehmen wäre.

Anm. (BSP.): zu C): Wenn $e_1 < m_x < e_2$ => m_x ist in dem Knoten k_0 aufzunehmen =>

```
      +----+----+----+----+
k0: | e1 | mx | e2 |    |
      +----+----+----+----+
```

Verfahren: Fallunterscheindung: (einfuegen (k_0, m_x)):

I.) k_0 hat a < 2M Schlüssel: => m_x wird gemäß Sortierordnung on k_0 eingefügt.

II.) k_0 hat a = 2M Schlüssel: => Folgende Sritte sind auszuführen:

i) Bilde die Menge aller Schlüssel auf k_0 und m_x in Sortierordnung:

Schlüsselmenge: $S(k_0, m_x) = \{ s_1, s_2, \ldots, m_x, \ldots, s_{a-1}, s_a \}$ $| S(k_0, m_x) | = a+1 = 2M+1$

$= \{ t_1, t_2, \ldots, t_k, t_{k+1}, \ldots, t_{2M+1} \}$

ii) Bestimme in $S(k_0, m_x)$ den mittleren Schlüssel

t_j => $S(k_0, m_x)$ $\{ t_1, t_2, \ldots t_{j-1} \}$ $\cup$ $\{ t_j \}$ $\cup$ $\{ t_{j+1}, \ldots, t_{2M+1} \}$

$\{ t_1, t_2, \ldots t_{j-1} \}$ $\triangleq$ *Schlüsselmenge für einen neuen Knoten k_1*

$\{ t_{j+1}, \ldots, t_{2M+1} \}$ $\triangleq$ *Schlüsselmenge für einen neunen Knoten k_2*

iii) Bilde die Knoten k_1 mit den Schlüsseln $t_1, \ldots, t_{j-1}$, k_2 mit $t_{j+1}, \ldots, t_{2M+1}$

Lösche den Knoten k_0.

iv) Versuch, den Schlüssel t_j in den Parent-Knoten k_p von k_0 einzufügen:

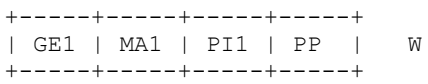
a) k_0 ist nicht der Wurzeöknoten, d.h. k_p existiert => Rekursiver Aufruf einfuegen(k_p, t_j)

b) k_0 ear der Wurzelknoten => Bilde eine neue Wurzel w mit einzigem Schlüssel t_j (d.h. k_1, k_2 sind die Kinderknoten der Wurzel w).

BSP.: Aus der folgenden Schlüsselsequenz MSEQ soll ein Bayer-Baum der Ordnung M=2 aufgebaut werden:

MSEQ = { PI1, MA1, GE1, GE1, PP, PI2, GE2, FSA, BVS1, DN1, DE, DB, AD, BVS2, RA, SWP, DB2, VMA, EKS, DN2, DSS }

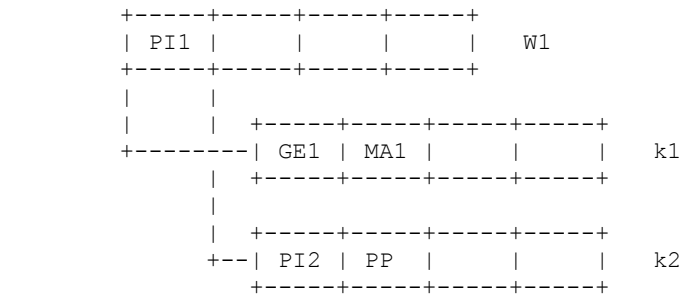
1.



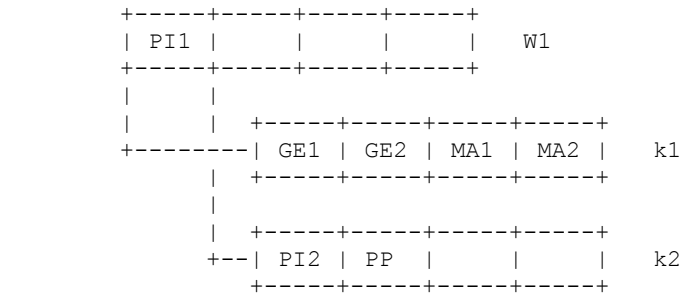
2.

