Datenbanksysteme

Zusammenfassung

HTA Horw

Rainer Meier
Käserei
6288 Schongau
skybeam@skybeam.ch

Klasse: 4
2002 - 2006

2005-09-14

1. Inhaltsverzeichnis

2. Einleitung

Die strukturierte Verwaltung von Daten wird immer wichtiger. Aus diesem Grunde werden DBMS (Database Management Systems) immer wichtiger.

Ein DBMS besteht aus einer Menge von Daten und den zur Datenverarbeitung notwendigen Programmen.

Die folgenden Probleme treten immer häufiger auf und lassen sich durch den Einsatz eines DBMS verringern:

- Redundanz und Inkonsistenz: Die Daten werden häufig mehrfach abgelegt und sind zudem häufig inkonsistent. Das heisst die Daten sind an zwei abgelegten Orten nicht identisch.
- Beschränkte Zugriffsmöglichkeiten: Es ist schwierig verteilt abgelegte Daten in unterschiedlichen Datenformaten miteinander zu verknüpfen.
- Probleme des Mehrbenutzerbetriebes: Sind die Dateien auf unterschiedlichen Dateisystemen gespeichert ist es schwierig für alle Benutzer Zugriffsrechte zu vergeben. Ausserdem ist der gleichzeitige Zugriff auf die Daten meist nicht vorgesehen oder geschützt.
- Verlust von Daten: Backups können zwar helfen Daten zu rekonstruieren aber einen bekannt konsistenten Zustand wiederherzustellen ist meist extrem schwierig.
- Integritätsverletzung: Sind die Daten nicht von einem zentralen System verwaltet und geprüft, dann kann die Integrität der Daten nicht gewährleistet werden.
- Sicherheitsprobleme: Zugriffsrechte lassen sich praktisch nicht realisieren wenn die Daten beispielsweise verstreut in mehrere Tabellen/Dokumente auf eine Dateiablage erfasst werden.
- Hohe Entwicklungskosten: Da keine definierte Schnittstelle besteht müssen Programme für den Datenzugriff häufig komplett neu geschrieben werden.

2.1. Datenmodelle

Das Datenmodell beschreibt die Datenobjekte und die darauf anwendbaren Operationen. Dabei unterscheidet man grob in zwei Definitionssprachen:

- Data Definition Langugage (DDL): Definition der Struktur/Entitäten
- Data Manipulation Language (DML): Operationen auf die Daten selbst (anfragen/query, einfügen, modifizieren, löschen etc.)

2.2. Modelle des Entwurfs

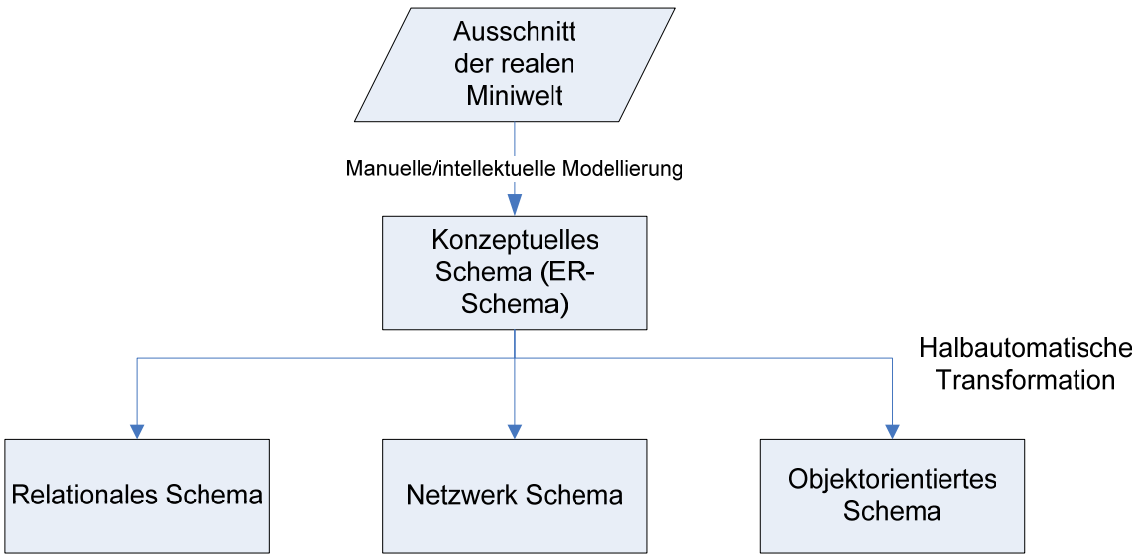

Abbildung 1 Übersicht der Datenmodellierung

Für den konzeptuellen Entwurf wird meist das ER-Modell (Entity-Relationship-Model) verwendet. Konzeptuelle Datenmodelle werden meist nur über die DDL (Data Definition Language) definiert da sie nur die Datenstruktur und nicht die Daten selber definieren.

2.3. Architektur

Die folgende Abbildung zeigt den Schematischen Aufbau eines DBMS:

