Vorlesung

Multimedia-Datenbanksysteme

Sommersemester 2014

Wolfgang Kowarschick

Hochschule Augsburg

Inhalt

Vorwort

Die Vorlesung Multimedia-Datenbanksysteme ist ein zentraler Bestandteil des Studiengangs „Interaktive Systeme“. Heutzutage ist es unmöglich, ein größeres Web-Projekt ohne Datenbank zu realisieren. In dieser Vorlesung sollen die wichtigsten Grundlagen von Datenbanksystemen für den Einsatz in Medienanwendungen vermittelt werden.

Der Inhalt dieser Vorlesung basiert u. a. auf dem Vorlesungsskript des Datenbank-Pioniers Prof. Rudolf Bayer, meinem Lehrer und Doktorvater. Er hat mir dankenswerterweise seine handschriftlichen Aufzeichnungen zur Verfügung gestellt: Bayer [1996].

Ich werde immer wieder gefragt, welche guten Bücher zum Thema Datenbanken ich empfehlen kann. Diese Frage kann ich direkt nicht beantworten, da ich meist Fachliteratur verwende, die weit über den vorgestellten Vorlesungsstoff hinausgeht. Im Literaturverzeichnis sind jedoch einige,wie z. B. Standardwerke aufgeführt:

- Ramakrishnan und Gehrke [2002] (teuer, Anfang 2010: Taschenbuch)
- Elmasri und Navathe [2002] (sehr umfangreich)
- Ullman [1988, 1989] (*das* Standardwerk, *sehr* tiefgehend)

Kapitel 1

Einführung

1.1 Modelle, Fakten, Datenmodelle und Daten

(vgl. Bayer [1996])

Der Mensch kann die Welt/Realität nicht direkt erfassen. Er bildet letztlich immer (zumindest auf geistiger Ebene) *Modelle* der Realität. Neue Erkenntnisse kann er nur gewinnen, wenn er neue *Fakten* in seine Modellwelt einordnen oder die Modellwelt geeignet abändern kann.

Realitätsausschnitte	**Modelle**	**Fakten**
Musik	Notensystem	Eine kleine Nachtmusik
Thermodynamik	Differentialgleichungen	$f' = f$
Chemie	Formeln	H_2O, CO_2
Meereswellen	Chaostheorie	Simulationen
Städte, Länder etc.	Landkarten	Augsburger Stadtplan

Mathematik und Informatik bilden ebenfalls abstrakte Modelle für die „Realität“, allerdings wesentlich formaler als dies mit natürlichen Sprachen möglich ist.

Fakten werden in der Informatik normalerweise in Form von so genannten *Daten*, d. h. als *Zeichenketten* modelliert. Multimedia-Daten (Audio, Video, Bilder etc.) werden von Datenbanksystemen meist lediglich als so genannte BLOBs (Binary Large OBjects), d. h. als lange Folge von Bits dargestellt, deren Bedeutung dem jeweiligen DB-System unbekannt sind.

Für gegebene Realitätsausschnitte können, um die Bedeutung von Fakten festzulegen, so genannte *Datenmodelle* entwickelt werden (siehe beispielsweise die bereits bekannte „objektorientierte Modellierung“).

Die *Abbildung eines Realitätsausschnittes* auf Fakten eines Modells oder – spezieller – auf Daten eines Datenmodells stellt immer einen so genannten *Abstraktionsschritt* dar. Abstraktion ermöglicht einerseits ein besseres Verständnis, bedeutet andererseits aber einen *Präzisionsverlust*, d. h., (hoffentlich) unwesentliche Teile der Realität werden vernachlässigt.[1]

[1] Der Präzisionsverlust ist manchmal allerdings erheblich. Man denke nur an die zahllosen Interpretationsmöglichkeiten eines einzelnen Musikstückes wie der kleinen Nachtmusik.

Datenmodelle sind normalerweise nicht sehr umfangreich, die zugehörigen Datenbestände können dagegen häufig beliebig groß werden. Die Aufgabe von *Datenbank-Management-Systemen* (*DBMS*) ist es, große, langlebige Datenbestände effizient zu verwalten. Unter einem *Datenbank-System* (*DBS*) verstehe ich ein Datenbank-Management-System (die eigentliche DB-Software) zusammen mit den zugehörigen Datenmodellen, Datenbeständen und weitere Komponenten (wie grafische Benutzerschnittstelle, Report Generator, WWW-Schnittstelle etc.). Mit Datenbanksystemen werden wir uns in dieser Vorlesung befassen.

Definitionen

Hier werden die oben eingeführten Begriffe noch einmal kurz zusammengefasst. Im Kontext des Relationenmodells (Kapitel 3) werden diese Begriffe genauer definiert.

Datenmodell Abbildung der Realität auf ein Modell, das von einem Rechner verarbeitet werden kann.

Daten Repräsentation von Fakten gemäß einem zuvor definierten Datenmodell.

Datenbank (DB) Ein Datenmodell zusammen mit einer Menge von zugehörigen Daten.

Datenbank-Management-System (DBMS)
Ein Programm-System, das zur Verwaltung von Datenbanken eingesetzt werden kann.

Datenbank-System (DBS)
Ein Datenbank-Management-System zusammen mit einer oder mehrerer konkreten Datenbanken.

Aufgaben eines DBMS

1. Daten dauerhaft speichern (Dauerhaftigkeit/Persistenz)
2. Suche nach bestimmten Daten ermöglichen (Retrieval, Select)
3. Modifikationen von Daten ermöglichen (Insert, Delete, Update)
4. Datenmodell verwalten (Create Table, Create View, Create Index, …)
5. Wartezeiten vermeiden (Effizienz: Indexe, Optimierung)
6. Mehrbenutzerfähigkeit (User + Programme)
   (Parallelität, Transaktionen, Datenschutz: Benutzerverwaltung, Rechte)
7. Fehler im Datenbestand vermeiden
   (Datenkonsistenz: Integrität, Transaktionen)

8. Konsistenten Zustand im Fehlerfall wieder herstellen
   (Datensicherheit: Recovery)

Vergleich zu File-System

1. O.k.: Daten werden in Dateien gespeichert
2. Navigierende Suche (Verzeichnisbaum), primitiv, teilweise Erweiterungen (z. B. Google-Desktop[2])
3. O.k.
4. Datenmodelle gibt es nicht
5. B-Baum-basierte File-Systeme
6. ja, aber wenn zwei Benutzer an derselben Datei arbeiten, kommt es i. Allg. zu Datenverlust
7. keine oder nur wenig Unterstützung
8. moderne File-Systeme bieten Journaling, Backupsysteme gibt es als externe Tools

Beispiele für Datenbank-Anwendungen

- Banken (die Datenbankpioniere)
- Deutsche Bahn AG[3]
- Bibliotheken (Library of Congress[4] (LoC), viele multimediale Daten)
- Meteorologisch Institute
- NASA/ESA (Satelliten, Marssonde etc.)
- Versandhandel (Kataloge)
- Suchmaschinen (Google, Bing, ...)
- Medienplttformen (Flickr, Youtube, ...)
- etc.

Datenbestände der DB AG

- Fahrpläne
- Tarife
- Streckennetz
- Wagenpark

[2] desktop.google.com
[3] www.bahn.de
[4] www.loc.gov

- Stromnetz
- Personal
- etc.

Frage: Wie sieht beispielsweise ein Datenmodell für Fahrpläne aus?

kurze Antwort: Fahrpläne werden normalerweise als Tabellen realisiert. Das Datenmodell legt die Strukturen dieser Tabellen sowie die erlaubten Inhalte der einzelnen Zeilen und Spalten fest.

Datenbestände von Bibliotheken

- Texte, Bilder, Tonträger, Videos
  (z. B. LoC: Originalwalzen für Edison-Phonographen, Originalfilme von Laurel/Hardy etc.)
- Kataloge
- Benutzerverwaltung
- Bestellwesen
- etc.

Geo-Informationssysteme

Ein Spezialgebiet der DB-Welt stellen die so genannten Geo-Informationssysteme (GIS) dar. In derartigen Systemen geht es darum, 2- und 3-dimensionale Landkarten zu verwalten. Die Kostenrelation eines typischen GIS-Systems sieht nach Aussagen von Mitarbeitern der Fakultät für Geowissenschaft der Ludwig-Maximilians-Universität München ungefähr wie folgt aus:

Kosten Hardware: Faktor 1

Kosten Software: Faktor 10 (DBMS)

Kosten Datenerfassung: Faktor 100 (Landvermessung etc.)

1.2 Historie

Die historische Entwicklung von Datenbank-Management-Systemen:

1. Hierarchische DBMS (IMS von IBM)
2. Netzwerk-DBMS (CODASYL; UDS von Siemens)
3. Relationale DBMS (System R von IBM)

4. Objektorientierte DBMSe (überspitzt formuliert: Rückschritt zu 2.)
   (sehr effizient für schema-konforme Anfragen, wenig effizient für nicht-schema-konforme Anfragen – genauso wie Netzwerk-DBS).
5. Objektrelationale DBMS sowie DBMS für Spezialaufgaben:
   Suchmaschinen (Google, ...), Data-Warehousing/-Mining, XML-DBMS, Streaming Server

1.3 Datenbank-Modelle

Datenbank-Modelle, d. h. Datenmodelle für Datenbanken sollen folgende Eigenschaften haben:

1. verständlich, insbesondere nicht zu umfangreich
2. ausreichend präzise und ausreichend vollständig für bestimmte Zwecke
3. effizient realisierbar, d. h., eine gewünschte Auskunft kann bzgl. eines zugehörigen Datenbestandes „schnell“ erteilt werden
4. leicht modifizierbar, um schnell an neue Gegebenheiten angepasst werden zu können, ohne dass vorhandene Datenbestände wertlos werden

Lösungen:

ad 1. einheitliche Formalisierungsmethode mit *wenigen* formalen Elementen

ad 2. möglichst mächtige Formalisierungsmethode

ad 3. kleiner, aber mächtiger Satz von *effizient* zu realisierenden Elementaroperationen

Im Jahr 1970 kam Codd auf die Idee, (mathematische) *Relationen* (d. h. ungeordnete, duplikatfreie *Tabellen*) und *relationale Operationen* (Vereinigung, Durchschnitt, Differenz, kartesisches Produkt (Join)) zur Verwaltung von Massendaten einzusetzen (Codd [1970]).

Es hat sich im Lauf von bald 30 Jahren gezeigt, dass dieser Formalismus, so einfach er auch ist, sehr wohl geeignet ist, hochkomplexe Datenmodelle zu erstellen und die zugehörigen Daten effizient zu verwalten.

Das *Relationenkalkül* hat allerdings auch einige *Schwächen*. Und so verwundert es nicht, dass im Laufe der letzten 30 Jahre auch andere Formalismen entwickelt wurden, allen voran die *objektorientierten* und die *XML-Systeme*. Verdrängen werden sie die relationalen Systeme allerdings nicht so schnell – es zeichnet sich vielmehr ab, dass in Zukunft Mischungen aus diesen Welten, die so genannten *objektrelationalen Systeme* mit XML-Unterstützung das Rennen machen werden.

Definitionen

Das ***kartesische Produkt*** mehrerer Mengen ist die Menge aller Tupel, die sich aus diesen Mengen bilden lassen.

Formaler: Ein ***kartesisches Produkt*** $D_1 \times \ldots \times D_n$ ist die Menge aller Tupel $(v_1, \ldots, v_n)$, wobei die i-te Komponente v_i eines jeden Tupels in der Menge (Domäne) D_i enthalten sein muss.

Eine ***Relation*** ist eine Teilmenge eines kartesischen Produktes. Eine Relation lässt sich sehr gut als Tabelle mit je einem Tupel pro Zeile darstellen. Daher bezeichnet man Relationen in Zusammenhang mit Relationalen DBS (RDBS) häufig auch als Tabellen.