PI2 => S(W, PI2) = { GE1, MA1, PI1, PI2, PP }
k1 W k2

=> neue Wurzel: W1:

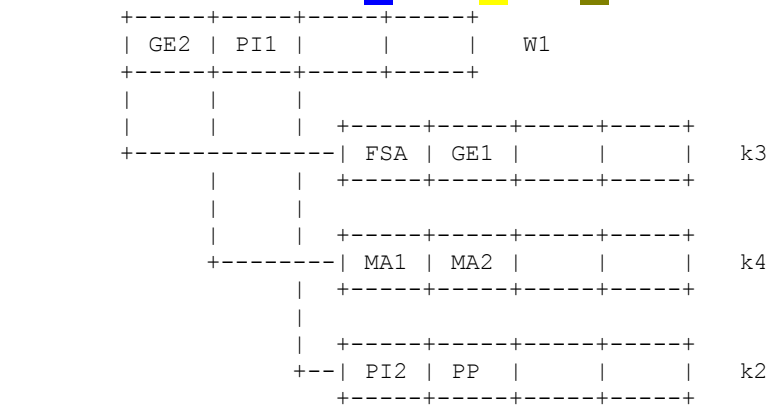


MA2 -> k1
GE2 -> k1

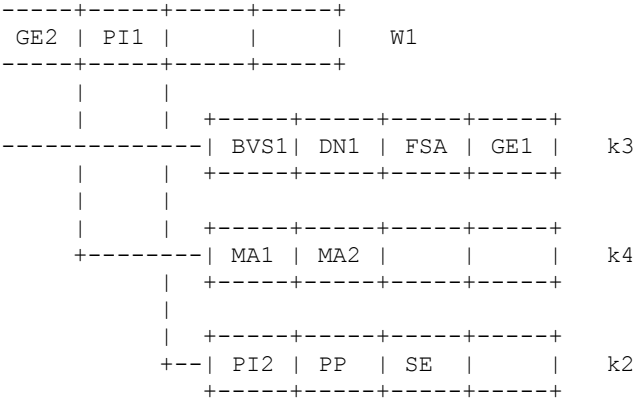


3.

FSA => S(k1, FSA) = { FSA, GE1, GE2, MA1, MA2 }
k3 W1 k2

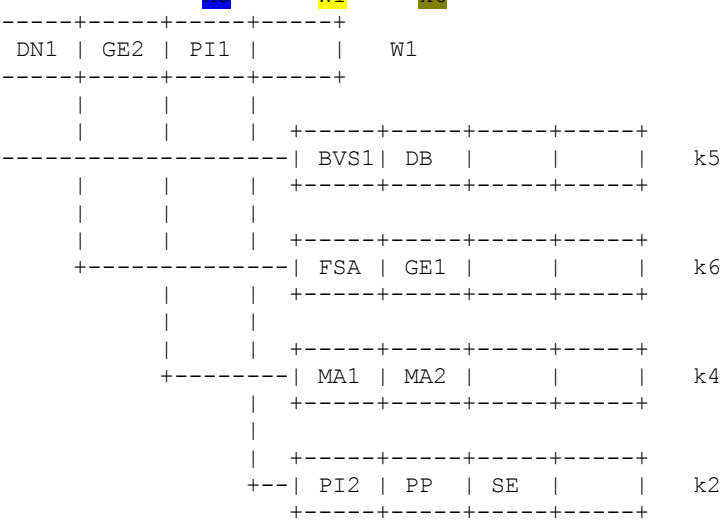


```
BVS1 -> k3
DN1  -> k3
SE   -> k2
```