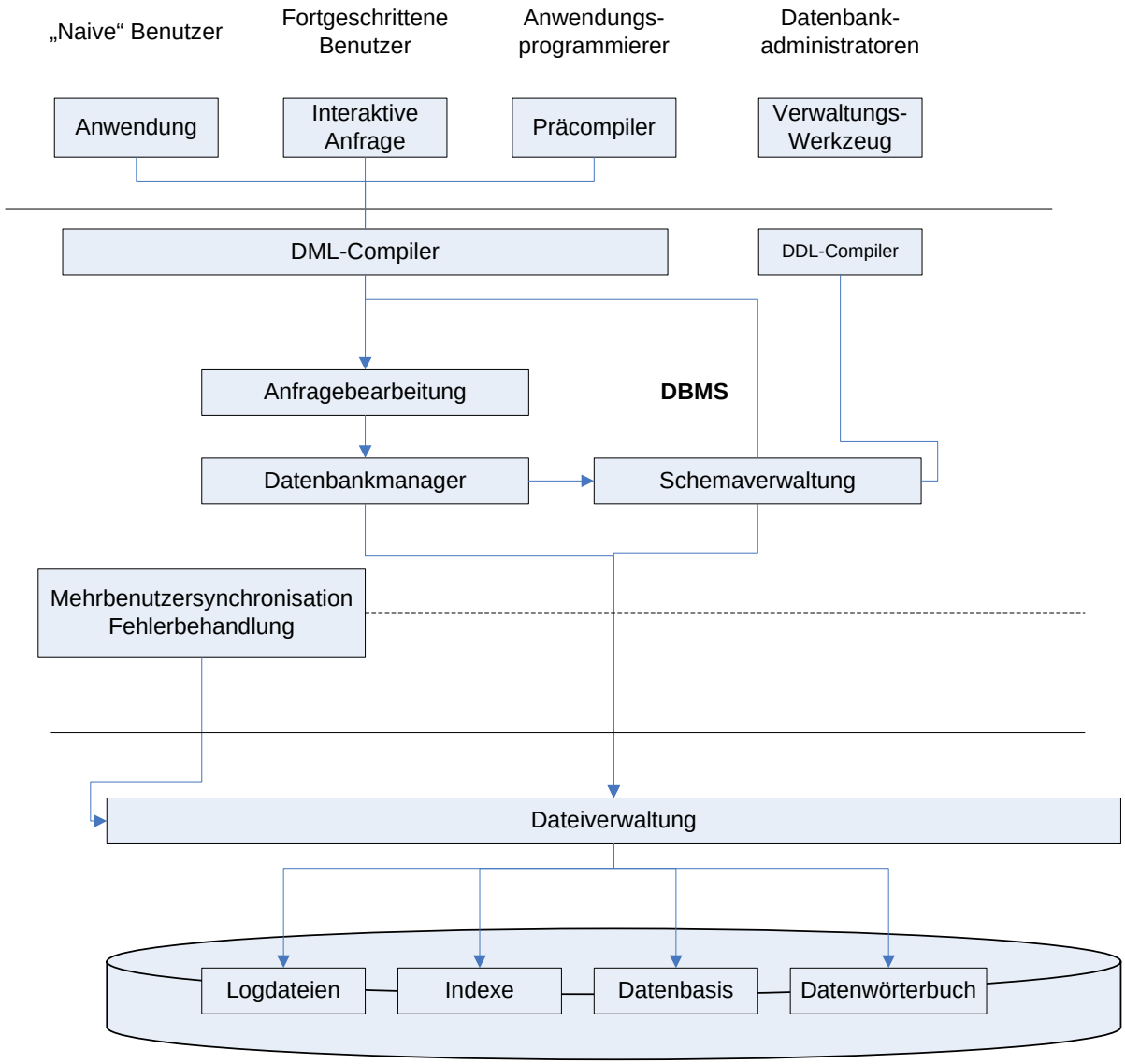

Abbildung 2 Architektur eines DBMS

3. Datenbankentwurf

Ein sauberer Datenbankentwurf ist sehr wichtig, da die Kosten für einen Fehler jeweils etwa um den Faktor 10 von der Anforderungsanalyse zur Entwurfsphase und schliesslich zur Realisierungsphase steigen.

Der Entwurf erfolgt dabei in Phasen:

- Anforderungs-Analyse: In dieser Phase werden die Anforderungsspezifikationen festgelegt.
- Konzeptueller Entwurf: In dieser Phase wird die Informationsstruktur festgelegt (ER-Schema)
- Implementations-Entwurf: Logische Datenbankstruktur wird festgelegt
- Phayischer Entwurf: Physische Datenbankstruktur

3.1. Entity-Relationship (ER) Modell

Das ER-Modell definiert wie der Name schon sagt Entitäten (Gegenstände) und ihre Beziehungen untereinander.

Ein ER-Modell besteht aus folgenden Elementen:

- Entitäten: Werden als Rechtecke dargestellt und entsprechen weitgehend den Gegenständen der Realen Welt.
- Attribute: Werden als ovale dargestellt und über ungerichtete Kanten mit den Entitäten verbunden. Hat ein Attribut eine Schlüsselfunktion so wird es unterstrichen.
- Beziehungen: Werden als Rauten mit entsprechender Beschriftung realisiert. Sie werden mit ungerichteten Kanten mit den Entitäten verbunden. Beziehungen mit der selben Entität nennt man rekursiv.

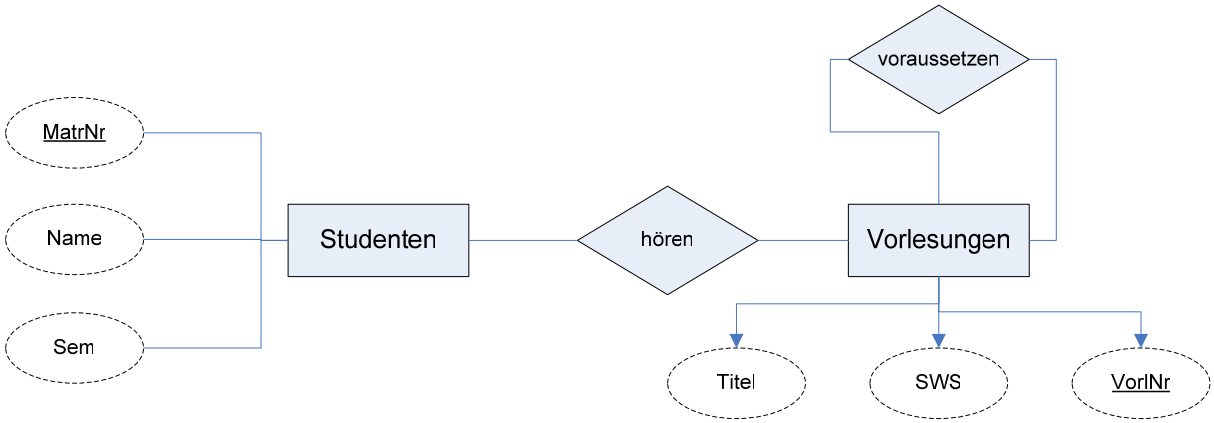

Abbildung 3 Konzeptuelles Schema

Schlüssel sind Attribute, die einen Datensatz der Entität eindeutig kennzeichnen. Häufig wird dazu ein künstliches Attribut geschaffen (z.B. MatrNr).

3.2. Beziehungen

Es gibt verschiedene Arten von Beziehungen, die im folgenden kurz erläutert werden:

3.2.1. 1:1 Beziehung

Diese Beziehung entsteht, wenn einem Element aus Entität 1 höchstens einem Element aus Entität 2 zugewiesen wird und umgekehrt einem Element aus Entität 2 höchstens einem Element aus Entität 1.

Die Folgende Abbildung zeigt eine 1:1 Beziehung, auch Bezeichnet als c:c (‚can' zu ‚can') oder 0..1:0..1.

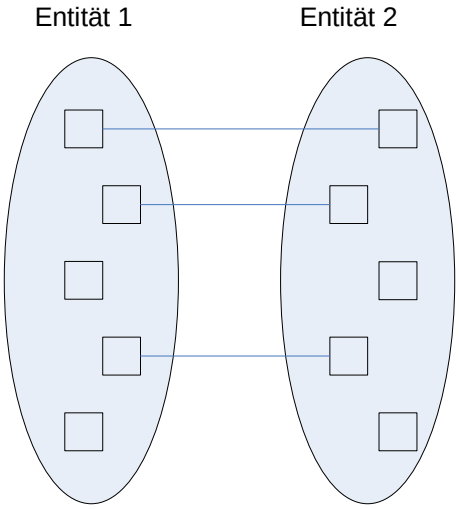

Abbildung 4 1:1 Beziehung (c:c)

3.2.2. 1:N Beziehung

Eine Beziehung bei der jedem Element aus Entität 1 beliebig viele Elemente aus Entität 2 zugeordnet werden könne aber jedes Element aus Entität 2 maximal mit einem Element aus Entität 2 in Verbindung stehen kann.

Die Folgende Abbildung zeigt eine 1:N Beziehung, auch Bezeichnet als c:mc (‚can' zu ‚multiple-can') oder 0..1:*.

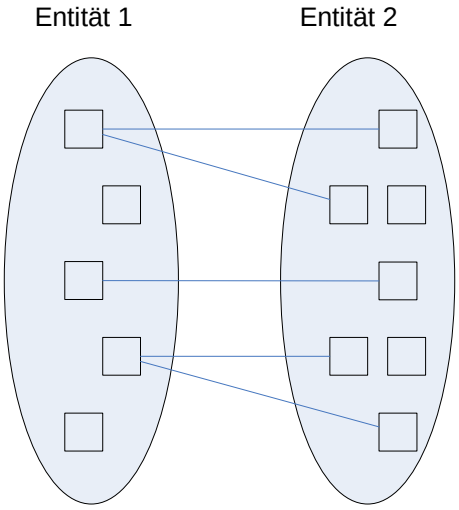

Abbildung 5 1:N Beziehung (c:mc)

3.2.3. N:M Beziehung

Eine Beziehung bei der keinerlei Restriktionen gelten müssen. Jedes Element aus Entität 1 kann mit beliebig vielen Elementen aus Entität 2 in Verbindung stehen. Umgekehrt kann jedes Element aus Entität 2 mit beliebig vielen Elementen aus Entität 1 in Verbindung stehen.