In RDBS werden i. Allg. Attribute in einen Tupel durch Namen $a_1, \ldots, a_n$ und nicht durch die Position $1, \ldots, n$ identifiziert. Derartige Tupel schreibe ich als $(a_1 : D_1, \ldots, a_n : D_n)$.

Beispiele (für kartesische Produkte)

$\{a, b\} \times \{1, 2\} \times \{x, y\} =$
$\{(a, 1, x), (a, 1, y), (a, 2, x), (a, 2, y), (b, 1, x), (b, 1, y), (b, 2, x), (b, 2, y)\}$

$\mathbb{N} \times \mathbb{N} = \{(1, 1), (1, 2), \ldots, (2, 1), (2, 2), \ldots, (3, 1), (3, 2), \ldots, \ldots, \}$

$\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ = zweidimensionaler Vektorraum

$\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R}^3$ = dreidimensionaler Vektorraum

Beispiel (für eine Relation)

$\{(1, 1), (2, 4), (3, 9), (4, 16)\} \subset \mathbb{N} \times \mathbb{N}$
bzw.
$\{(x : 1, y : 1), (x : 2, y : 4), (x : 3, y : 9), (x : 4, y : 16)\}, x \in \mathbb{N}, y \in \mathbb{N}$

```
                x y
1 1             1 1
2 4     bzw.    2 4
3 9             3 9
4 16            4 16
```

1.4 Extreme Ansätze für DBMS

1.4.1 Natürlichsprachliche DBMS

Die an leichtesten handzuhabenden DBMS wären diejenigen, die direkt natürlichsprachliche Beschreibungen (= Datenmodelle) und Fakten verarbeiten könnten (siehe Bordcomputer von Raumschiff Enterprise).

Darstellung:	Text + Bilder, Töne ...
Auskünfte:	Information Retrieval (teilweise in Telefonauskunft-Systemen realisiert)
Änderungen:	Text-/Bild-/Audio-/... Editoren
Vorteile:	einfach handzuhaben, mächtig
Nachteile:	derzeit leider nur ansatzweise realisierbar, da *volles* Sprachverständnis implementiert werden müsste

1.4.2 3GL-DBMS

Wenn natürlichsprachliche Ansätze nicht funktionieren, warum dann nicht einfach vorhandene algorithmische Sprachen (C++, Java, Pascal ...), d. h. Sprachen der dritten Generation (3GL)[5] einsetzen?

Darstellung:	direkte Nachbildung von z. B. Papierformularen durch Listen, Bäume, Geflechte etc.
Auskünfte:	Spezialprogramme
Änderungen:	Spezialprogramme
Vorteile(?):	gewohnte Repräsentation und Verarbeitung (Gegenbeispiel Bauernhof: mechanisches Pferd – gewohntes Verhalten –, besser ist jedoch ein Traktor als Ersatz für Pferde!)
Nachteile:	aufwändig, starr, Mehrbenutzerbetrieb schwierig
Beispiele:	frühe Flug- und Banksysteme

1.5 4GL-DBMS

Für heutige Datenbanken wurden Sprachen und Formalismen oberhalb von 3GL-Sprachen aber natürlich unterhalb natürlichsprachlicher Systeme entwickelt.

[5] 1GL = Hardware-Codierung, 2GL = Assemblersprachen

Relationale DBMS

Darstellung: Relationen (Tabellen) + Integritätsbedingungen (Invarianten), um fehlerhafte Datenbankinhalte zu vermeiden.

Auskünfte: Deklarative (nicht-algorithmische), d. h. beschreibende Sprache: relationale Algebra (SQL: `SELECT ... FROM ... WHERE ...`).

Änderungen: Satz von Modifikationsoperationen (SQL: `INSERT`, `DELETE`, `UPDATE`) unter Beachtung von Integritätsbedingungen.

Vorteile: Einfach handzuhaben, effizient (mittels Hilfsdatenstrukturen wie B-Bäumen (Bayer und McCreight [1970]), R-Bäumen (Guttman [1984]) etc.), weit verbreitet.

Nachteile: *Nicht turingmächtig*, d. h., es gibt Probleme, die zwar mit algorithmischen Sprachen, nicht jedoch mit gängigen relationalen Systemen gelöst werden können, da die meisten RDBMS Rekursion nicht unterstützen. Leider sind diese Probleme nicht nur theoretischer, sondern durchaus praktischer Natur. Beispiel: Wegsuche in großen Netzwerken wie Bahnnetz, Stromnetz, Internet etc.

Workaround: Einbettung von (relationalen) 4GL-Sprachen in 3GL-Sprachen $\Rightarrow$ *impedance missmatch*. Oder man nimmt ein SQL3-konformes RDBMS, da erst in diesem SQL-Standard Rekursion eingeführt wurde (Gulutzan und Pelzer [1999], Türker [2003]). Leider wird derzeit der SQL3-Standard im Hinblick auf Rekursion von den meisten RDBMS gar nicht oder nur rudimentär unterstützt (z. B. PostgreSQL 8.4: `WITH RECURSIVE`).

Objektorientierte DBMS

Darstellung: Klassen und Objekte

Auskünfte: Methoden

Änderungen: Methoden

Vorteile: verständlich und benutzerfreundlich, turingmächtig

Nachteile: kein mathematischer Formalismus (nur Ansätze) $\Rightarrow$ Effizienz leidet wegen fehlender Optimierungstechniken (Kempe et al. [1995]).

sehr jung und daher unausgereift im Vergleich zu relationalen Systemen;

Integritätsbedingungen werden nur sehr rudimentär unterstützt.

1.6 Besondere Aspekte von DBMS

Im Vergleich zu typischen 3GL-Anwendungen weisen DBMS einige Besonderheiten auf:

Persistenz
: Dauerhafte Speicherung der Daten muss nicht erst implementiert werden.

Öffentlichkeit, Mehrbenutzerfähigkeit
: Mehrere/viele Benutzer können ohne Programmieraufwand gleichzeitig auf die Daten der Datenbank zugreifen.

Qualität der Daten
: – Transaktionen (Robustheit im Fehlerfall: Fehleingaben, Stromausfall, …)
– Recovery (Robustheit im Katastrophenfall)
– Integritätssicherung (Gewährleistung von Datenkonsistenz)

Performanz
: Trotz Massendaten werden Anfragen schnell bearbeitet

1.6.1 Persistenz

Die Daten leben wesentlich länger als einzelne Programmausführungen. In 3GL-Sprachen kann dies allerdings mittels Dateien realisiert werden, in denen die Daten zwischen zwei Programmausführungen abgelegt werden.

1.6.2 Öffentlichkeit und Qualität

Die Daten werden von (vielen) verschiedenen Programmen verschiedener Programmautoren bearbeitet ⇒ Qualität und Richtigkeit der Daten sind fraglich.

Fehler verseuchen daher langsam eine Datenbank und leben lange
⇒ DBs „verrotten“.

Beispiel aus der Praxis: Bibliotheks-DB mit Erscheinungsjahren 98 sowie 1998 etc. abhängig vom Erfasser ⇒ Sortierung nach Erscheinungsjahren schlägt fehl.

Abhilfe: geeignete Schutzmaßnahmen

- Integritätsüberwachung
- Zurücksetzen auf fehlerfreien Zustand im Fehlerfall (Abort, Recovery)
- Benutzerverwaltung und Zugriffsschutz
- Teamwork, Groupwork, CSCW (computer supported cooperative work)

Dies alles wird von 3GL-Sprachen nicht unterstützt.

1.6.3 Simultanbearbeitung

Dieselben Daten werden gleichzeitig (parallel auf Mehr- oder pseudoparallel auf Einprozessorrechnern) von mehreren Programmen bearbeitet.

$\Rightarrow$ Synchronisation des Ablaufs durch das DBMS

Dies muss man in 3GL-Sprachen selbst programmieren (Prozesse, Threads). Gute DBMS unterstützen dies hingegen mit Hilfe von so genannten *Transaktionen* (= elementare Arbeitsschritte eines DBMS).

In diesem Zusammenhang ist der Begriff der *Serialisierbarkeit* von Bedeutung: Eine Ablaufsteuerung heißt *serialisierbar* (*seriell, konsistent*), wenn es für jede Menge $t_1, \ldots, t_n$ von parallellaufenden Transaktionen mindestens eine serielle (lineare) Anordnung dieser Transaktionen gibt, sodass die Nacheinanderausführung der einzelnen Transaktionen in dieser Reihenfolge dasselbe Ergebnis liefert wie die parallele Ausführung.

z. B. $\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \end{pmatrix} \equiv t_2; t_4; t_1; t_3$

Gegenbeispiel

Transaktion `A`: `lese a; a:=a+1, schreibe a;`
Transaktion `B`: `lese a; a:=a*2, schreibe a;`

`a:=5; A; B;` $\Rightarrow$ `a=12`
`a:=5; B; A;` $\Rightarrow$ `a=11`
`a:=5; B||A;` (`A` und `B` laufen parallel)
$\Rightarrow$ `a=12` oder `a=11` oder `a=6` oder `a=10`

Ein möglicher paralleler Ablauf:

Transaktion A	Transaktion B
`lese a` (=5)	
	`lese a` (=5)
`a := a+1` (=6)	
	`a := a*2` (=10)
	`schreibe a` (=10)
(über)`schreibe a` (=6)	

Eine serielle Ablaufsteuerung verhindert derartige Fehler (`a=6` oder `a=10`).

Die Serialisierbarkeit kann z. B. garantiert werden, indem jede Transaktion alle Ressourcen sperrt, die sie zur Bearbeitung ihrer Aufgabe benötigt. Dabei muss die Transaktionssteuerung allerdings dafür Sorge tragen, dass es nicht zu Deadlocks kommt.

Transaktion A:
: hat Sperre auf Resource `a`, wartet auf Freigabe von Resource `b`

Transaktion B:
: hat Sperre auf Resource `b`, wartet auf Freigabe von Resource `a`

Dies kann z. B. vermieden werden, indem Transaktionen Recourcen immer nur in einer bestimmten Reihenfolge anfordern dürfen oder indem Transaktionen alle benötigten Ressourcen stets nur auf einmal zugewiesen bekommen. Auch *Livelocks* müssen verhindert werden. Von einem Livelock spricht man, wenn eine Transaktion dauerhaft auf die Zuweisung von Ressourcen warten muss, weil ständig andere Transaktionen bevorzugt werden.

ACID-Transaktion

Alle guten DBMS unterstützen so genannte ACID-Transaktionen (vgl. z. B. Elmasri und Navathe [2002]). ACID ist eine Abkürzung für:

A: Atomicity (Atomizität, Atomarität)
: Eine Transaktion wird entweder ganz oder gar nicht ausgeführt.

C: Consistency (Konsistenz)
: Eine Transaktion wird nur dann erfolgreich beendet, wenn danach (weiterhin) alle Integritätsbedingungen erfüllt sind.

I: Isolation
: Eine Transaktion muss so ausgeführt werden, als ob sie allein ausgeführt werden würde. Das heißt, andere parallel laufende Transaktionen verändern das Ergebnis nicht. Mehrere Benutzer können gleichzeitig auf die Daten zugreifen, ohne dass es zu Inkonsistenzen kommt. Serialisierbarkeit wird immer garantiert.

D: Durability (Dauerhaftigkeit)
: Die Ergebnisse einer erfolgreich beendet Transaktion sind dauerhaft verfügbar (auch wenn das zugehörige Bearbeitungsprogramm beendet wird), solange nicht eine andere Transaktion weitere Modifikationen vornimmt.

ACID-Transaktionen sollten i. Allg. relativ kurz sein, da sonst evtl. bestimmte Resourcen längerfristig nicht von anderen Transaktionen genutzt werden können. Für lange Transaktionen gibt es andere Transaktionsmodelle, die vor allem nicht die strenge Atomizität fordern.

1.6.4 Performanz

Ein DBMS sollte in der Lage sein, große Mengen von Daten effizient zu verwalten, sonst darf es sich nicht Datenbank-Management-System nennen.