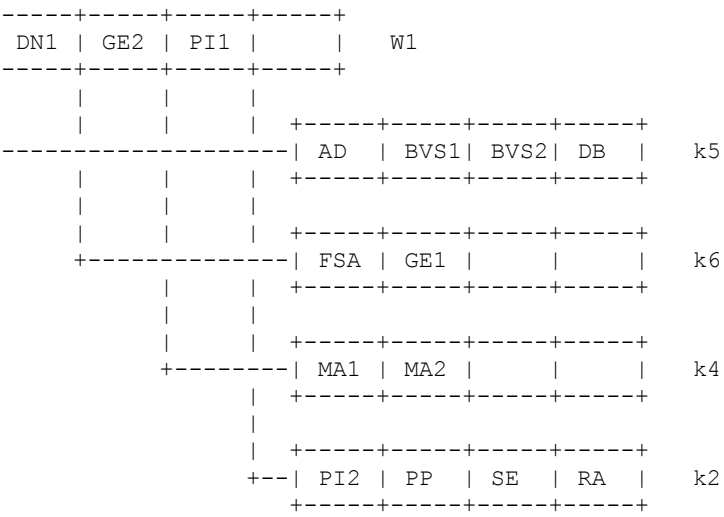


4.

DB => S(k3, DB) = { BVS1, DB, DN1, FSA, GE1 }
k3 W1 k6

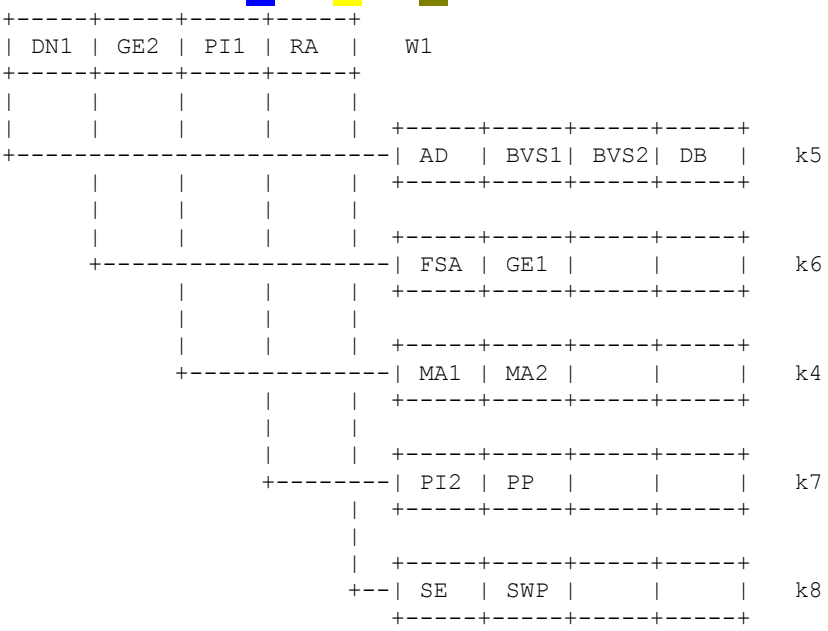



```
AD   -> k5
BVS2 -> k5
RA   -> k2
```



5.