Die Folgende Abbildung zeigt eine N:M Beziehung, auch Bezeichnet als (mc:mc (‚multiple-can' zu ‚multiple-can') oder *:*.

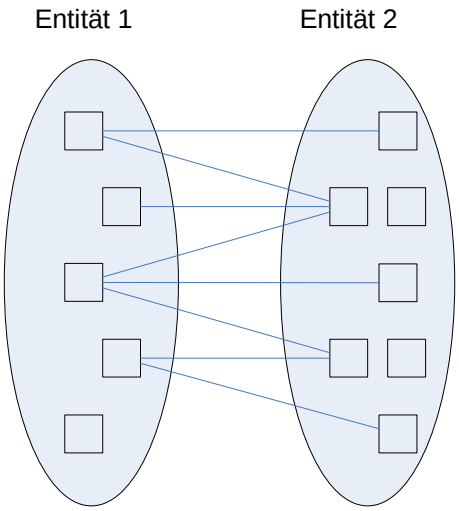

Abbildung 6 N:M Beziehung (mc:mc)

4. Relationales Modell

Im Relationalen Datenmodell gibt es nur flache Tabellen (Relationen). Die Verarbeitung geschieht Mengenorientiert. Über Referenzen kann von einem Datensatz zum nächsten navigiert werden.

Eine Relation ist die Teilmente des Kartesischen Produktes von n Domänen. Eine Domäne ist dabei eine Menge von n nicht notwendigerweise unterschiedlichen Wertebereichen.

4.1. Schema-Definition

In den meisten Systemen werden die Relationen als Tabellen dargestellt. Die Spalten stellen dabei di Attribute (auch Felder genannt) dar. Diese müssen innerhalb einer Relation eindeutig benannt werden.

Schema-Definitionen können nach folgendem Muster geschehen:

```
Telefonbuch:{[Name: string, Adresse: string, TelefonNr: integer]}
```

In eckigen Klammern steht dabei wie die Tupel aufgebaut sind (Attribut-Name und Typ). Die geschweiften Klammern geben dabei an, dass es sich bei der Relationsausprägung um eine Menge handelt.

4.2. Relationale Darstellung von Beziehungen

Zuerst wird aus dem ER-Diagramm aus jeder Beziehung eine Relation gebildet. Einige davon können später wieder zusammengefasst werden.

5. Relationale Algebra

Ausser der reinen Strukturbeschreibung braucht man auch eine Sprache mit der man Informationen aus der Datenbank extrahieren kann. Mithilfe der relationalen Algebra kann man Abfragen formulieren.

5.1. Selektion

Auswahl derjenigen Tupel, die das sogenannte Selektionsprädikat erfüllen.

Beispiel: $\sigma_{Semester>10}(Studenten)$

Selektiert alle Studenten, die in der Spalte Semester einen Wert > 10 eingetragen haben.

5.2. Projektion

Extrahieren von Spalten aus der Argumentrelation.

Beispiel: $\Pi_{Rang,Name}(\text{Professoren})$

Wählt die Spalten Rang und Name aus der Tabelle Professoren aus.

5.3. Vereinigung

Relationen mit gleichem Schema (gleiche Attributnamen und Attributtypen) können zu einer einzigen Relation zusammengefasst werden.

Beispiel: $\Pi_{PersNr,Name}(Assistenten) \cup \Pi_{PersNr,Name}(\text{Pr}ofessoren)$

Führt die beiden Tabellen Assistenten und Professoren zu einer einzigen neuen mit den Spalten PersNr und Name zusammen.

5.4. Mengendifferenz

Definiert die Menge aller Tupel, die in einer Relation vorkommen aber nicht in der zweiten.

Beispiel: $\Pi_{MatrNr}(Studenten) - \Pi_{MatrNr}(prüfen)$

Ergibt eine Tabelle, die alle Matrikelnummern der Studenten enthält, die nicht in der Tabelle Prüfen aufgeführt werden.

5.5. Kartesisches Produkt (Kreuzprodukt)

Enthält alle möglichen Paare von Tupeln aus zwei Relationen.

Beispiel: $R \times S$

Enthält n*m Tupel wobei n die Anzahl Tupel in R ist und m die Anzahl Tupel in S. Jedes Tupel aus R wird mit allen Tupeln aus S zusammengeführt.

5.6. Umbenennung von Relationen und Attributen

Manchmal wird eine Relation in einer Abfrage zweimal benötigt. Dazu muss mindestens eine umbenannt werden.

Beispiel: $\rho_{V1}(voraussetzen)$

Benennt die Tabelle voraussetzen in V1 um.

Praktisches Beispiel:

$$\Pi_{V1.Vorgänger}\left(\sigma_{V2.Nachfolger=5216 \wedge V1.Nachfolger=V2.Vorgänger}\left(\rho_{V1}(voraussetzen) \times \rho_{V2}(voraussetzen)\right)\right)$$

Bildet das Kreuzprodukt aus V1 und V2 (zweimal die Relation ‚voraussetzen') und selektiert dann die Zeilen in der der Nachfolger (aus V2) die ID 5216 hat und schränkt die Ausgabe auf die Vorgänger ein. In anderen Worten wird hier der die ID des Vorgängers des vor-Vorgängers der ID 5216 gesucht.

5.7. Der relationale Verbund (Join)

Über Joins werden Mehrere Relationen miteinander verbunden. Die Verbindung geschieht über eine Schlüsselspalte. Es gibt verschiedene Arten einen Join durchzuführen:

5.7.1. Natürlicher Join / inner Join

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⋈

C	D	E
C1	D1	E1
C3	D2	E2

=

A	B	C	D	E
A1	B1	C1	D1	E1

5.7.2. Linker äusserer Join

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⟕

C	D	E
C1	D1	E1
C3	D2	E2

=

A	B	C	D	E
A1	B1	C1	D1	E1
A2	B2	C2	-	-

5.7.3. Rechter äusserer Join

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⟖

C	D	E
C1	D1	E1
C3	D2	E2

=

A	B	C	D	E
A1	B1	C1	D1	E1
-	-	C3	D2	E2

5.7.4. Äusserer Join

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⟗

C	D	E
C1	D1	E1
C3	D2	E2

=

A	B	C	D	E
A1	B1	C1	D1	E1
A2	B2	C2	-	-
-	-	C3	D2	E

5.7.5. Semi-Join

Semi-Join der linken mit der rechten Relation:

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⋉

C	D	E
C1	D1	E1
C3	D2	E2

=

A	B	C
A1	B1	C1

Semi-Join der rechten mit der linken Relation:

Linke Tabelle | Operation | Rechte Tabelle | Resultat

A	B	C
A1	B1	C1
A2	B2	C2

⋊

C	D	E
C1	D1	E1
C3	D2	E2

=

C	D	E
C1	D1	E1

5.8. Baumdarstellung

Relational-Algebraische Ausdrücke lassen sich als Baum darstellen:

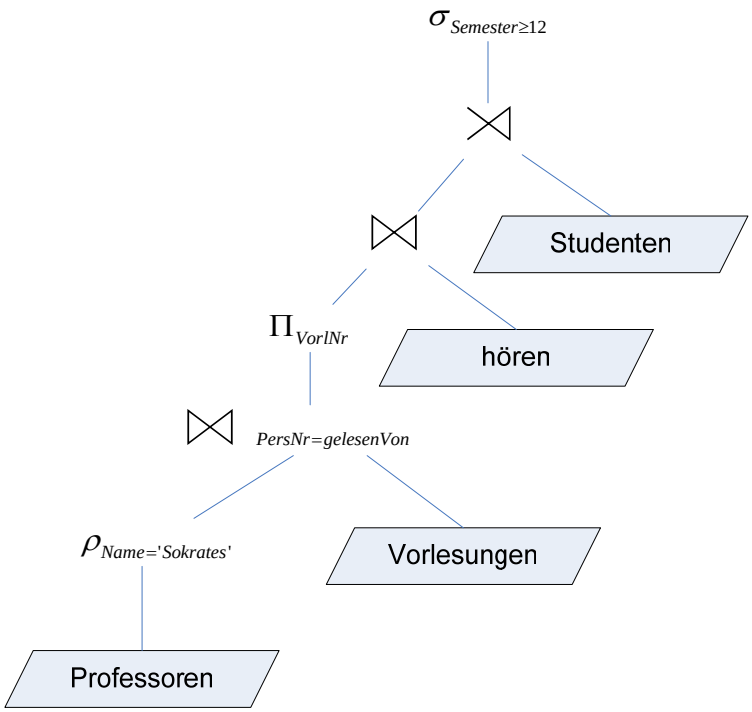

Abbildung 7 Baumdarstellung eines Algebraischen Ausdrucks

6. Relationale Anfragesprachen

Heute hat sich die Anfragesprache SQL weitgehend durchgesetzt. Sie setzt auf den Grundlagen der relationalen Algebra und der Theorie des Relationenkalkül auf.

Anfragesprachen wie SQL sind meist deklarativ. Der Benutzer gibt an welche Daten er haben möchte und nicht wie die Auswertung vorgenommen wird.

6.1. Datentypen

Jedes relationale Datenbanksystem stellt mindestens drei grundlegende Datentypen zur Verfügung:

- Zeichenketten: character, char varying/varchar
- Zahlen: int/integer, numberic, float
- Datum: date

6.2. Schemadefinition

Geschieht bei SQL mit dem ‚create table' Befehl:

```
create table Professoren (
    PersNr integer not null,
    Name   varchar(10) not null,
    Rang   character(2)
);
```

6.3. Schemaveränderung

Mit dem 'alter table' Befehl lassen sich Relationen manipulieren:

```
alter table Professoren add (Raum integer);
alter table Professoren modify (Name varchar(30));
SQL-92:
alter table Professoren add column Raum integer;
alter table Professoren alter column Name varchar(30);
```

6.4. Einfügen von Tupeln

Mit dem ‚insert into' Befehl lassen sich Daten einfügen:

```
insert into Professoren values (2136, 'Curie', 'C4', 36);
```

6.5. SQL-Abfragen

Mit dem 'select' Befehl lassen sich Anfragen durchführen.

```
select PerNr, Name
from Professoren
where Rang = ‚C4';
```

6.5.1. Eliminieren von Dupplikaten (distinct)

```
select distinct Rang
from Professoren;
```

6.5.2. Anfragen über mehrere Relationen

Anfragen können sich auch über mehrere Relationen erstrecken und diese direct miteinander verknüpfen.

```
select Name, Titel
from Professoren, Vorlesungen
where PersNr = gelesenVon and Titel = ‚Mäeutik';
```

Die obige Anfrage verknüpft die Tabellen Professoren und Vorlesungen über die Personalnummer, die in der Relation Professoren ‚PersNr' heisst und in der Relation Vorlesungen ‚gelesenVon'. Ausserdem werden nur Resultate in denen der Titel der Vorlesung ‚Mäeutik' ist ausgegeben.

Um die Spalten deutlicher zu kennzeichnen können sie auch über den Relationsnamen angesprochen werden:

```
select Name, Titel
from Professoren, Vorlesungen
where Professoren.PersNr = Vorlesungen.gelesenVon
    and Vorlesungen.Titel = ‚Mäeutik';
```

6.5.3. Aggregatfunktion und Gruppierung

Durch das Zusammenfassen auf Tupelmengen können Aggregatsfunktionen angewendet werden:

- avg: Durchschnittswert
- min: Der kleinste Wert
- max: Der grösste Wert
- sum: Summe aller Werte

Die Folgende Abfrage selektiert die durchschnittliche Anzahl Semester aller Studenten.

```
select avg(Semester)
from Studenten
```

Die Folgende Abfrage selektiert die durchschnittliche Semester-Wochen-Stunden für jeden lesenden Professor:

```
select gelesenVon, sum(SWS)
from Vorlesungen
group by gelesenVon;
```

Hier ist die ‚group by' Klausel nötig, da wir jeweils alle Einträge mit der selben gelesenVon Nummer zusammenfassen wollen (SUM).

Zusätzlich lassen sich die mit ‚group by' gebildeten Gruppen noch mit der ‚having' Klausel einschränken:

```
select gelesenVon, sum(SWS)
from Vorlesungen
group by gelesenVon
    having avg(SWS) >=3;
```

Möchten wir die Ausgabe um den Namen des Professors ergänzen brauchen wir zusätzlich die Professoren-Tabelle. In der where-Klausel werden die zutreffenden Tupel des Kreuzproduktes gefiltert:

```
select gelesenVon, Name, sum(SWS)
from Vorlesungen, Professoren
group by gelesenVon, Name
    having avg(SWS) >=3;
```

6.6. Geschachtelte Abfragen

Select-Anweisungen können auf vielfältige Weise verknüpft und geschachtelt werden:

```
select *
from prüfen
where Note = (select avg(Note) from prüfen);
```

Unteranfragen müssen immer geklammert werden!

Unteranfragen können auch Attribute der äusseren Anfrage verwenden:

```
select PersNr, Name, (select sum(SWS) as Lehrbelastung
                      from Vorlesungen
                      where gelesenVon = PersNr)
from Professoren;
```

```
select s.*
from Studenten s
where exists
    (select p.*
    From Professoren p
    Where p.GebDatum > s.GebDatum);
```

Der Exists Operator liefert true falls die Unteranfrage mindestens ein Ergebnistupel liefert.

```
select s.*
from Studenten s
where s.GebDatum <
    (select max(p.GebDatum)
    From Professoren p);
```

Temporäre Tabellennamen:

```
select tmp.MatrNr, tmp.Name, tmp.VorlAnzahl
from (select s.MatrNr, s.Name, count(*) as VorlAnzahl
    from Studenten s, hören h
    where s.MatrNr = h.MatrNr
    group by s.MatrNr, s.Name) tmp
where tmp.VorlAnzahl > 2;
```

6.6.1. Quantifizierte Anfragen

Der Exitenzquantor wird in SQL durch ‚exists' realisiert. Er liefert true, wenn die Unteranfrage keine leere Menge liefert.

```
select Name
from Professoren
where not exists (
    select *
    from Vorlesungen
    where gelesenVon = PersoNr
);
```

6.6.2. Nullwerte

Es gibt einen speziellen Wert mit dem Namen ‚null'. Dieser kann für alle Datentypen verwendet werden und bedeutet soviel wie ein unbekannter oder nicht definierter Wert.

Das Ergebnis von Abfragen die null-Werte liefern sind manchmal überraschend:

```
select count(*)
from Studenten
where Semester < 13 or Semester >= 13
```

Gibt es nun Studenten deren Semester-Attribut 'null' ist werden diese nicht mitgezählt.