Eine Tabelle kann problemlos mehrere Millionen Einträge enhalten (Man denke nur an die Flensburger Verkehrssünder-Datei).

Ein einzelnes Element einer Tabelle kann mehrere Gigabyte groß sein – wenn z. B. Videofilme in der Datenbank gespeichert werden.

Eine Datenbank von mehreren Terabyte Größe ist somit keine Utopie (Beispiel: `www.flickr.com`, hier werden in jeder Minute mehrere Tausend Bilder eingefügt). Trotzdem sollten Anfrage- und Modifikations-Operationen schnell bearbeitet werden können. Bei SAP-Systemen gilt beispielsweise eine mittlere Bearbeitungszeit von zwei Sekunden oder länger für einen Arbeitsschritt eines Sachbearbeiters als zu lang (Kowarschick [1999], Wilhelm et al. [2001]). Oder denken Sie an die Suchmaschine Google: Für jede Suchanfrage wurden bereits 2009 mehr als acht Milliarden Dokumente durchsucht.[6] Trotzdem erfolgt die Antwort in Sekundenbruchteilen.

Um eine derartige Performaz zu ermöglichen, werden verschiedene Techniken eingesetzt.

Zum einen werden Daten in besonderen Datenstrukturen abgelegt: B-Bäume (heute meist Präfix-B-Bäume), Hash-Arrays (für Plattenspeicherung optimiert), R-Bäume (für räumliche Daten), UB-Bäume (mehrdimensionaler Index für räumliche Daten, Data Warehouses etc.), ISAM-Dateien (Index Sequenzielle Dateien, schon älter) etc. (siehe z. B. Elmasri und Navathe [2002]; UB-Bäume: Ramsak et al. [2000])

Neben der geeigneten Speicherung der Daten spielt die Anfrage-Optimierung eine wichtige Rolle. Spezielle Optimierer stellen Anfragen so um, dass sie möglichst schnell, aber dennoch fehlerfrei beantwortet werden können. Dies ist möglich, wenn den DB-Operationen ein wohldefiniertes (d. h. mathematisch beschreibbares) Verhalten zu Grunde liegt. Das ist insbesondere bei relationalen Systemen der Fall (leider aber nicht bei objektorientierten Systemen).

Zu guter Letzt muss es für die einzelnen Datenbank-Operationen effiziente Algorithmen geben, wie z. B. für relationale Operationen. Insbesondere können hier viele Algorithmen parallisiert werden. Das heißt, mit Parallelrechnern kann die Performaz häufig auf das gewünschte Maß erhöht werden (Skalierbarkeit).

1.7 B-Baum

Der B-Baum wurde 1970 von R. Bayer und Ed McCreight entwickelt (Bayer und McCreight [1970]). Die Definitionen und Sätze dieses Abschnitts wurden im Wesentlichen von Rudolf Bayer (Bayer [1982]) übernommen.

[6] www.google.com/intl/de/options/; Punkt „Websuche“; 10.9.2009.

Definition

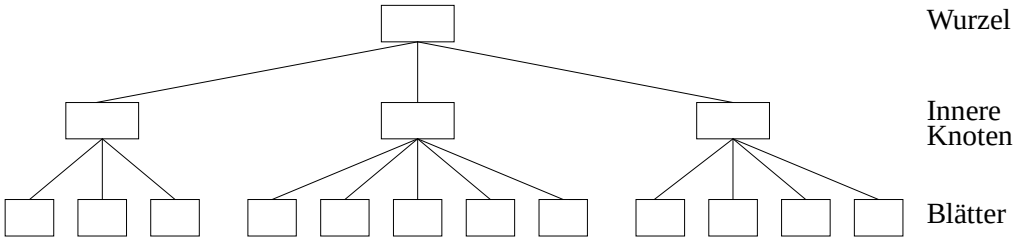
Es seien b und h zwei natürliche Zahlen mit $b > 0$ und $h \geq 0$.

Ein B-Baum mit der Breite b (genauer: mit dem *minimalen Verzweigungsgrad* $b + 1$) und der Höhe h ist entweder ein leerer Baum (falls $h = 0$) oder ein geordneter Baum mit folgenden Eigenschaften:

1. Jeder Pfad von der Wurzel zu einem Blatt enthält genau h Knoten (und $h - 1$ Kanten); der Baum ist *vollständig balanziert*.
2. Mit Ausnahme der Wurzel und der Blätter hat jeder Knoten mindestens $b + 1$ Kindknoten.
3. Die Wurzel hat, sofern sie kein Blatt ist, mindestens 2 Kindknoten.
4. Kein Knoten hat mehr als $2b + 1$ Kindknoten.

Beispiel

In einem B-Baum der Breite $b = 2$ und der Höhe $h = 3$ muss jeder Knoten innerhalb des Baumes zwischen $b + 1 = 3$ und $2b + 1 = 5$ Kindknoten haben:



Ein Knoten ist wie folgt aufgebaut:

Dabei sind v_i Verweise auf Kindknoten, s_i Suchschlüssel und d_i Daten, die zum jeweiligen Suchschlüssel gehören.

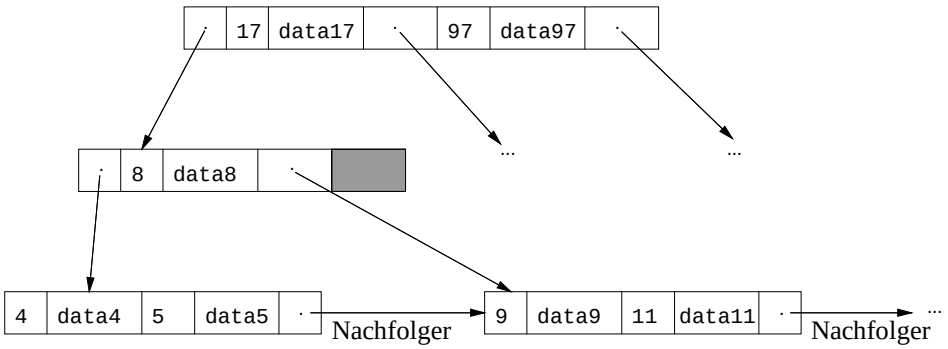
Ein Datum wird ausgehend von der Wurzel rekursiv gesucht:

- Befindet sich der gesuchte Schlüssel in einem Knoten, so ist das Datum (d. h. der Schlüssel und die zugehörigen Daten) in diesen Knoten.
- Ansonsten muss in einem Kindknoten nach dem Schlüssel gesucht werden.
- Ist der gesuchte Schlüssel kleiner als s_1, suche rekursiv in Kind v_1
  Ist der gesuchte Schlüssel kleiner als s_2, suche rekursiv in Kind v_2
  usw.
- Sonst suche rekursiv in Kind v_n

Die Suche bricht erfolglos ab, wenn man sich in einem Blatt befindet, in dem der Schlüssel nicht enthalten ist.

Beispiel

Baum der Höhe 3 und der Breite 1



Ein B-Baum der Breite b und der Höhe $h > 0$ enthält mindestens

$$\begin{aligned} K_{min}(b,h) &= 1 + 2 \cdot ((b+1)^0 + (b+1)^1 + \ldots + (b+1)^{h-2}) \\ &= 1 + \tfrac{2}{b}((b+1)^{h-1} - 1) \end{aligned}$$

Knoten und höchstens

$$\begin{aligned} K_{max}(b,h) &= (2b+1)^0 + (2b+1)^1 + \ldots + (2b+2)^{h-2}) \\ &= \tfrac{1}{2b}((2b+1)^h - 1) \end{aligned}$$

Knoten. Da jeder Knoten *außer der Wurzel* mindestens b Schlüssel und jeder Knoten und höchstens $2b$ Schlüssel enthält, sind in einem Baum der Breite b und der Höhe $h > 0$ insgesamt mindestens

$$D_{min}(b,h) = 1 + b \cdot \tfrac{2}{b}((b+1)^{h-1} - 1) = 2(b+1)^{h-1} - 1$$

und höchstens

$$D_{max}(b,h) = 2b \cdot \tfrac{1}{2b}((2b+1)^h - 1) = (2b+1)^h - 1$$

Schlüssel (d. h. Datensätze) enthalten.

Beispiel

Es sei $b = 100$. Dann gelten für einen Baum der Höhe h folgende Grenzen:

Höhe h	K_{min}	D_{min}	K_{max}	D_{max}
0	0	0	0	0
1	1	1	1	200
2	3	201	202	40 400
3	307	20 401	40 603	8 120 600
4	20 607	2 060 601	8 161 204	1 632 240 800

Vorteile eines B-Baumes

- Die Knotengröße wird so gewählt, dass bei *einem Plattenzugriff* immer ein ganzer Knoten gelesen wird.
- Ein Baum ist (bei großem b) sehr breit und flach (siehe obiges Beispiel).
- Um auf einen Datensatz zuzugreifen, sind daher nur wenige Plattenzugriffe notwendig (insbesondere, wenn die Wurzel und die obersten ein bis zwei Ebenen im Hauptspeicher gecached werden).
- Auch die Einfüge-, Lösch- und Modifikationsoperationen sind sehr billig ($O(h)$).
- Wenn man jeden Blattknoten mit seinem Nachfolger verlinkt, kann man die gesamten Daten soriert gemäß dem Sortierschlüssel direkt von der Platte lesen.

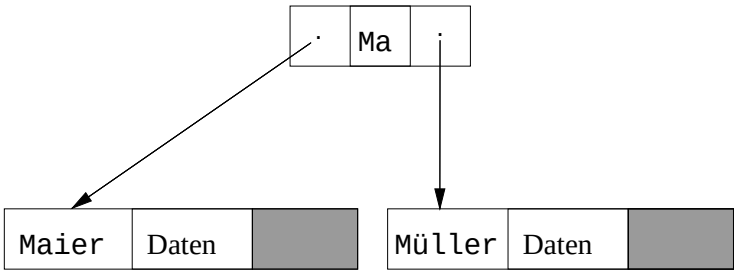
Heute werden i. Allg. Varianten des B-Baumes verwendet.

Beim B^+-Baum (R. Bayer spricht vom B^*-Baum) werden Daten nur noch in den Blättern gespeichert. Das hat zwar den Nachteil, das für jedes Datum stets genau h Knoten gelesen werden müssen, weil sich alle Daten in den Blättern befinden. Dies ist jedoch eigentlich kein wirklicher Nachteil, da auch beim normalen B-Baum der allergrößte Teil der Daten in den Blättern gespeichert wird.

Der wesentliche Vorteil des B^+-Baums ist, dass in den Zwischenknoten wesentlich mehr Platz für Schlüssel/Verweis-Paare vorhanden ist. Das heißt, die Breite b kann wesentlich größer gewählt werden. Und damit ist die Höhe eines B^+-Baums i. Allg. deutlich geringer, als die eines „normalen“ B-Baums. Das heißt, im Durchschnitt wird ein Knoten in einem B^+-Baum deutlich schneller gefunden, als in einen B-Baum.

Nochmals vergrößert werden kann die Breite b, wenn man in den Zwischenknoten nicht die vollständigen Suchschlüssel speichert, sondern nur Präfixe davon, die eindeutig festlegen, in welchen Kindknoten man weitersuchen muss. Derartige Bäume heißen Präfix-B^+-Bäume (bzw. bei Bayer Präfix-B^*-Bäume).

Beispiel



Eine weitere Optimierungsmöglichkeit besteht darin, bei einem Datenüberlauf in einem Knoten nicht sofort einen neuen Knoten zu erzeugen, sondern erst zu prüfen, ob in einem Geschwisterknoten noch Platz vorhanden ist, wohin die überzähligen Daten ausgelagert werden können. Durch diesen Trick schafft man es, den durchschnittlichen Füllgrad der Knoten zu erhöhen, d. h. die Anzahl der Knoten weiter zu reduzieren.

Kapitel 2

Entity-Relationship-Modelle

Ursprung: Chen [1976]

2.1 Entities

Die *Entity-Relationship-Modelle* (*ER-Modelle*) können als Vorläufer der modernen *objektorientierten Modelle*, wie z. B. OMT (Rumbaugh et al. [1993]) oder UML (Jeckle et al. [2003]), angesehen werden.