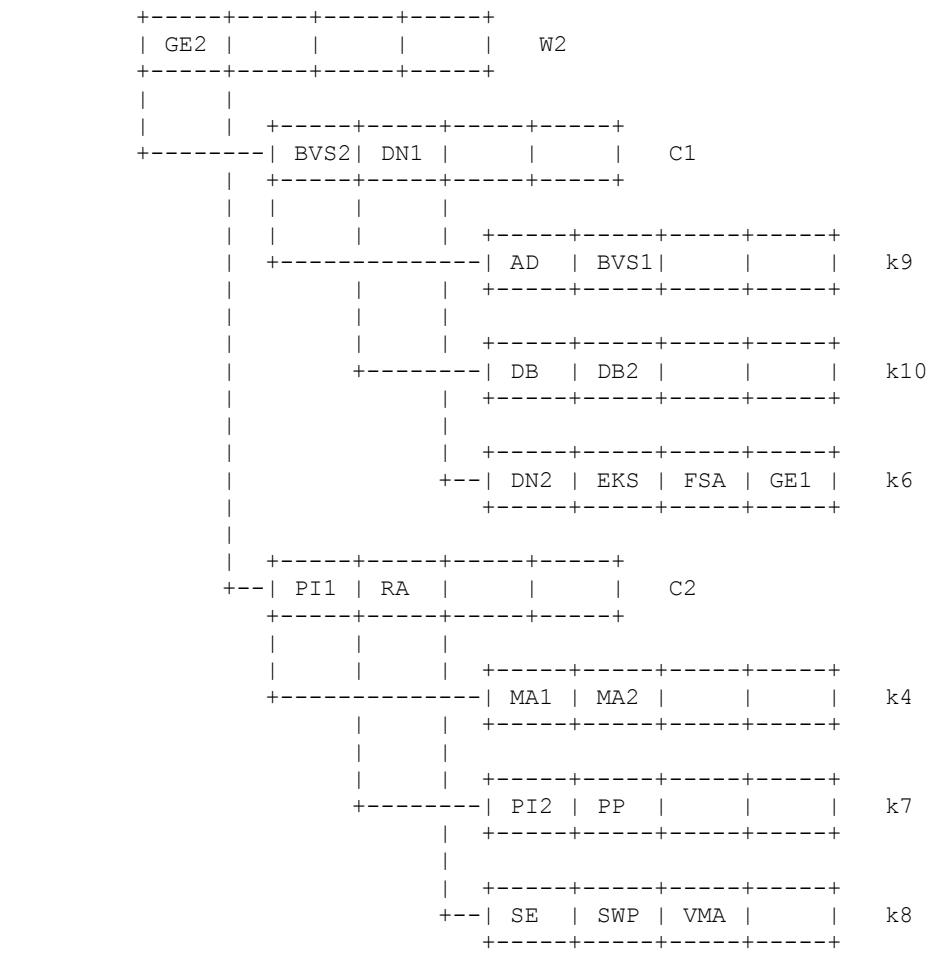
```
SWP => S(k2, SWP) = { PI2, PP, RA, SE, SWP }
                      k7      W1     k8
```



6.

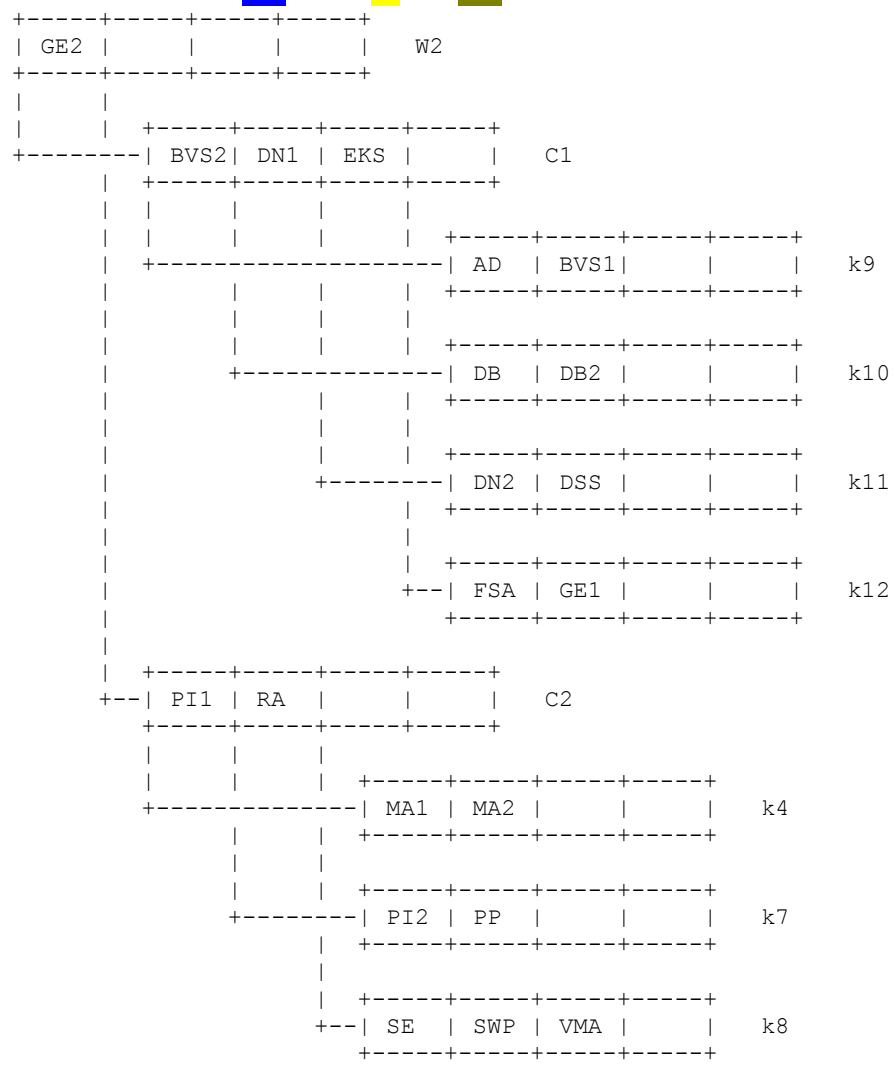
```
DB2  => S(k5, DB2)  = { AD, BVS1, BVS2, DB, DB2 }
                        k9      W1      k10
BVS2 => S(W1, BVS2) = { BVS2, DN1, GE2, PI1, RA }
                        C1      W2      C2
```