Regeln für die Behandlung von ‚null'-Werten:

- Arithmetische ausdrücke: Ist ein Operator null wird das Ergebnis null. 1 + null = null, null * 10 = null.
- SQL kennt nicht nur true oder false sondern auch den Wert unknown. Bei Vergleichsoperationen wird ‚unknown' zurückgeliefert, wenn einer der Werte null ist.
- In einer where-Bedingung werden nur Tupel weitergereicht, für die die Bedingung true ist. Insbesondere werden Tupel, für die die Bedingung unknown auswertet, nicht ins Ergebnis aufgenommen.
- Bei einer Gruppierung wird null als ein eigenständiger Wert in eine eigene Gruppe eingeordnet.

6.6.3. Spezielle Sprachkonstrukte

Between kann benutzt werden um einen Wertebereich abzudecken:

```
select *
from Studenten
where Semester >= 1 and Semester <=4;
```

ist gleichbedeutend mit

```
select *
from Studenten
where Semester between 1 and 4;
```

Bereichsangaben können folgendermassen gemacht werden:

```
select * from Studenten where Semester in (1,2,3,4);
```

Platzhalter können mit dem like-Operator verwendet warden:

```
select * from Studenten where Name like ‚T%eoph_astos’;
```

‘%’ ist dabei der Platzhalter für beliebig viele Zeichen. ‚_’ steht als Platzhalter für ein einziges Zeichen.

6.6.4. Joins in SQL-92

Im from-Teil können folgende Schlüsselwörter verwendet werden:

- cross join: Kreuzprodukt
- natural join: Natürlicher Join
- join oder inner join: Theta-Join
- left, right oder full outer join: äusserer Join

```
select *
from R1 join R2 on R1.A = R2.B;
```

6.7. Sichten

Sichten sind quasi gespeicherte SQL-Abfragen, die sich wie Relationen direct ansprechen und weiter auswerten lassen.

Die Folgende Sicht bietet eine eingeschränkte „Ansicht“ der Relation ‚prüfen’.

```
Create view prüfenSicht as
    Select MatrNr, VorlNr, PersNr
    From prüfen;
```

7. Datenintegrität

Ein DBMS muss nicht nur die Daten ablegen und abfragen können sondern auch die Konsistenz der Daten sicherstellen. Dabei muss das System immer von einem Konsistenten Zustand in den nächsten konsistenten Zustand überführt werden.

Man unterscheidet statisch e und dynamische Bedingungen. Eine statische Bedingung muss von jedem Zustand der Datenbank erfüllt werden. Eine dynamische Bedingung wird bei Zustandsänderungen geprüft.

7.1. Referentielle Integrität

Die Attribute eines Schlüssels identifizieren ein Tupel eindeutig innerhalb einer Relation (Primärschlüssel). Verwendet man den Schlüssel einer Relation als Attribute einer anderen Relation, so spricht man von einem Fremdschlüssel.

In SQL gibt es für jeden der drei Schlüsselbegriffe eine Beschreibungsmöglichkeit:

- unique: Kennzeichnet einen Schlüsselkandidaten
- primary key: Kennzeichnet einen Primärschlüssel. Dies impliziert automatisch das ‚not null' Attribut.
- foreign key: Kennzeichnet einen Primärschlüssel. Fremdschlüssel können null sein falls nicht ‚not null' spezifiziert wurde. Ein ‚unique foreign key' modelliert eine 1:1-Beziehung.

Zusätzlich lässt sich das Verhalten bei Updates oder beim Löschen festlegen:

- cascade: Wird der Referenzierte Datensatz verändert (gelöscht/aktualisiert), so werden die Änderungen weitergegeben
- set null: Wird der refernzierte Datensatz verändert (gelöscht/aktualisiert), so wird der Fremdschlüssel auf den wert null gesetzt.
- restrict: Eine Änderung des referenzierten Datensatzes ist nicht erlaubt.

Überprüfung statischer Integritätsbedingungen:

Durch das Schlüsselwort ‚check' können Bedingungen geprüft werden:

- check Semester between 1 and 13...
- check Rang in (‚C2', ‚C3', ‚C4')...
- check ((S1 is not null and S2 is null and …) or (S1 is not null and S2 is not null and …))

Beispiele in SQL:

```
create table Studenten (
     MatrNr       integer primary key,
     Name         varchar(30) not null,
     Semester     integer check Semester between 1 and 13
);

create table Professoren (
     PersNr       integer primary key,
     Name         varchar(30) not null,
     Rang         character(2) check(Rang in ('C2', 'C3', 'C4')),
     Raum         integer unique);

create table Assistenten (
     PersNr       integer primary key,
     Name         varchar(30) not null,
     Fachgebiet   varchar(30),
     Boss         integer,
     Foreign key  (Boss) references Professoren on delete set null);
```

```
create table Vorlesungen (
     VorlNr       integer primary key,
     Titel        varchar(30),
     SWS          integer,
     gelesenVon   integer references Professoren on delte set null);

create table hören (
     MatrNr       integer references Studenten on delete cascade,
     VorlNr       integer references Vorlesungen on delete cascade,
     Primary key  (MatrNr, VorlNr));

create table voraussetzen (
     Vorgänger    integer references Vorlesungen on delete cascade,
     Nachfolger   integer references  Vorlesungen on delete cascade,
     Primary key  (Vorgänger, Nachfolger));

create table prüfen (
     MatrNr       integer references Studenten on delete cascade,
     VorlNr       integer references Vorlesungen,
     PersNr       integer references Professoren on delete set null,
     Note         numeric(2,1) check(Note between 0.7 and 5.0),
     primary key  (MatrNr, VorlNr));
```

7.2. Trigger

Ein Trigger ist eine Prozedur, die bei der Erfüllung einer bestimmten Bedingung vom DBMS gestartet wird.

```
create trigger keineDegradierung
before update on Professoren
for each row
when (oldl.Rang is not null)
begin
    if :old.Rang = ‘C3’ and :new.Rang = ‘C2’ then
          :new.Rang := ‘C3’;
    end if;
    if :old.Rang = ‘C4’ then
                :new.Rang := ‘C4’;
    end if
    if :new.Rang is null then
          new.Rang :=:old.Rang;
    end if;
end
```

8. Datenorganisation

Es gibt verschiedene Möglicheiten die Daten zu organisieren. Auf physikalischer Ebene spielt natürlich die Art und Geschwindigkeit des physikalischen Mediums (Memory, Festplatten, RAID etc.) eine Rolle. Auf logischer Ebene spielt die Organisation eine Rolle um die Daten möglichst schnell wieder finden zukönnen.

Dazu werden insbesondere zwei Arten von Bäumen eingesetzt:

8.1. B+-Baum

Die Blattebene enthält die Basisdaten in sortierter Form.

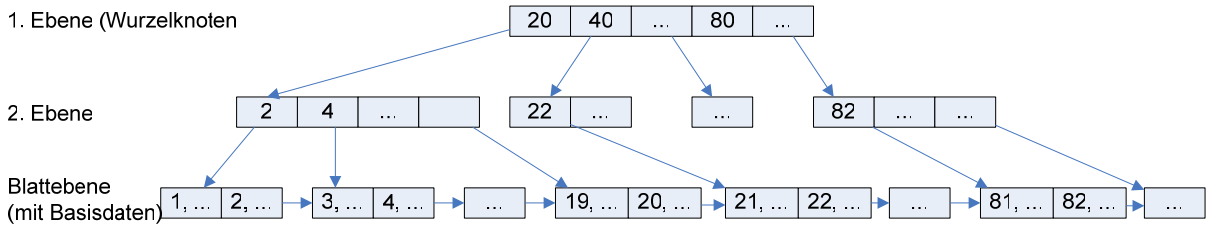

Abbildung 8 B+-Baum

Hier dargestellt passen zwei Datensätze auf einen Block. Es sind jeweils 10 Einträge auf den Blöcken/Knoten der mittlerene Ebenen dargestellt.