Objekte werden im ER-Modell *Entities* genannt. Im Kontext von Datenbanken spricht man von *Tupeln*. Entities haben *Attribute*, den Methodenbegriff gibt es allerdings – zumindest im ursprünglichen ER-Modell von Chen [1976] – nicht. Eine Menge von Entities mit denselben Attributen (aber i. Allg. unterschiedlichen Attributwerten) heißt *Entity-Menge* (*entity set*). Im Kontext von Datenbanken spricht man von *Tabellen oder Relationen*. Tabellen werden allerdings auch zur Repräsentation von Beziehungen (siehe Abschnitt 2.3 und Kapitel 3) eingesetzt. Daher gilt, dass jeder Entity-Menge eine Tabelle in der zugehörigen Datenbank zugeordnet ist. Umgekehrt gibt es aber nicht für jede Tabelle in der Datenbank eine zugehörige Entity-Menge.

Entity-Mengen entsprechen den *Klassen-Extensionen* in objektorientierten Systemen. Einen Entitynamen zusammen mit den zugehörigen Attributnamen und -typen bezeichnet man als *Entitytyp*. Entitytypen entsprechen *Klassen-Definitionen* in objektorientierten Systemen.

Heute werden meist so genannte EER-Modelle (Extended ER) eingesetzt. Diese haben viele Aspekte der objektorientierten Modellierung übernommen, wie z. B. die Vererbung. Im weiteren subsumiere ich unter dem Begriff ER alle gängigen ER-Varianten, d. h. auch EER-Modelle.

Ein wesentlicher Unterschied zu objektorientierten Modellierungstechniken ist jedoch, dass Entities normalerweise keine OIDs im objektorientierten Sinne zugeordnet sind. Anstelle dessen werden so genannte *Schlüsselattribute* gekennzeichnet, deren Werte eine Entity eindeutig identifizieren. Ein typisches Schlüsselattribut ist `personalnummer`. In ER-Modellen ist der Designer also selbst verantwortlich für die Vergabe von Identifikatoren (siehe Abschnitt 2.2).

Ein anderer Unterschied zu objektorientierten Modellen ist, dass in ER-Modellen meist direkt mit öffentlichen Attributen gearbeitet wird. Methoden und Kapselung werden normalerweise nicht unterstützt.

Entity-Typen werden von mir ähnlich wie UML-Klassen (Jeckle et al. [2003]) notiert. Schlüsselattribute (siehe Abschnitt 2.2) werden durch Unterstreichen oder (UML-konform) durch die Integritätsbedingung `{PK}` (für Primary Key) gekennzeichnet.

Attribute, die den Wert `NULL` annehmen dürfen, wenden mit einem Stern (`*`) oder (UML-konform) durch die Angabe der Integritätsbedingung `{NULLABLE}` gekennzeichnet.

Die *Attributtypen* (= *Domänen*) notiere ich häufig nicht. In diesen Fällen ist der Typ entweder irrelevant oder selbsterklärend. Im Zweifelsfall kann `STRING` (= `VARCHAR`) für den fehlenden Attributtyp verwendet werden.

Beispiel für einen Entity-Typ zusammen mit einem zugehörigen Tupel/Objekt in Quasi-UML-Notation und als Tabelle:

Klasse:

person
id: INTEGER name: STRING geb: DATE* tel: STRING

person	
id:	INTEGER {PK}
name:	STRING
geb:	DATE {NULLABLE}
tel:	STRING

Objekt:

wolfgang
id: 1 name: 'Wolfgang' geb: 1961 tel: '3745'

person

id	**name**	**geb**	**tel**
1	'Wolfgang'	1961	3745
2	'Sabine'	NULL	3465
...			

2.2 Schlüssel

Um die Tupel einer Relation eindeutig identifizieren zu können, führt man die so genannten *Schlüssel* ein.

Eine Entity-Menge ist im Wesentlichen eine *Menge* von Tupeln:

```
{(name: 'Wolfgang', geb: 1961, tel: '3475'),
 (name: 'Sabine', geb: NULL, tel: '3465'),
 ...
}
```

Da es sich um eine Menge handelt, kann es jedes Tupel höchstens einmal geben. In unserem Betrieb gibt es also keine zwei Wolfgangs, die 1961 geboren sind und die Telefonnummer 475 haben.

In einer Entity*menge* bilden daher alle Attribute zusammen immer einen Identifikator für das Tupel, den so genannten *trivialen Schlüssel*. Im obigen Beispiel besteht der triviale Schlüssel aus den drei Attributen `name`, `geb`, `tel`.

Allerdings kann es in einem Betrieb mit vielen 1000 Mitarbeitern schon mal passieren, dass zwei Mitarbeiter den gleichen Namen und das gleiche Geburtsdatum haben. Wenn dann auch noch beide im gleichen Zimmer mit nur einem Telefon sitzen, kann man diesen Sachverhalt in einer Datenbank, in der Name, Geburtstag und Telefonnummer als Schlüssel dienen, nicht mehr darstellen.

Dieses Problem lösen die Unternehmen mit einem eindeutigen Identifikator wie z. B. einer Personalnummer, d. h. mit einem weiteren Attribut, welches für jeden Mitarbeiter eindeutig ist. Der Identifikator entspricht einem OID in objektorientierten Systemen.

```
{...
 (id: 7, name: 'Wolfgang', geb: 1961, tel: '475'),
 (id: 8, name: 'Wolfgang', geb: 1961, tel: '475'),
 ...
}
```

In diesem Fall bildet bereits das Attribut `id` alleine schon einen Schlüssel.

Definition

Es sei r eine Relation mit den Attributen $A = \{a_1, \ldots, a_n\}$ (Attributnamen).

B und C seien zwei Teilmengen von A: $B, C \subseteq A$.

C heißt *funktional abhängig* von B, in Zeichen $B \rightarrow C$, wenn *zu jedem Zeitpunkt* je zwei Tupel der Relation r, die in den Attributen B übereinstimmen, auch in den Attributen C übereinstimmen.

Definition

Es sei r wieder eine Relation mit den Attributen $A = \{a_1, \ldots, a_n\}$.

Eine Teilmenge $S \subseteq A$ heißt *Schlüsselkandidat* (*candidate key*) von r, wenn $S \rightarrow A$, d. h. wenn *alle* Attribute A von r funktional von S abhängen.

S heißt *echter Schlüsselkandidat*, wenn überdies keine echte Teilmenge $S' \subset S$ Schlüsselkandidat ist. Ansonsten heißt S *unechter* Schlüsselkandidat.

Anmerkung

Die Menge A aller Attribute einer Relation r ist immer ein Schlüsselkandidat, der so genannte *triviale Schlüsselkandidat*, da r als Menge keine Duplikate enthält.

In unserem Beispiel gibt es diverse Schlüsselkandidaten. Zum Beispiel:

`id`	$\rightarrow$ `id,name,geb,tel`	(echt)
`id,geb`	$\rightarrow$ `id,name,geb,tel`	(unecht)
`id,name,geb,tel`	$\rightarrow$ `id,name,geb,tel`	(trivial)

(Welche Eigenschaften muss eine Teilmenge $S \subset A$ haben, damit sie ein unechter Schlüsselkandidat für unser Beispiel ist?)

Im Weiteren verstehe ich unter „*Schlüsselkandidat*“ immer „*echter Schlüsselkandidat*“, da unechte Schlüsselkandidaten i. Allg. uninteressant sind.

Für jede Tabelle wird normalerweise ein so genannter *Primärschlüssel* (primary key, PK) definiert, indem aus der Menge aller (echten) Schlüsselkandidaten einer ausgewählt wird. In ER-Diagrammen werden die Primär-Schlüsselattribute durch Unterstreichen oder – UML-konform – durch Integritätsbedingungen markiert.

person
<u>id</u>
name
geb*
tel

bzw.

person
<u>id</u>: INTEGER
name: STRING
geb: DATE*
tel: STRING

bzw.

person
id {PK}
name
geb {NULLABLE}
tel

bzw.

person
id: INTEGER {PK}
name: STRING
geb: DATE {NULLABLE}
tel: STRING

Anmerkung

Ein Primärschlüssel legt häufig automatisch auch den Primärindex fest. Der Primärindex (meist ein B-Baum) beschleunigt die Suche nach Tupeln, deren Primärschlüssel bekannt ist.

Attribute, deren Wert nicht unbedingt angegeben werden muss (die also den Wert `NULL` annehmen können), werden mit einem Stern `*` markiert. Beachten Sie, dass auch Schlüsselattribute den Wert `NULL` annehmen können, meist ist dies jedoch – wie in unserem Beispiel – nicht der Fall. Im SQL3-Standard (SQL:1999, SQL:2003 und Nachfolgende) sind `NULL`-Werte in Schlüsselattributen allerdings nicht erlaubt, da `NULL` unter anderem als „vorhandener, aber unbekannter Wert“ interpretiert werden kann. Unter dieser Interpretation kann man nicht sicher ausschließen, dass die beiden Tupel

```
name: 'Wolfgang', geb: 1961 und
name: 'Wolfgang', geb: NULL
```

nicht doch dasselbe Objekt bezeichnen. Der Wert `NULL` kann allerdings auch als „Wert ist nicht vorhanden“ interpretiert werden, z. B.:

```
name: 'Sibylle', ehemann: NULL
```

In einem derartigen Fall wäre es von Vorteil, wenn auch der SQL-Primärschlüssel `NULL`-Werte enthalten dürfte. Dies ist jedoch nur für SQL-Schlüsselkandidaten (`unique`) gestattet.

Im Zusammenhang mit Schlüsseln gibt es viele Fragen:

1. Welche Attribut-Kombinationen kommen als Schlüssel in Frage?
2. Welche dieser Schlüsselkandidaten sind minimal, d. h. echt (man kann kein Attribut weglassen, ohne die Schlüsseleigenschaft zu zerstören).
3. Welcher der Schlüsselkandidaten soll als *Primärschlüssel* verwendet werden? Der Zugriff auf Primärschlüssel-Attribute kann besonders effizient erfolgen, wenn für diese Attribute ein *Primärindex* anlegt wird. Dabei hängt der Effizienzgewinn i. Allg. auch noch von der Reihenfolge der Schlüsselattribute ab, da diese die Sortierreihenfolge der Tupel im Primärindex festlegt.
4. Ist ein Schlüsselkandidat wirklich für jeden nur denkbaren Datenbankzustand ein Schlüssel? (Oft denkt ein DB-Designer nicht daran, dass es auch in einem Fünf-Personen-Betrieb **irgendwann einmal** zwei „Sepp Maier“ geben kann und so die Attribute `nachname` und `vorname` gar keinen Schlüsselkandidaten bilden.)
5. Ist ein Schlüssel eindeutig bzgl. einer Relation oder bzgl. der ganzen Datenbank? (OIDs sind bzgl. einer oder gar mehrerer Datenbanken eindeutig.)
6. Sind Schlüssel wiederverwendbar (Personalnummern, Matrikelnummern, Autonummern)?

Beispiel 1

Relation: `pruefung` (Inhalt: alle schriftlichen Prüfungen eines Semesters)
Attribute: `pruefer, student, fach, datum, uhrzeit, raum`

Wenn der Student und das Fach vorgegeben sind, liegen auch der (Erst-)Prüfer, das Datum, die Uhrzeit und der Raum fest. Es bestehen also folgende *funktionalen Abhängigkeiten*:

`student, fach` $\rightarrow$ `pruefer`
`student, fach` $\rightarrow$ `datum`
`student, fach` $\rightarrow$ `uhrzeit`
`student, fach` $\rightarrow$ `raum`

Aber auch wenn der Student und der genaue Prüfungstermin vorgegeben sind, sind die restlichen Werte eindeutig festgelegt:

`student, datum, uhrzeit` $\rightarrow$ `pruefer, fach, raum`

Daneben kann es noch folgende Integritätsbedingung geben:

`pruefer, fach` $\rightarrow$ `datum, uhrzeit`

Daraus ergeben sich folgende Schlüsselkandidaten:

`student, fach`
`student, datum, uhrzeit`

Achtung

Eine funktionale Abhängigkeit mag auf den ersten Blick als richtig erscheinen, aber in Wirklichkeit gar nicht **immer** erfüllt sein. So könnte es für ein Fach durchaus zwei Erstprüfer geben (z. B. Medienprojekt im Studiengang IAM). Dann würden die funktionalen Abhängigkeiten

`student, fach` $\rightarrow$ `pruefer` und
`student, datum, uhrzeit` $\rightarrow$ `pruefer`

nicht gelten. Das Ergebnis wären zwei andere Schlüsselkandidaten (Welche?).