VMA -> k8
EKS -> k6
DN2 -> k6



7.

DSS => S(k6, DSS) = { DN2, DSS, EKS, FSA, GE1 }
k11 W2 k12



8.4 Löschen von Schlüsseln in einem Bayer-Baum der Ordnung M: (M ∈ ℕ, M ≥ 2)

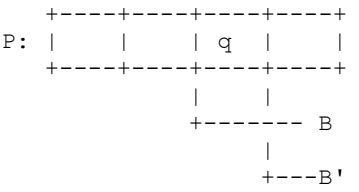
Geg.: Ein Bayer-Baum der Ordnung M, ein Schlüssel m_x, der im Bayer-Baum liegt.

Fallunterscheidung:

1. m_x kiegt in einem Blattknoten B.

   1.1 B hat a Schlüssel mit a > M: => m_x wird in B gelöscht.

   1.2 B hat genau M Schlüssel: 0> Bilde folgende Schlüsselmenge S(B, B'), die aus allen Schlüsseln von B ohne m_x, allen Schlüsseln des Nachbarblattknotens B' (hier eine Leseordnung nach links bzw. Nach rechts gemäß lexikographischer Anordnung des Baums auswählen) und dem zugehörigen mittleren Schlüssel des Parentknotens P von B, B' besteht.



$S(B, B') = \{ r_1, r_2, \ldots, r_{M-1}, q, s_1, \ldots, s_c \}$ mit $M \leq c \leq 2M$
$\{ r_1, r_2, \ldots, r_{M-1} \} \in B;\ \{ q, s_1, \ldots, s_c \} \in B'$

   1.2.a) $|S(B, B')| > 2M$ => S(B, B') hat genügend Schlüssel für die Balance-Operation: Wähle in S(B, B') den mittleren Schlüssel s_y: Setze s_y anstelle von q in den Parentknoten P. Ersetze B durch das Blatt B1, das alle Schlüssel $w \in S(B, B')$ mit $w < s_y$ enthält. Esetze B' durch das Blatt B2, das alle Schlüssel $w \in S(B, B')$ mit $w > s_y$ enthält.

   1.2.b) $|S(B, B')| = 2M$ => alle Schlüssel von S(B, B') werden in einen Blattknoten B1 eingefügt, B und B' werden gelöscht. => Prüfe den Parentknoten P, ob nach Entfernen des Schlüssels q noch genügend Schlüssel in P enthalten sind: JA => Fertig!
   Nein => REKURSION: Bilde die schlüsselmenge S(P, P'), wobei P' der Nachbarparentknoten gemäß Leseordnung (1.2 oben) ist. Der mittlere Schlüssel kommt aus dem Parentknoten Q von P, P'.
   Unterfälle:

   I. $|S(P, P')| > 2M$ => BALANCE (siehe 1.2.a) oben) => Fertig!

   II. $|S(P, P')| = 2M$ => Rekursion, die maximal zur Wurzel führt: Im Fall, wo Q genau einem Schlüssel enthält: => die alte Wurzel wird gelöscht. Der Knoten, der aus der Schlüsselmenge S(P, P') gebildet wird, ist die neue Wurzel.