Pro Relation kann nur ein B+-Baum erstellt werden da ansonsten ja die Basisdaten mehrmals abgelegt würden.

8.2. B*-Baum

Die Basisdaten sind nicht im Baum gespeichert – auch nicht im Blattknoten!. Die Blattknoten beinhalten verweise auf die realen Datensätze:

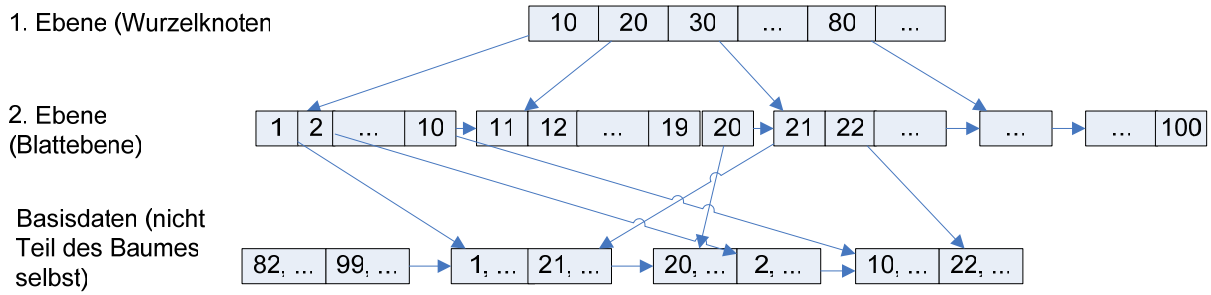

Abbildung 9 B*-Baum

Der B*-Baum wird meist für weitere Indexe auf weitere Attribute verwendet, da dabei die Daten nicht doppelt abgelegt werden.

8.3. Berechnung der Baumtiefe eines B+-Baumes

Die Baumtiefe eines B+-Baumes ist nicht so einfach zu berechnen da in den Blattknoten nicht gleichviele Elemente wie in den oberen Baumebenen gespeichert werden können.

Die allgemeine Formel lautet:

$$\left(\frac{B}{L_1}\cdot f\right)^{n-1}\cdot\left(\frac{B}{L_2}\cdot f\right)=X$$

Oder umgeformt:

$$\log_{\left(\frac{B}{L_1}\cdot f\right)}\left(\frac{X\cdot L_2}{B\cdot f}\right)=n-1$$

$$\frac{\log\left(\frac{X\cdot L_2}{B\cdot f}\right)}{\log\left(\frac{B}{L_1}\cdot f\right)}+1=n$$

Wobei

- B: Die Blocklänge in Byte ist
- L_1: Die Datensatzlänge (Verweis + Datensatz-ID)
- L_2: Die Datensatzlänge auf den Blattknoten (Datensatz-ID + Nutzdaten)
- f: den durchschnittlichen Füllgrad (Normalerweise rechnet man mit ¾ bzw. 0.75)
- X: Anzahl der Datensätze
- n: Die Baumhöhe (muss natürlich auf die nächste ganze Zahl aufgerundet werden)
- 0.75 ist die durchschnittliche Auslastung eines Blockes (3/4).

Beispiel:

1'000'000 Personensätze, 4kB Blockgrösse (4096 Byte), 4 Byte PersNr + 4 Byte Verweis = 8 Byte Datensatzlänge (L_1) und und angenommene 50 Byte Datensätze in den Blattknoteen:

- $\frac{\log(16276)}{\log(384)}+1=\frac{4.212}{2.584}+1=2.63$
- Dies muss natürlich auf die nächste ganze Zahl (also 3) aufgerundet werden

8.4. Berechnung der Baumtiefe eines B*-Baumes

Die Baumtiefe eines B*-Baumes kann folgendermassen berechnet werden:

$$nächsteGanzeZahl\left(Log_{\frac{B}{L}*f}(X)\right)$$

Wobei

- B: Die Blocklänge in Byte ist
- L: Die Datensatzlänge (Verweis + Daten)
- f: den durchschnittlichen Füllgrad (Normalerweise rechnet man mit ¾ bzw. 0.75)
- X: Anzahl der Datensätze
- 0.75 ist die durchschnittliche Auslastung eines Blockes (3/4).

Beispiel:

1'000'000 Personensätze, 4kB Blockgrösse (4096 Byte), 4 Byte PersNr + 4 Byte Verweis = 8 Byte Datensatzlänge:

- Berechne Einträge pro Block: 4096/8*0.75=384 Datensätze pro Block
- $Log_{384}(1'000'000)=2.33$
- Die Baumhöhe beträgt 3

9. Transaktionen

Transaktionen werden bei SQL durch ‚commit' abgeschlossen. Eigenschaften von Transaktionen (ACID Prinzip):

- Atomicy (Atomarität): Eine Trasaktion wird entweder vollständig in der Datenbasis festgeschrieben oder vollständig verworfen.
- Consistency: Eine Transaktion hinterlässt immer einen konsistenten Zustand.
- Isolation: Nebenläufige, parallel ausgeführte Transaktionen dürfen sich nicht gegenseitig beeinflussen.
- Durability (Dauerhaftigkeit): Eine abgeschlossene Transaktion bleibt dauerhaft in der Datenbank erhalten. Auch bei Systemfehlern (Hardware oder Systemsoftware).

10. Objektorientierte Datenmodellierung (ODMG)

Objektorientierte Datenbanken könne gegenüber Relationalen Datenbankmodellen intuitiver zu bedienen sein. Im folgenden wird das objektorientierte Prinzip anhand des ODMG-Standards beschrieben.

10.1. Attribute

```
class Professoren {
    attribute long PersNr;
    attribute string Name;
    attribute string Rang;
};
```

10.2. 1:1 Beziehung

```
class Professoren {
    Attribute long RaumNr;
    …
    Relationship Räume residiertIn inverse Räume::beherbergt;
};
class Räume {
    attribute long raumNr;
    attribute short Grösse;
    …
    Relationship Professoren beherbergt
                inverse Professoren:: residiertIn;
};
```

10.3. 1:N Beziehungen

```
class Professoren {
    …
    relationship set<Vorlesungen> liesst inverse Vorlesungen::gelesenVon;
};
class Vorlesungen {
    …
    Relationship Professoren gelesenVon inverse Professoren::liest;
};
```

10.4. N:M Beziehungen

```
class Studenten {
    …
    Relationship set<Vorlesungen> hört inverse Vorlesungen::Hörer;
};
class Vorlesungen {
    …
    Relationship set<Studenten> Hörer inverse Studenten::hört;
};
```

10.5. Ternäre Beziehungen

Bei Beziehungen mit 3 oder mehr Relationen ist es nötig eine Beziehungs-Klasse zu erstellen (ähnlich der Beziehungs-Klasse bei m:n Beziehungen in relationalen Systemen:

```
class Prüfungen {
    attribute struct Datum {
          short Tag; short Monat; short Jahr; } PrüfDatum;
    attribute float Note
    relationship Professor Prüfer inverse Professoren::hatGeprüft;
```

```
    relationship Studenten Prüfling inverse Studenten::wurdeGeprüft;
    relationship Vorlesungen Inhalt inverse Vorlesungen::wurdeAbgeprüft;
};
```

10.6. Extensionen und Schlüssel

Eine Extension ist die Menge aller Instanzen eines Objekttyps.

```
class Studenten (extent AlleStudenten key MatrNr) {
    attribute long MatrNr;
    attribute string Name;
    attribute short Semester;
    relationship set <Vorlesungen> hört inverse Vorlesungen::Hörer;
    relationship set <Prüfungen> wurdeGepr inverse Prüfungen::Prüfling;
};
```

10.7. Ableitung, Vererbung

```
class Angestellte (extent AlleAngestellten) {
    attribute long PersNr;
    attribute string Name;
    attribute date GebDatum;
    short Alter();
    long Gehalt();
};
class Assistenten extends Angestellte (extent AlleAssistenten) {
    attribute string Fachgebiet;
};
```

10.8. Abfragen per OQL

OQL ist syntaktisch an SQL angelehnt. Auch hier gibt es den select-from-where-Block.