Fazit

Erst *sinnvolle* funktionale Abhängigkeiten festlegen, dann die Schlüsselkandidaten (automatisch) ermitteln, dann geeignete Schlüssel (intuitiv) wählen.

Beispiel 2

Schlüsselkandidaten für `adressen` in einem fiktiven Land:

Attribute `city, street, zip`

Die Postleitzahl `zip` legt den Ort eindeutig fest[1] und für jede Straße einer Stadt gibt es eine eindeutige Postleitzahl[2]:

`zip` $\rightarrow$ `city`
`city, street` $\rightarrow$ `zip`

Unter diesen Voraussetzungen gibt es zwei Schlüsselkandidaten:

`city, street`
`zip, street`

Aufgrund der beiden Anmerkungen (siehe Fußnoten) ergibt sich, dass dieses Beispiel realitätsfern ist. Viel besser ist es, ein künstliches Schlüsselattribut `id` einzuführen. Es gilt dann nur folgende funktionale Abhängigkeit:

`id` $\rightarrow$ `city, street, zip`

Und damit gibt es nur noch einen Schlüsselkandidaten:

`id`

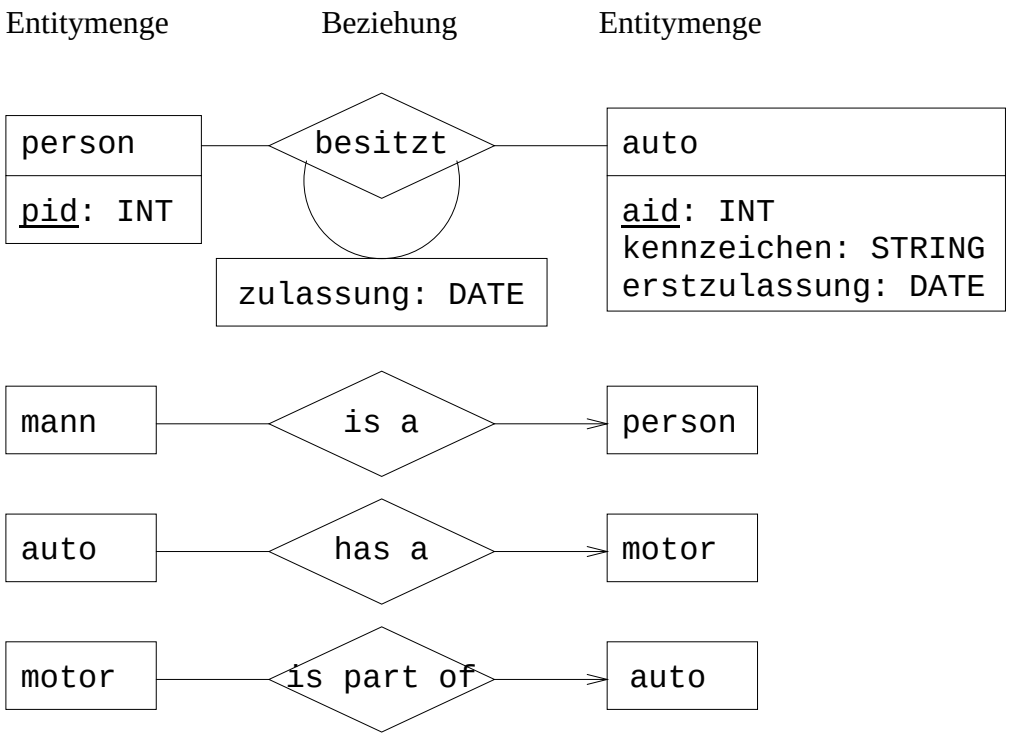
[1] Anmerkung von Michael Schulze: Diese funktionale Abhängigkeit gilt in Deutschland nicht. Es gibt mehrere Orte mit gemeinsamer PLZ, z. B. Ottobrunn und Riemerling. Aber zumindest für Städte könnte diese funktionale Abhängigkeit gelten.

[2] Anmerkung von Claus Tews: Diese Abhängigkeit gilt i. Allg. in Deutschland auch nicht. Es gibt verschiedene Orte und Städte (z. B. Berlin) mit gleichen Straßennamen (z. B. Unter den Linden). Anmerkung von WK: Und in Städten wie München gibt es Straßen mit mehr als einer Postleitzahl.

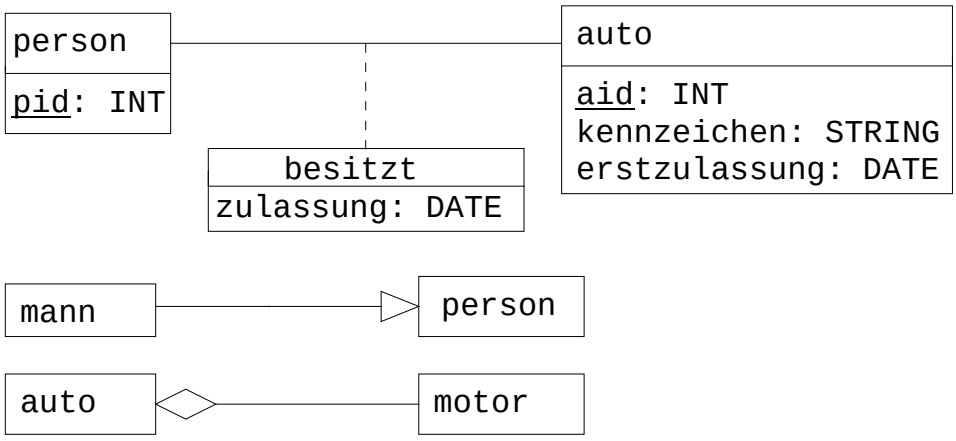
2.3 Relationships

Zwischen Entitymengen können *Beziehungen* (*Relationships*) bestehen. In UML heißen Beziehungen *Assoziationen*. Beziehungen können ebenso wie Entitytypen eigene Attribute besitzen.

Beispiele



UML-Notation

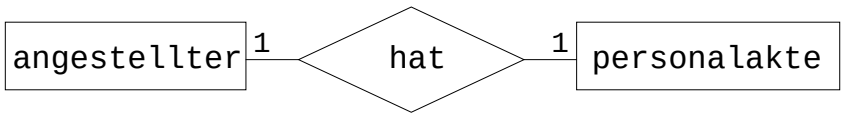


2.3.1 Vielfachheiten

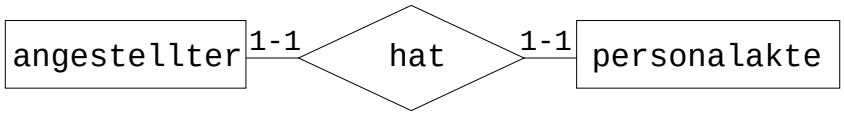
Genauso wie in UML kann man die *Vielfachheit* (*Multiplizität*) einer Beziehung genauer festlegen. Man beachte, dass die im Folgenden vorgestellte grafische Notation nur beispielhaft ist. Jedes ER-Modellierungswerkzeug verwendet seine eigene Symbolik. Ich gebe drei typische Varianten an, eine übliche ER-Notation, eine alternative ER-Notation und die entsprechende UML-Notation. Beachten Sie, dass im Falle der alternativen ER-Notation die Multiplizitäten in einer anderen Reihenfolge als bei der Standard-Notation und der UML-Notation geschrieben werden.

1:1-Beziehungen

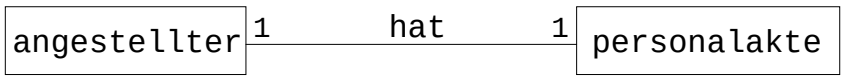
Jeder Angestellte hat genau eine Personalakte. Umgekehrt gehört jede Personalakte zu genau einem Angestellten.



Standard-ER



alternatives ER

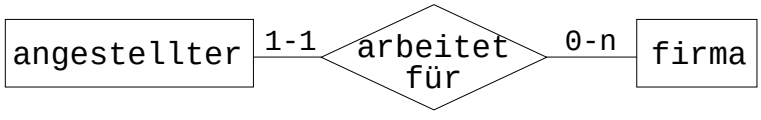


UML

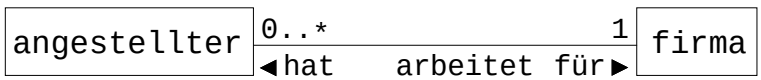
1:n-Beziehungen und n:1-Beziehungen

Jeder Angestellte arbeitet für genau eine Firma, jede Firma hat beliebig viele Angestellte.

Standard-ER



alternatives ER



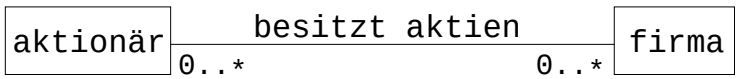
UML

m:n-Beziehungen

Jede Person kann von beliebig vielen Firmen Aktien besitzen, und jede Firma kann beliebig viele Aktionäre haben.

Standard-ER

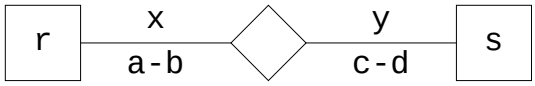
alternatives ER



UML

2.4 Veranschaulichung der Standardbeziehungen

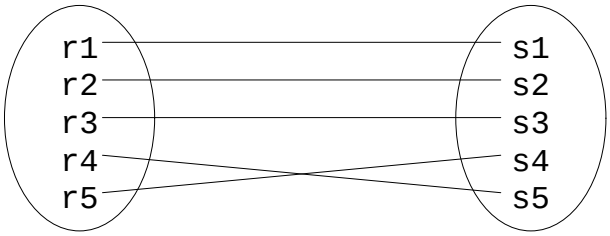
Zwei Tabellen `r` und `s` stehen in einer Beziehung zueinander.



Verschiedene Arten von Beziehungen für verschiedene Werte von x:y bzw. – alternativ – a-b, c-d werden im Folgenden veranschaulicht.

1:1-Beziehungen

Für jedes Element aus `r` gibt es genau ein Element aus `s`, und umgekehrt.

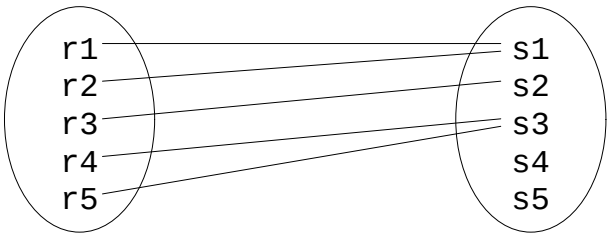


x:y = 1:1

alternativ:
a-b, c-d = 1-1, 1-1

1:n- bzw. n:1-Beziehungen

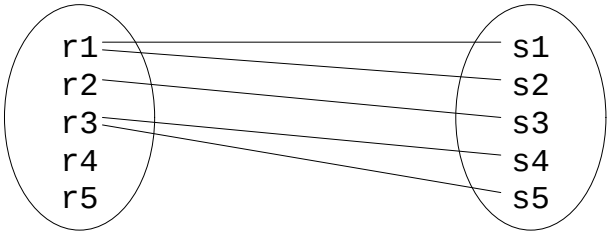
Für jedes Element aus `r` gibt es genau ein Element aus `s`; für jedes Element aus `s` gibt es beliebig viele (`n`) Elemente aus `r`.



x:y = n:1

alternativ:
a-b, c-d = 1-1, 0-n

Für jedes Element aus `r` gibt es beliebig viele (`n`) Elemente aus `s`; für jedes Element aus `s` gibt es genau ein Element aus `r`.