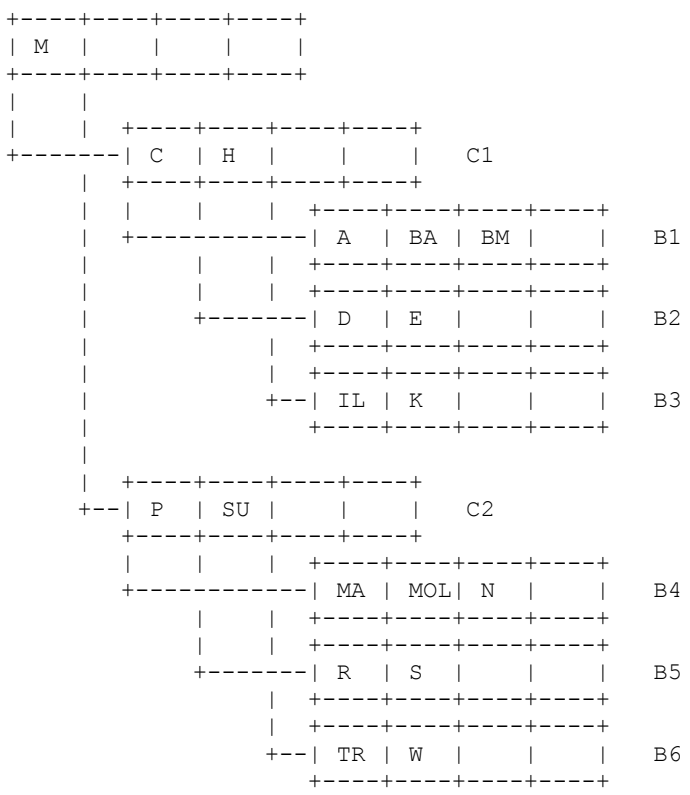
2. mx liegt in einem Nicht-Blattknoten C: => Schritte:

   a. Suche den zu mx nächsten benachbarten Schlüssel z in einem Blattknoten B (hierdurch wird eine Leseordnung nach links oder rechts festgelegt.)

   b. mx wird in C gelöscht und durch z ersetzt.

   c. Lösche z in B gemäß Fall 1.

BSP.: Gegeben ist der folgende Bayer-Baum der Ordnung M=2:



L.1: Lösche BM => Fall: 1.1 => B1 = | A | BA | | |

L.2: Lösche C => Fall: 2 (da C Schlüssel im Nichtblattknoten C1 ist)

=> Tausche Schlüssel C mit D aus B2 (Leseordnung = rechts)

=> Zustand: C1: | D | H | | |

=> Lösche D in B2 => Fall 1.2 => Schlüsselmenge:

S(B2, B3) = { E, H, IL, K }, da |S(B2, B3)| = 4 = 2M

=> Fall: 1.2.b)

=> B2, B3 werden gelöscht

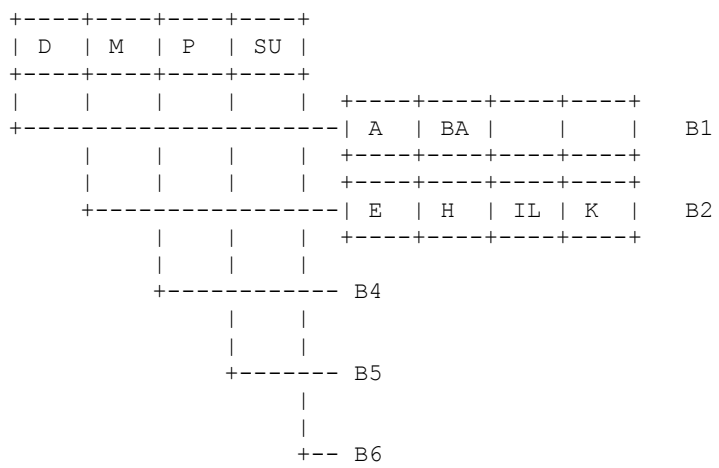
=> Neuer Blattknoten: B7 = | E | H | IL | K |