Beispiel:

```
select p.Name
from p in AlleProfessoren
where p.Rang = „C4“;
```

Anfragen, die Tupel anstelle von einzelnen Objekten zurückgeben, enthalten den Tupelkonstruktor struct:

```
select struct(n: p.Name, r: p.Rang)
from p in AlleProfessoren
where p.Rang = „C4“;
```

10.8.1. Geschachtelte Anfragen

```
select struct( n: p.Namen, a: sum(select v.SWS from v in p.liest))
from p in AlleProfessoren
where avg(select v.SWS from v in p.liest) >2;
```

Dies entspricht weitgehend der Group by Klausel bei SQL.

10.8.2. Pfadausdrücke

Es besteht die Möglichkeit direkt zwischen den Objekten zu traversieren. Dadurch ist es häufig möglich ohne Join-Ausdrücke auszukommen:

```
select s.Name
from s in AlleStudenten, v in s.hört
where v.gelesenVon.Name = „Sokrates“;
```

10.8.3. Erzeugung von Objekten

Um anstelle von Literalen Objekte zu erhalten kann folgendes Konstrukt verwendet warden:

```
select p
from p in AlleProfessoren
where p.Name = “Sokrates”;
```

Hier wird das Objekt der Person mit dem namen “Sokrates” zurückgegeben und nicht lediglich der Name oder die MatrNr.

10.8.4. Operationsaufruf

Es können auch direkt Operationen auf Objekten aufgerufen werden:

```
select a.Name
from a in AlleAngestellte
where a.Gehalt() > 100000;
```

11. XML Datenmodellierung

Ein XML-Dokument besteht prinzipiell aus 3 Teilen:

- Präambel (optional): XML-Version und Encoding-Informationen
- Schema (optional): Die sogenannte Document Type Definition (DTD) oder XML Schema definiert die Struktur des Dokumentes
- Wurzelelement: Es gibt nur ein Wurzelelement, welches aber beliebig viele und beliebig tief geschachtelte Unterelemente beinhalten kann.

Beispiel:

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<!-- wie erwähnt ist die obige Präambel optional -->

<!-- Schema als DTD -->
<!DOCTYPE Buch [
   <!ELEMENT Buch (Titel, Autor*, Verlag)>
   <!ATTLIST Buch Jahr CDATA #REQUIRED>
   <!ELEMENT Titel (#PCDATA)>
   <!ELEMENT Autor (#PCDATA)>
   <!ELEMENT Verlag (#PCDATA)>
]>

<!-- Wurzelelement -->
<Buch Jahr="2004">
   <Titel>Datenbanksysteme: Eine Einführung</Titel>
   <Autor>Alfons Kemper</Autor>
   <Autor>Andre Eikler</Autor>
   <Verlag>Oldenourg Verlag</Verlag>
</Buch>
```

Das Schema ist optional, sofern es aber aufgeführt ist muss es auch eingehalten werden. Das Schema wird in der Praxis eher in einer eigenen DTD-Datei liegen.

Das Schema im Beispiel beschreibt, dass das Element Buch ein Attribut ‚Jahr' haben muss. Dies wird durch das REQUIRED Attribut ausgedrückt. Optionale Attribute können als IMPLIED deklariert werden. Des weiteren muss ein Unterelement Titel und Verlag angegeben werden. Das Unterelement Autor kann null bis beliebig viele male vorkommen. Der Stern kann auch durch ein Plus (+) ersetzt werden, dann muss mindestens ein Unterelement Autor angegeben werden.

Der Typ der Attribute in diesem Beispiel ist PCDATA (parsable character data).

11.1. Rekursive Schemata

In XML gibt es keine Restriktion für die Rekursion. Somit kann ein Element beliebig viele Unterelemente des selben Typs enthalten welches wiederum Elemente das selben Typs enthalten kann usw.

```
<?xml version="1.0" encoding='ISO-8859-1'?>

<!-- Schema als DTD -->
<!DOCTYPE Bauteil[
    <ELEMENT Bauteil (Beschreibung, Bauteil*)>
    <!ATTLIST Bauteil Preis CDATA #REQUIRED>
    <!ELEMENT Beschreibung (#PCDATA)>
]>

<-- Wurzelelement -->
<Bauteil Preis="350000">
    <Beschreibung>Maybach 620 Limousine</Beschreibung>
    <Bauteil Preis="5000">
          <Beschreibung>V12-Biturbo Motor mit 620 PS</Beschreibung>
          <Bauteil Preis="2000">
                <Beschreibung>Nockenwelle</Beschreibung>
          </Bauteil>
    </Bauteil>
    <Bauteil Preis="7000">
          <Beschreibung>Kühlschrank</Beschreibung>
    </Bauteil>
</Bauteil>
```

11.2. XML Namensräume

Werden in einem Dokument mehrere Schemadefinitionen verwendet, so muss sichergestellt werden, dass das Element eindeutig gekennzeichnet ist. Dies geschieht durch sogenannte Namensräume. Jedem Schema wird ein Namensraum zugewiesen über den das Element eindeutig identifiziert werden kann.

```
...
<Universität xmlns=http://www.db.uni-passau.de/Universität
    xmlns:lit="http://www.db.uni-passau.de/Literatur
    UnivName="Virtuelle Universität der Grossen Denker">
    <UniLeitung>
...
          <Vorlesung>
                <Titel>Informationssysteme</titel>
...
                <lit:Buch lit:Jahr="2003">
                <lit:Titel>Datenbanksysteme: Eine Einführung</lit:Titel>
                <lit:Autor>Andre Eikler</lit:Autor>
                <lit:Autor>Alfons Kemper</lit:Autor>
                <lig:Verlag>Oldenbourg Verlag</lig:Verlag>
                </lit:Buch>
          </Vorlesung
```

Das Element ‚Titel' kommt im Namensraum Universität und Literatur vor. Durch Zuweisung des Namensraumes ‚lit:' können die Elemente unterschieden werden. Natürlich könne man auch dem Universitäts-Namensraum einen Namen wie ‚uni:' zuweisen.

11.3. XML Schema

Die bisher gewählte DTD Definition hat einige Nachteile. So ist z.B. die Schemadefinition selber kein gültiges XML-Dokument und es existieren wenige Datentypen wie PCDATA. Ausserdem lassen sich keine komplexen Integritätsbedingungen definieren.

Hier hilft XML Schema Definition (XSD) nach. Ein XSD-Dokument ist selber eine XML-Datei mit dem Namensraum ‚xsd:', die vorher an die XMLSchema Definition gebunden wird:

```
<?xml version="1.0" encoding='ISO-8859-1'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.db.uni-passau.de/Universität">