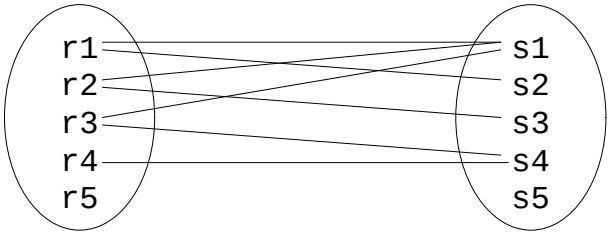


x:y = 1:n

alternativ:
a-b, c-d = 0-n, 1-1

m:n-Beziehungen

Für jedes Element aus `r` gibt es beliebig viele Elemente aus `s`, und umgekehrt.



x:y = n:m

alternativ:
a-b, c-d = 0-m, 0-n

2.5 Weitere Beziehungen

Wie Sie bereits von UML kennen sollten, kann man Multiplizitäten noch wesentlich genauer beschreiben:

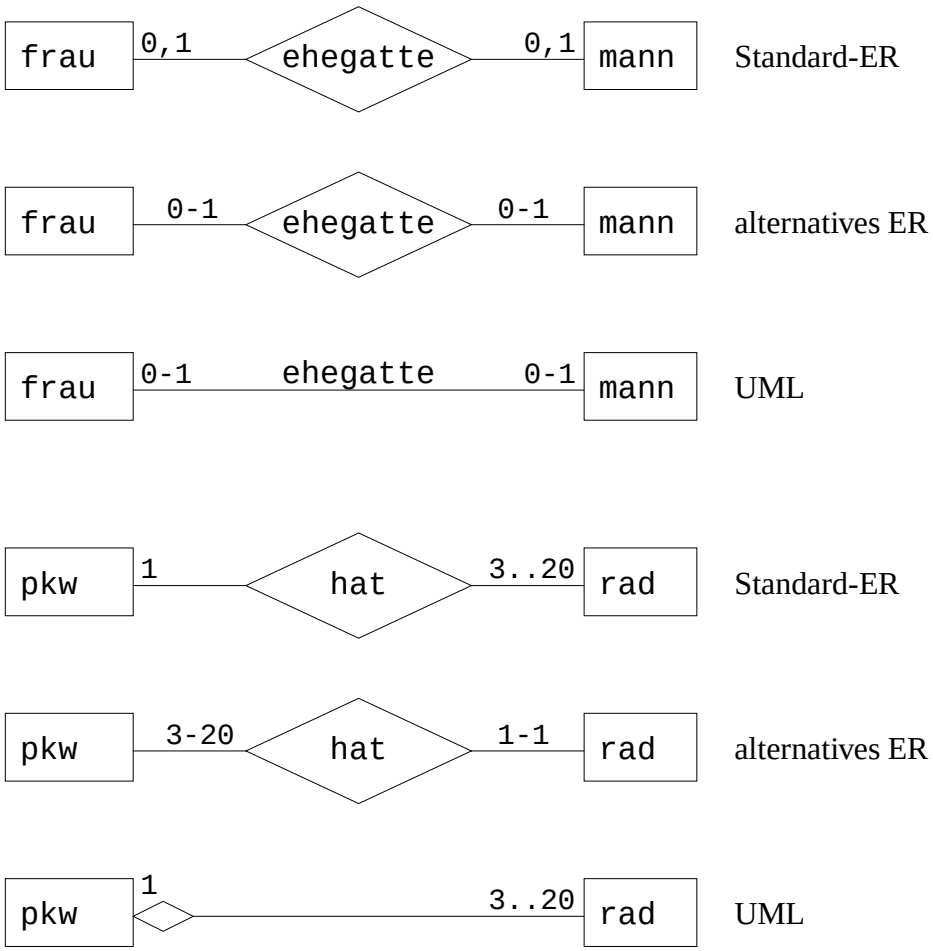
Optionalität: Jede Frau hat höchstens einen (Ehe-)Mann und umgekehrt.

Mengenbeschränkungen:
Ein PKW hat mindestens 3 (Isetta) und höchstens 20 Räder.

Oder-Assoziation:
Ein Verkehrsflugzeug hat entweder Propeller- oder Düsenantrieb.

Et cetera.

Diese Beziehungen können auch in ER-Diagrammen ausgedrückt werden:

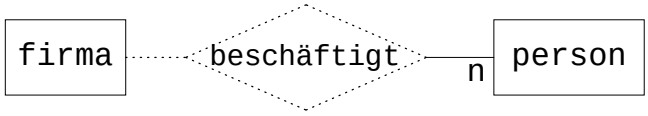


Bei der Überführung eines ER-Modells oder eines UML-Modells in ein relationales Datenbankschema ist allerdings die (grobe) Unterscheidung von 1:1-, 0,1:1-, 0,1:0,1-, 1:n-, 0,1:n- und m:n-Beziehungen zunächst meist ausreichend, da damit eine wesentliche Klasse von Integritätsbedingungen, die *Fremdschlüssel* (siehe Abschnitt 3.2.1), festgelegt ist. Eine genauere Spezifikation der Multiplizitäten wird erst in einem zweiten Schritt benötigt, und zwar bei der Formulierung von komplexeren Integritätsbedingungen, die das DBMS automatisch z. B. mit Hilfe von Triggern überwachen soll.

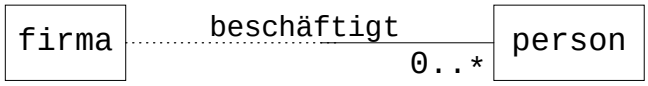
Die hier vorgestellte Art der Notation von Multiplizitäten hat (sowohl in Standard-ER-Diagrammen als auch in UML-Diagrammen – nicht aber in den alternativen ER-Diagrammen!) einen gravierenden Nachteil: Sie ist nur für Zweierbeziehungen, nicht jedoch für Dreier-, Vierer-, Fünfer-. . . Beziehungen geeignet. Der Grund dafür ist, das die Multiplizitäten in Standard-ER- und UML-Diagrammen immer bei *gegenüberliegenden* Entity-Mengen notiert werden.

Das folgende Diagramm liest man wie folgt:

> Eine Firma beschäftigt beliebig viele Angestellte.

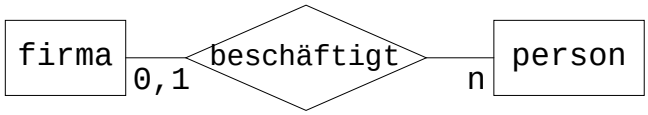


Standard-ER

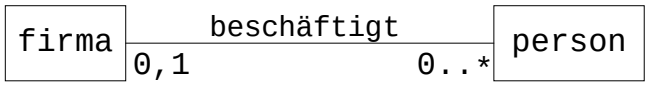


UML

Das vollständige Diagramm



Standard-ER

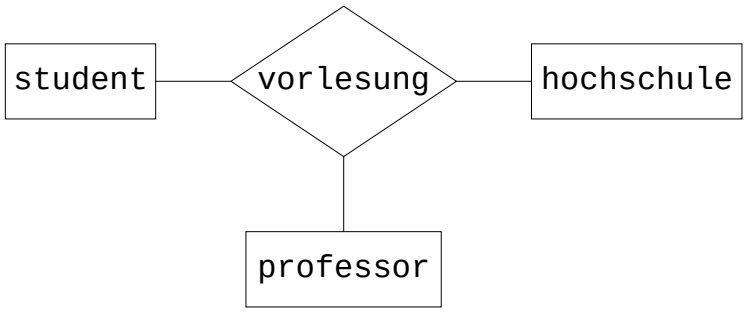


UML

liest man folgendermaßen:

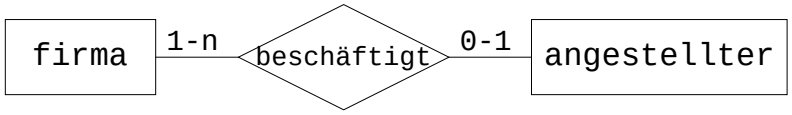
> Eine Firma beschäftigt beliebig viele Angestellte, und jeder Angestellter ist bei 0 oder 1, d. h. bei höchstens einer Firma beschäftigt.

Das funktioniert bei Mehrfachbeziehungen nicht mehr. Wie sehen hier die Multiplizitäten aus?



ER

In den alternativen ER-Diagrammen werden die Multiplizitäten daher genau anders herum plaziert:



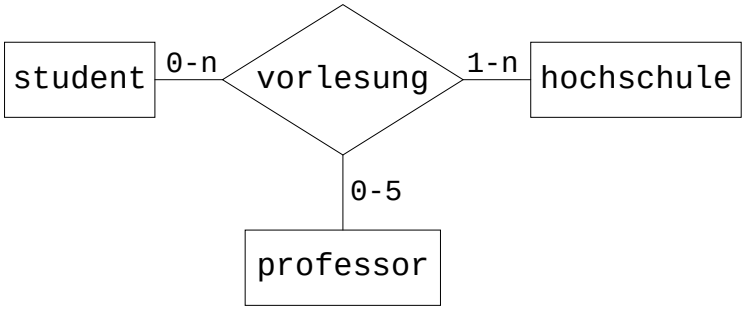
alternatives ER

Dieses Diagramm liest man wie folgt:

> Eine Firma beschäftigt 1 bis n Angestellte, d. h. beliebig viele Angestellte, mindestens jedoch einen, und jeder Angestellter ist bei keiner oder einer, d. h. bei höchstens einer Firma beschäftigt.

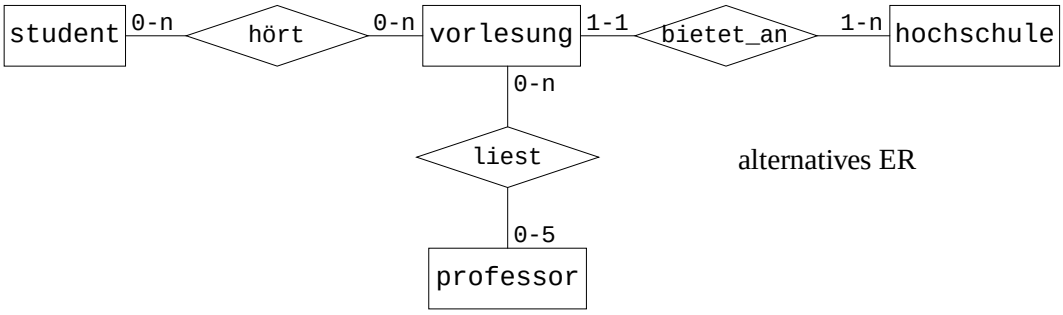
Für das obige Diagramm mit der Dreifachbeziehung können mit Hilfe der alternativen Notation folgende Integritätsbedingungen ausgedrückt werden:

> Ein Student hört beliebig viele Vorlesungen, eine Hochschule bietet viele Vorlesungen an (mindestens eine), ein Professor hält bis zu 5 Vorlesungen.