=> Rekursion (da S(C1) = M-1 (C1 = | D | | | |))

S(C1, C2) = { D, M, P, SU }

=> W1 = | D | M | P | SU | ist neue Wurzel.

Nach Abschluss der Löschoperation L.2 hat der Bayer-Baum folgende Gestallt:



8.5 Abschätzung des Suchaufwands in einem Bayer-Baum

Gegeben: Ein Bayer-Baum der Ordnung M, ein Benutzerschlüssel x.

Gesucht: Aufwand, um zu entscheiden, ob x in einem Knoten des Bayer-Baums enthalten ist oder nicht.

AUFWAND ~ Anzahl h der Knoten, die von der Wurzel bis zu einem Blatt zu durchlaufen sein.

Ansatz: Ein minimal gefüllter Bayer-Baum (pro Knoten ist nur die Mindestanzahl an Schlüsseln gespeichert).

Ebene	Anzahl der Knoten	Anzahl der Schlüssel
0	1	1
1	2	$2 \cdot M$
2	$2 \cdot (M+1)$	$2 \cdot M \cdot (M+1)$
3	$2 \cdot (M+1)^2$	$2 \cdot M \cdot (M+1)^2$
…	…	…
L	$2 \cdot (M+1)^{(L-1)}$	$2 \cdot M \cdot (M+1)^{(L-1)}$

Für die Summe SK aller Schlüssel gilt:

$$\begin{aligned} SK &= 1+2 \cdot M+2 \cdot M \cdot (M+1)+2 \cdot M \cdot (M+1)^2+\ldots+2 \cdot M \cdot (M+1)^{(L-1)} \\ &= 1+2 \cdot M \cdot \sum_{k=0}^{L-1} (M+1)^k = 1+2 \cdot M \cdot \left[\frac{(M+1)^{(L-1+1)}-1}{(M+1)-1} \right] = 1+2 \cdot M \cdot \left[\frac{(M+1)^L-1}{M} \right] \\ &= 1+2 \cdot \left[(M+1)^L-1 \right] = -1+2 \cdot (M+1)^L \end{aligned}$$

L: Anzahl der Knoten, die von der Wurzel bis zum Blatt zu durchlaufen sind.
=> Gleichung nach L auflösen