   <xsd:element name="Universität" type="UniInfoTyp"/>

   <xsd:complexType name="UniInfoTyp">
     <xsd:sequence>
       <xsd:element name="UniLeitung">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name"Rektor" type="xsd:string"/>
             <xsd:element name="Kanzler" type="xsd:string"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
       <xsd:element name="Fakultäten">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="Fakultät" minOccurs="0"
                    maxOccurs="unbounded" type="FaultätenTyp"/>
           </xsd:sequence>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
     <xsd:attribute name="UnivName" type="xsd:string"/>
   </xsd:complexType>

   <xsd:complexType name="FakultätenTyp">
     <xsd:sequence>
       <xsd:element name="FakName" type="xsd:string"/>
       <xsd:element name="ProfessorIn" minOccurs="0"
maxOccurs="unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="Name" type="xsd:string"/>
             <xsd:element name="Rang" type="xsd:string"/>
             <xsd:element name="Raum" type="xsd:integer"/>
             <xsd:element name="Vorlesungen" minOccurs="0"
                    type="VorlInfo"/>
           </xsd:sequence>
           <xsd:attribute name="PersNr" type="xsd:ID"/>
         </xsd:complexType>
       </xsd:element>
     </xsd:sequence>
   </xsd:complexType>

   <xsd:complexType name="VorlInfo">
     <xsd:sequence>
       <xsd:element name="Vorlesung" minOccurs="1" maxOccurs="Unbounded">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="Titel" type="xsd:string"/>
             <xsd:element name="SWS" type="xsd:integer"/>
           </xsd:sequence>
           <xsd:attribute name="VorlNr" type="xsd:ID"/>
           <xsd:attribute name="Voraussetzungen" type="xsd:IDREFS"/>
         </xsd:complexType>
       </xsd:element>
```

```
      </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Hierbei ist zu beachten dass die ID-Felder innerhalb des gesamten Dokumentes eindeutig sein müssen (nicht nur innerhalb einer „Realation“ wie bei relationalen Datenbanken). Somit daruf auch die VorlNr nicht die selbe ID wie die PersNr erhalten.

Die Typen IDREF und IDREFS sind verweise und können beliebige ID’s enthalten. Hier gilt es zu beachten, dass die ID’s nicht typisiert werden können. Somit kann das Feld ‚Voraussetzungen des Typs VorlInfo auch eine ID aus dem Feld PersNr enthalten (welche ja innerhalb des gesamten Dokumentes eindeutig ist). IDREFS beinhaltet im Vergleich zu IDREF eine liste anstatt eines einzelnen Wertes.

12. XQuery

XQuery ist eine XML-Anfragesprache. Ähnlich wie bei objekt-orientierten oder objekt-relationalen Anfragesprachen kann man sich mit XQuery mittels Pfadausdrücken durch die Elemente hangeln.

Die dazu verwendete Sprache ist XPath.

12.1. XPath

XPath ist eine Sprache um Pfadausdrücke in XML darzustellen. Hier soll stellvertretend nur die verkürzte XPath Syntax beschrieben werden.

Es gibt verschiedene Abkürzungen, die bei XPath verwendet werden können. Die wichtigsten hier in Kürze:

- ‚.': Mit dem Punkt wird der aktuelle Refernznoten angegeben
- ‚..': Mit dem doppelten Punkt wird der Vaterknoten des aktuellen Referenzknotesn bezeichnet (ähnlich dem Dateisystem auf einer Shell).
- ‚/': Hiermit wird der Wurzelknoten bezeichnet. Wenn das Zeichen innerhalb eines Pfadausdrucks vorkommt, dann dient es als Trennzeichen zwischen den einzelnen Schritten des Pfades (ähnlich dem Dateisystem auf einer Shell).
- ‚@': Mit diesem Operator werden die Attribute des aktuellen Knotens bezeichnet. Dadurch kann syntaktisch zwischen Knoten und Element unterschieden werden.
- ‚//': Hiermit werden alle Nachfahren des aktuellen Knotens einschliesslich des Refernznotens selbst bezeichnet (entspricht etwa dem Unix Pfadausdruck */<name>). Dazwischen können beliebige Pfadebenen liegen.
- ‚[n]': Wenn das Prädikat nur aus einem Zahlenwert ‚n' besteht dient es dazu das n-te Element auszuwählen.

Hier einige Beispiele:

```
doc(„Uni.xml“)/Universität/Fakultäten/
    Fakultät[FakName=“Physik“]//Vorlesung

doc(„Uni.xml“)/Universität/Fakultäten/
    Fakultät[position()=2]//Vorlesung
{gleichbeduetend mit}
doc(„Uni.xml“)/Universität/Fakultäten/
    Fakultät[2]//Vorlesung
```

12.2. Anfragesyntax von XQuery

Anfragen werden durch sogenannte FLOWR (for..let..where..order by...return) Ausdrücke spezifiziert. Ein kurzes Beispiel:

```
<Vorlesungsverzeichnis>
    {for $v in doc(„Uni.xml“)//Vorlesung
    Return $v
    }
</Vorlesungsverzeichnis
```

Wie man sehen kann ist die let- und where-Klausel optional.

Hier eine Anfrage mit where-Klausel in zwei verschiedenen Formen:

```
<Vorlesungsverzeichnis>
    {for $v in doc(„UNI.xml“)//Vorlesung[SWS=4]
    Return $v
    }
</Vorlesungsverzeichnis>
... oder
<Vorlesungsverzeichnis>
```

```
    {for $v in doc(„UNI.xml“)//Vorlesung
    Where $v/SWS = 4
    Return $v
    }
</Vorlesungsverzeichnis>
```

12.3. Geschachtelte Anfragen in XQuery

Beispiel einer geschachtelten Anfrage:

```
<VorlesungsVerzeichnisNachFakultät>
    {for $f in doc(„Uni.xml“)/Universität/Fakultäten/Fakultät
    Return
    <Fakultät>
          <FakultätsName>{$f/FakName/text()}</FakultätsName>
          {for $v in $f/ProfessorIn/Vorlesungen/Vorlesung
          return $v
          }
    </Fakultät>
</VorlesungsVerzeichnisNachFakultät>
```

12.4. Joins in XQuery

Self-Join mit der ‚contains'-Funktion:

```
<MäeutikVoraussetzungen>
    {for $m in doc(„Uni.xml“)//Vorlesung[Titel=“Mäeutik“],
          $v in doc(„Uni.xml“)//Vorlesung
    where contains($m(@Voraussetzungen,$v/@VorlNur)
    return $v/Titel
    }
</MäeutikVoraussetzungen>
```

Join:

```
<ProfessorenStammbaum>
    {for $p in doc(“Uni.xml”)//ProfessorIn,
          $k in doc(“Stammbaum.xml”)//Person,
          $km in doc(„Stammbaum.xml“)//Person,
          $kv in doc(„Stammbaum.xml“)//Person
    where $p/Name = $k/Name and $km/@id = $k/@Mutter and
          $kv/@id = $k/@Vater
    Return
    <ProfMutterVater>
          <ProfName>{$p/Name/text()}</ProfName>
          <MutterName>{$km/Name/text()}</MutterName
          <VaterName>{$kv/Name/text()}</VaterName>
    </ProfMutterVater>
</ProfessorenStammbaum>
```

12.4.1. Join Prädikat im Pfadausdruck

```
<GefährdetePersonen>
    {for $p in doc(„Stammbaum.xml“)//Personen[Name = „Kain“],
          $g in doc(„Stammbaum.xml“)//Person[
                      @Vater = $p/@Vater and @Mutter = $p/@Mutter]
    return $g/Name
    }
<GefährdetePersonen>
```

13. Abbildungsverzeichnis

14. Index