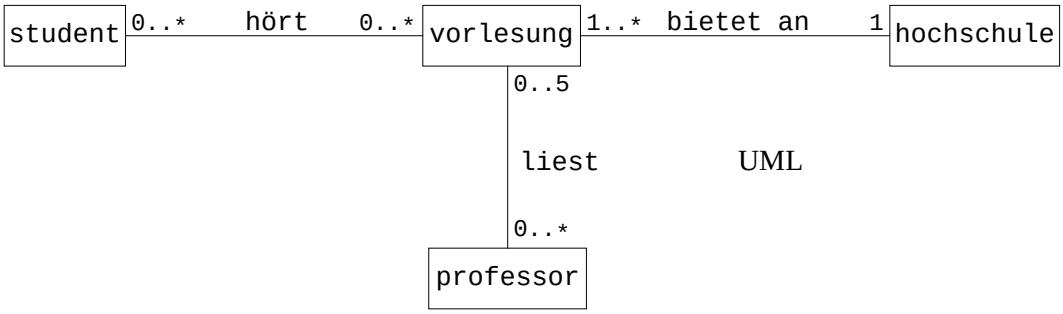


alternatives ER

Bei Modellierungstechniken wie UML oder Standard-ER, die mit Mehrfachbeziehungen nicht gut umgehen können, sollte man jede Mehrfachbeziehung durch eine neue Entity-Menge (Klasse) und mehrere Zweifachbeziehungen zu ersetzen.



alternatives ER



UML

Dieses Diagramm enthält sogar mehr Informationen als das vorherige. Zum Beispiel erfahren wir zusätzlich, dass eine Vorlesung von beliebig vielen Studenten gehört, aber nur von einer Hochschule angeboten werden kann.

Dennoch könnte es noch weitere Einschränkungen geben, die nicht direkt im Diagramm dargestellt werden können, wie z. B. „Studenten dürfen Vorlesungen nicht an mehr als zwei Hochschulen belegen“. Derartige Bedingungen werden normalerweise einfach textuell in das Diagramm eingefügt und später als (komplexe) Integritätsbedingungen formuliert.

Anmerkung 1

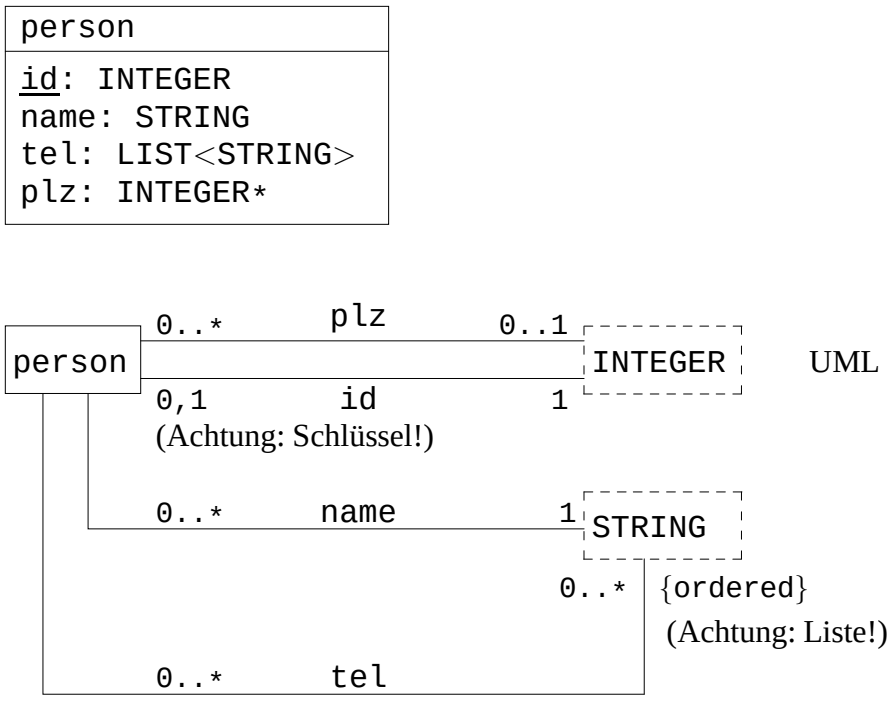
Entitymengen bzw. Klassen werden meist durch Substantive beschrieben, Beziehungen dagegen durch Verben (Person besitzt Auto). Leider ist diese Faustregel nicht allgemeingültig, wie man schon am Beispiel „hört Vorlesung“ gesehen hat.

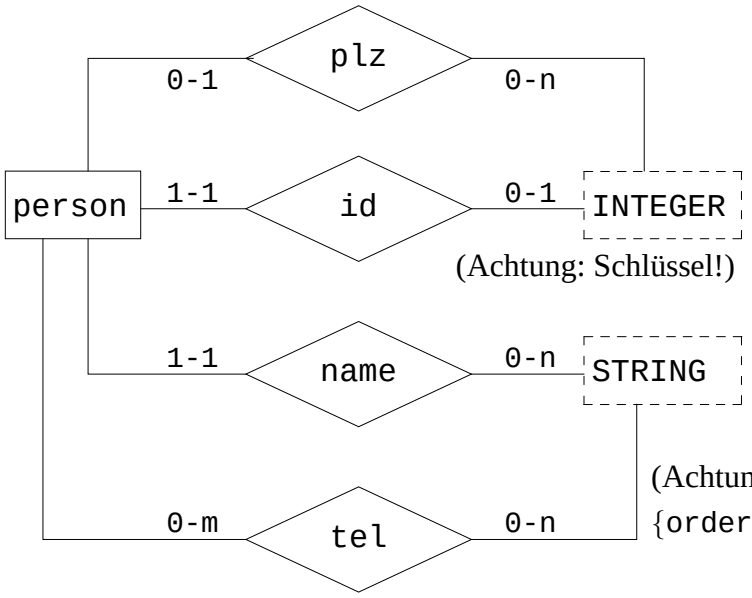
Weitere Beispiele: Prüfungen vs. prüft, Vorgang vs. bearbeitet

Wir werden im Folgenden noch sehen, dass in relationalen Datenbanken der Unterschied zwischen Entities und Relationships völlig verschwindet: Beides wird durch Tabellen dargestellt.

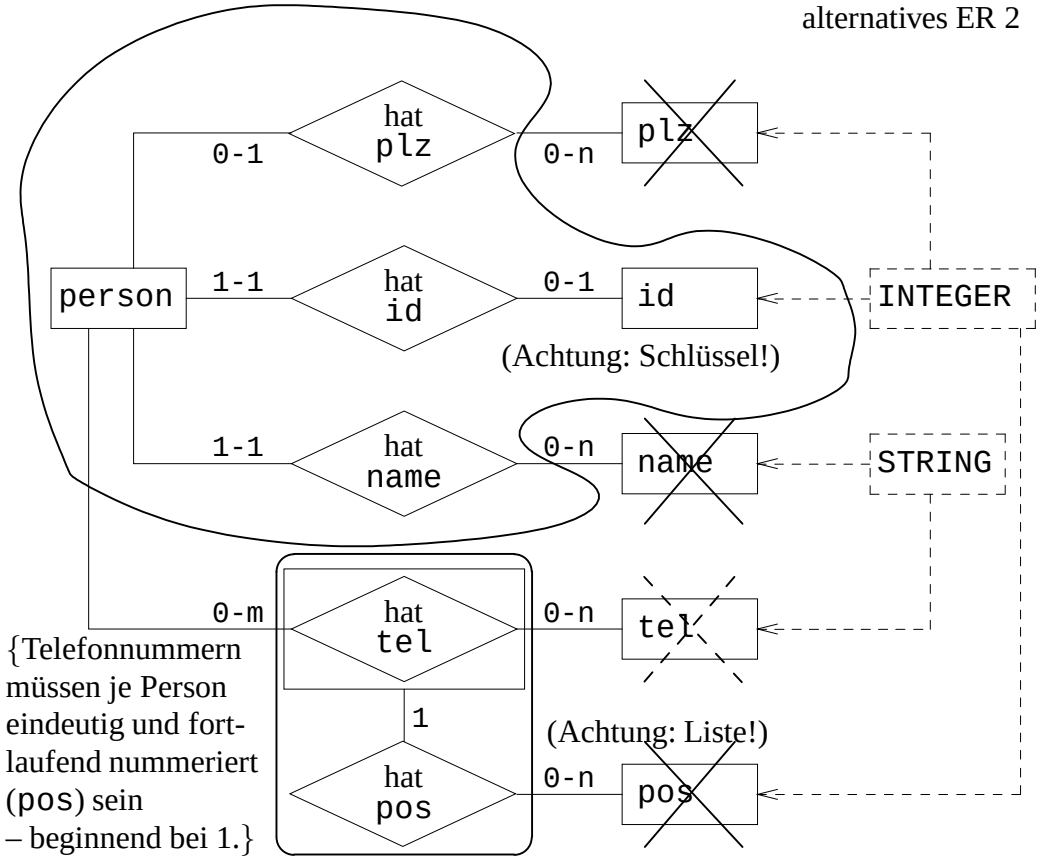
Anmerkung 2

Attribute können auch als Beziehungen zu ihren Domänen modelliert werden.





alternatives ER



Die gestrichelten Klassen sind so genannte Wert-Klassen, d. h. Klassen von Werten. Einen Wert kann man sich als ein Objekt vorstellen, das zu Anbeginn der Zeit erzeugt wurde und erst zum Ende der Zeit zerstört wird. Für einen Wert stimmt der Objekt-Identifikator mit dem Zustand überein. Da ein Objekt-Identifikator nicht geändert werden kann, kann auch der Zustand eines Wertes niemals geändert werden. Für einen Wert gibt es einen oder gar mehrere Objektnamen, mit denen in Programmen darauf zugegriffen werden kann. Zum Beispiel bezeichen `3`, `+3` und `03` alle denselben Integer-Wert mit dem OID `0000 0000 0000 0011` (rechnerinterne binäre Darstellung des Wertes).

Im Datenbank-Umfeld kann man sich eine Wert-Klasse als eine sehr große oder gar unendlich große Tabelle vorstellen, die alle möglichen Werte enthält:

Tabelle	`INTEGER`	`STRING`
Attributname	**`oid`**	**`oid`**
Werte	`0`	`''`
	`1`	`'a'`
	`-1`	`'b'`
	`2`	⋮
	`-2`	`'aa'`
	⋮	`'ab'`
		⋮

Derartige Tabellen werden allerdings nicht wirklich erzeugt. Sie wären doch etwas groß. :-)

Anmerkung 3

In dieser Vorlesung wird im Weiteren meist die bereits bekannte UML-Notation verwendet, auch wenn von Entities und Relationships die Rede ist. Es wird vorausgesetzt, dass die Grundprinzipien der objektorientierten Modellierung bekannt sind und dass der Unterschied ER $\leftrightarrow$ UML (ER: keine Methoden, keine OIDs – zumindest solange nicht mit objektrelationlen DBMS gearbeitet wird) klar ist.

Kapitel 3

Das Relationenmodell

3.1 Grundkonzepte

Das Relationenmodell (RM) von Codd [1970] kann – im Gegensatz zu den meisten objektorientierten Modellen – mathematisch exakt definiert werden. Hier sollen allerdings nur die wesentlichen Grund-Ideen sowie die Realisierung in SQL (Garcia-Molina et al. [2002] sowie nachfolgend genannte SQL-Standard-Publikationen) vorgestellt werden.

Es gibt diverse SQL-Standards:

- SQL:1989 (SQL 1)
- SQL:1992 (SQL 2, Date und Darwen [1993])
- SQL:1999 (SQL 3, Gulutzan und Pelzer [1999], Türker [2003])
- SQL:2003 (Erweiterung von SQL 3, Türker [2003], ISO:2003-1 [2006] bis ISO:2003-13 [2003])
- SQL:2006 (XML-Erweiterung, ISO:2006-14 [2006])
- SQL:2008 (einige Verbesserungen, nichts Weltbewegendes, SQL:2008-1 [2008] bis SQL:2008-14 [2008])

Meine Aussagen beziehen sich meist auf SQL3 (= SQL:1999, SQL:2003, SQL:2006, SQL:2008), hin und wieder aber auch auf SQL2 (= SQL:1992).