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

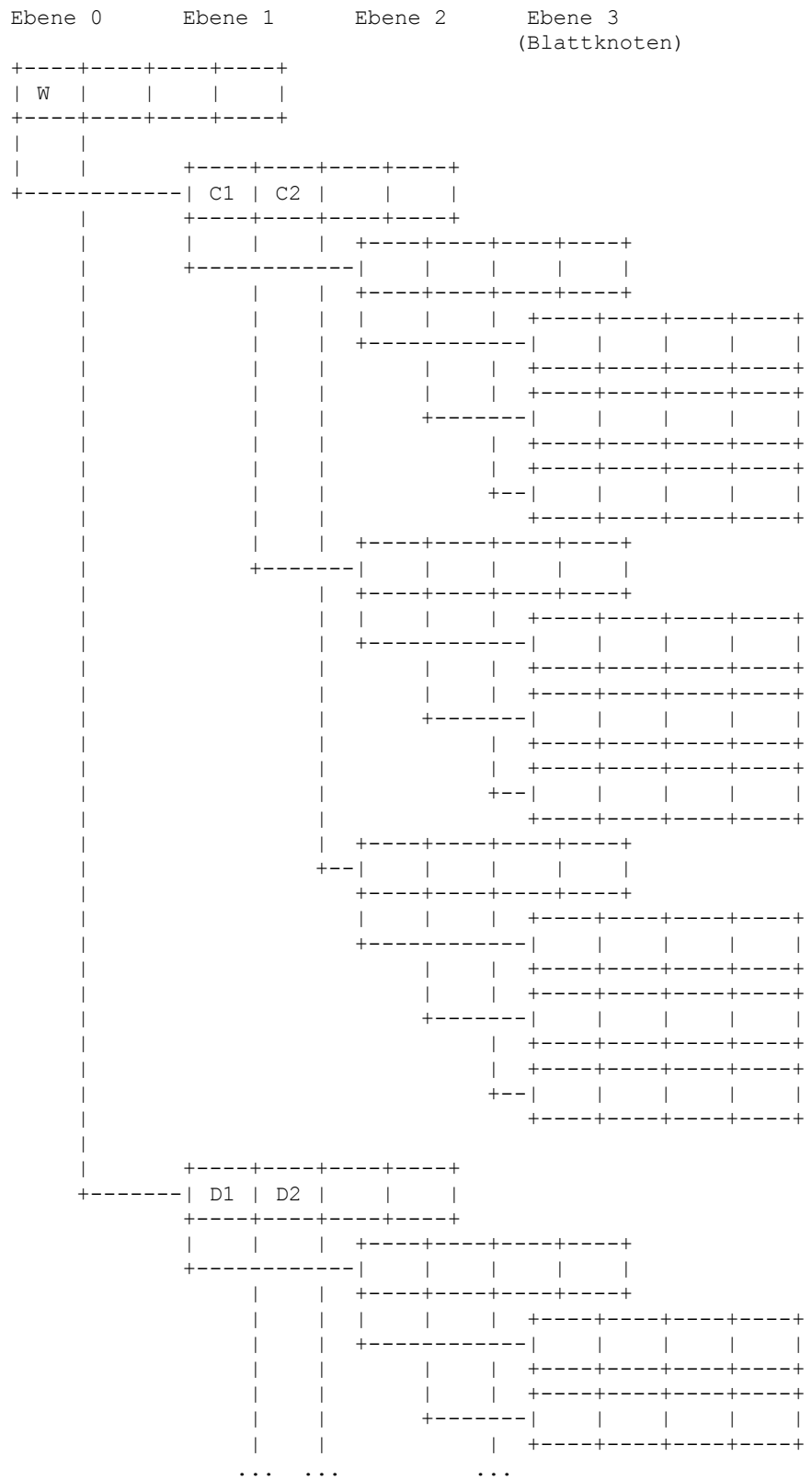
$$(M+1)^L = \frac{SK+1}{2} \quad | \log_{(M+1)}$$

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

Vergleich

SK	**Lineare Suche** $\sim \frac{SK}{2}$	**Binäre Suche** $\sim ld(SK)$	**Siche im Bayer-Baum** $\sim \log_{M+1}\left(\frac{SK+1}{2} \right)$
100	50	7	M=2 => 4
100 000	50 000	17	M=10 => 5
1 000 000 000	50 000 000	30	M=300 => 4

Aufwand(4) + Vergleiche (in der Ebene selber) ⇀


Ebene 0
Ebene 1
Ebene 2
Ebene 3
(Blattknoten)
W
C1
C2
D1
D2
...
...
...

1. Übung

a)

```
{
	"schema": "AUFTSCHEMA",
	"description": "Auftraege",
	"properties": {
		"AUFT": {
			"type": "object",
			"properties": {
				"KNAM": { "type": "string" },
				"AUFDAT": { "type": "string", "format": "date-time" },
				"POSLI": {
					"type": "array",
					"item": {
						"type": "object",
						"properties": {
							"ARTNR": "integer",
							"ARTBEZ": "string",
							"WERT": "number"
						}
					}
				},
				"AUFSUM": { "type": "integer" },
			}
		}
	}
}
```

b)

```
{
	"AUFT": {
		"KNAM": "string",
		"AUFDAT": "string",
		"POSLI": [{
			"ARTNR": "int",
			"ARTBEZ": "string",
			"WERT": "number"
		}],
		"AUFSUM": "int"
	}
}
```