1. **SQL-Domänen (SQL-Datentypen)**
   Eine *SQL-Domäne* (ein *SQL-Datentyp*) ist eine Menge von Werten. Typische Domänen sind `INTEGER`, `STRING` (SQL: `CHAR`, `VARCHAR`, `CLOB`,...), `BOOLEAN`, `FLOAT` etc., `INTEGER*` (:= `INTEGER` $\cup$ {`NULL`}), `STRING*` (:= `STRING` $\cup$ {`NULL`}) etc.
   In SQL enthält jede Domäne defaultmäßig den Null-Wert `NULL`. Braucht man diesen nicht, so muss man dies explizit angeben: `INTEGER NOT NULL` etc. (`NOT NULL` ist formal eine Integritätsbedingung). In den UML-Diagrammen gebe ich dagegen mit Hilfe des Sternchens `*` explizit an, wenn auch der `NULL`-Wert erlaubt ist.

2. **Relationen**
   Eine Teilmenge $r \subseteq D_1 \times \ldots \times D_n$ eines kartesischen Produktes von Domänen heißt *Relation*. Relationen kann man sich als Tabellen vorstellen in deren i-ter Spalte nur Elemente der Domäne D_i enthalten sind:

   Beispiel

   `person` $\subseteq$ `STRING` $\times$ `INTEGER*`

   Als Menge:

   ```
   {('Wolfgang', 1961), ('Marianne', NULL), ('Sibylle', 1987), ...}
   ```

   Als Menge mit benannten Attributen:

   ```
   {(name: 'Wolfgang', geb: 1961),
    (name: 'Marianne', geb: NULL),
    (name: 'Sibylle', geb: 1987),
    ...
   }
   ```

   Als Tabelle:

name	**geb**
'Wolfgang'	1961
'Marianne'	NULL
'Sibylle'	1987
⋮	

3. **Tupel**
   Ein Element einer Relation heißt *Tupel*. Tupel stellen also die Zeilen der Tabellen dar.

4. **Attribut**
   Die „Spalten“ einer Relation heißen *Attribute*. Sie erhalten i. Allg. eindeutige (und sinnvolle) Namen. Theoretisch können sie aber auch über ihre Position identifiziert werden. Zum Beispiel kann jedes Attribut via JDBC sowohl über den Namen, als auch über die Position angesprochen werden.

5. **Relationenschema**
   Ein *Relationenschema* besteht aus einem Relationnamen gefolgt von einer Liste von Attributnamen mit zugeordneten Domänen:
   $r(a_1 : D_1, \ldots, a_n : D_n)$
   In SQL wird ein Relationenschema mit Hilfe des Befehls `CREATE TABLE` definiert.

6. **Erweitertes Relationenschema**

Ein *erweitertes Relationenschema* ist ein Relationenschema zusammen mit Integritätsbedingungen (Primärschlüssel, Schlüsselkandidaten, Fremdschlüssel etc.).

7. **Relationales Datenbankschema**
   Eine Menge von erweiterten Relationenschemata heißt Datenbankschema. Jedes Datenmodell (ER-Modell) kann mit Hilfe eines automatisierten Verfahrens in ein Datenbankschema überführt werden (siehe Abschnitt 3.2).
8. **Relationale Datenbank**
   Ein Datenbankschema (Datenmodell) zusammen mit einer Menge von zugehörigen Relationen (Fakten) heißt *relationale Datenbank* (*RDB*).
9. **Relationales Datenbank-Management-System**
   Ein System wie z. B. PostgreSQL oder Oracle zum Verwalten von realtionalen Datenbanken heißt *relationales Datenbank-Management-System* (*RDBMS*).
10. **Relationales Datenbank-System**
    Ein RDBMS zusammen mit einer oder mehreren konkreten Datenbanken wird *relationales Datenbank-System* (*RDBS*) genannt.

SQL-Domänen

Im Folgenden gibt ich einige SQL-Domänen an. Beachten Sie allerdings, dass die meisten Datenbank-Systeme weitere, nicht-standardkonforme Domänen anbieten. Auf der anderen Seite ist auch nicht sichergestellt, dass alle Domänen des Standards unterstützt werden.

Im Folgenden gelte: $n \geq 0, p \geq q \geq 0$.

`BOOLEAN` (SQL:1999)
: Wahrheitswerte `TRUE`, `FALSE`, `UNKNOWN`

`CHARACTER(`n`) = CHAR(`n`)`
: String mit genau n Zeichen

`CHARACTER VARYING(`n`) = VARCHAR(`n`)`
: String mit höchstens n Zeichen

`CLOB` (SQL:1999)
: sehr große Strings (Character Large Objects)

`BIT(`n`)`
: Bit-String mit genau n Bits (B'01101011')

`BIT VARYING(`n`)`
: Bit-String mit höchstens n Bits

`BLOB` (SQL:1999)
: sehr große Binärdateien (Binary Large Objects)

`INTEGER`
: Integerzahlen

`SMALLINT`
: Integerzahlen (weniger Bytes, oft ineffizienter als `integer`)

`BIGINT` (SQL:2003)
: Integerzahlen (mehr Bytes)

`FLOAT(`p`)`
: Fließkommazahlen (p Stellengenauigkeit)

`NUMERIC(`p`,`q`)`
: Festkommazahlen, Vorzeichen, p Stellen, davon q Nachkommastellen

`DATE`
: Datum (`date '2005-12-24'`)

`TIME`
: Uhrzeit (`time '11:13:59'`)

`TIMESTAMP`
: Datum und Uhrzeit (`timestamp '1961-05-05 23:35:00'`)

`INTERVAL`
: Zeitinterval (`interval '48' Hour`)

`XML` (SQL:2003)
: XML-Dokumente (`<title>MMDB</title>`)

`ROW` (SQL:1999)
: Tupel (`ROW('86161', 'Augsburg', 'Baumgartnerstr. 16')`)

`ARRAY` (SQL:1999)
: beschränkte Listen (Typ: `VARCHAR(2) ARRAY(2)`,
Wert: `ARRAY['de','en']`)

`REF` (SQL:1999)
: Verweise (Referenzen) → objektorientierte Erweiterung

Abkürzungen

`CHARACTER = CHAR = CHARACTER(1)`
`INT = INTEGER`
`FLOAT = FLOAT(`p`)`, p ist DBMS-spezifisch
`REAL = FLOAT(`s`)`, s ist DBMS-spezifisch
`DOUBLE PRECISION = FLOAT(`d`)`, d ist DBMS-spezifisch
`NUMERIC(`p`) = NUMERIC(`p`,0)`
`NUMERIC = NUMERIC(`p`)`, p ist DBMS-spezifisch
`DECIMAL(`p`,`q`)` wie `NUMERIC(`p`,`q`)`,
allerdings: `NUMERIC`: genau p Stellen, `DECIMAL`: mindestens p Stellen

Leider sind diese Datentypen in vielen DBMS nicht standard-konform realisiert. Zum Beispiel gilt in TransBase: `CHAR(`n`) = VARCHAR(`n`)`. Vor allem die Datums- und Zeit-Datentypen sind meist sehr proprietär realisiert.

Anmerkungen

1. *Objektorientierte Datenbanken* (OODB) werden ebenso wie RDBen über *Datenmodelle* und *Fakten* definiert:
   Datenmodelle: Klassendefinitionen (+ Methodenimplementierungen)
   Fakten: Objekte (in Klassenextensionen)
2. *Relationen* sind Mengen im mathematischen Sinn:
   $\Rightarrow$ a) Es gibt keine Duplikat-Tupel.
   b) Die Reihenfolge der Tupel ist irrelevant.
   Achtung: Laut SQL-Standard sind Relationen, mit Ausnahme der gepeicherten Tabellen, *Multimengen* (mit Duplikaten, ohne Reihenfolge). Listen (mit Duplikaten und festgelegter Reihenfolge) können i. Allg. nur mit Hilfe von speziellen Positionsattributen simuliert werden.
3. Bei der Definition einer Relation als Teilmenge eines kartesischen Produktes ist die Reihenfolge der Spalten wesentlich. Diese ist jedoch nicht mehr relevant, wenn die *Spalten eindeutig benannt* werden, was mit etwas mehr mathematischen Formalismus problemlos möglich ist. Daher verwende ich in Zukunft fast ausschließlich benannte Attribute.
4. Die Struktur einer Datenbank, d. h. das *Datenbankschema* wird durch so genannte *DDL-Anweisungen* (*Data Definition Language*, `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE` etc.) festgelegt.
5. Der Inhalt einer Relation (die Fakten) wird durch so genannte *DML-Anweisungen* (Data Manipulation Language, `SELECT`, `INSERT`, `UPDATE`, `DELETE`) abgefragt und modifiziert. Das Datenbankschema selbst wird meist auch in (leider meist nicht-SQL3-standard-konformen) Tabellen abgelegt (die so genannten *Systemtabellen*) und kann daher i. Allg. mit DML-Befehlen (`SELECT`) gelesen werden. Die Manipulation der Systemtabellen erfolgt dagegen (fast) immer mit DDL-Befehlen.
6. Für RDBS gibt es meist eine wesentliche Einschränkung. Als Attributdomänen dürfen keine komplexe Datentypen wie `LIST`, `SET` oder `BAG` verwendet werden, sondern nur primitive Datentypen wie `INTEGER`, `FLOAT`, `BOLLEAN` und `STRING` (`VARCHAR`, `CLOB`)[1]. Neuerdings (seit SQL:1999) werden auch Arrays fester Länge sowie Tupel (`ROW`) und `XML` unterstützt.

[1] `STRING` wird in vielen Sprachen wie Java oder C++ als komplexer Datentyp (Array von Characters) behandelt. In SQL ist es jedoch ein primitiver Datentyp. Sehr große Strings (CLOB = Character Large Objects) müssen allerdings meist besonders behandelt werden.

Anstelle der Tabelle

name	**hobbies**
'Wolfgang'	{'Akkordeon','Fahrrad','Flöte'}
'Sonja'	{'Flöte','Ballett'}

muss man folgende Tabelle erzeugen:

name	**hobby**
'Wolfgang'	'Akkordeon'
'Wolfgang'	'Fahrrad'
'Wolfgang'	'Flöte'
'Sonja'	'Flöte'
'Sonja'	'Ballett'

3.1.1 Erste Normalform

Eine Relation ohne komplexe Attribute heißt *Relation in erster Normalform* (1NF). Eine Relation, die dies nicht erfüllt, heißt Non-First-Normal-Form-Relation (NFNF).

Allerdings kann man auf die obige Weise nicht jede NFNF-Relation verlustfrei in eine 1NF-Relation überführen. Wenn es in der obigen Relation eine zweite Person `Wolfgang` mit anderen Hobbies (`{Fahrrad,Judo}`) gäbe, könnten beide nach der Normalisierung nicht mehr unterschieden werden. (Das wäre nur dann möglich, wenn in der nicht-normalisierten Tabelle als Primärschlüssel nicht `(name)`, sondern `(name,hobby)` definiert werden würde.) In diesem Fall muss ein zusätzlicher (künstlicher) Identifikator eingeführt werden.

pid	**name**	**hobby**
1	'Wolfgang'	'Akkordeon'
1	'Wolfgang'	'Fahrrad'
1	'Wolfgang'	'Flöte'
2	'Wolfgang'	'Fahrrad'
2	'Wolfgang'	'Judo'
3	'Sonja'	'Flöte'
3	'Sonja'	'Ballett'

Diese Tabelle enthält jedoch redundante Informationen: Die Information, welchen Namen eine Person hat, ist mehrfach gespeichert. Diese kann bei Modifikation wie z. B. Namensänderung zu Inkonsistenzen führen (Update-Anomalien). Daher ist es besser, die Informationen auf zwei oder drei Tabellen zu verteilen:

person	
pid	**name**
1	’Wolfgang’
2	’Wolfgang’
3	’Sonja’

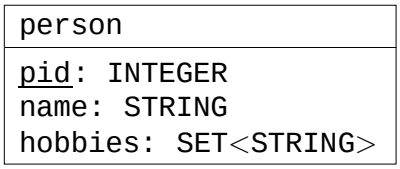
hobby	
pid	**hobby**
1	’Akkordeon’
1	’Fahrrad’
1	’Flöte’
2	’Fahrrad’
2	’Judo’
3	’Flöte’
3	’Ballett’

bzw.

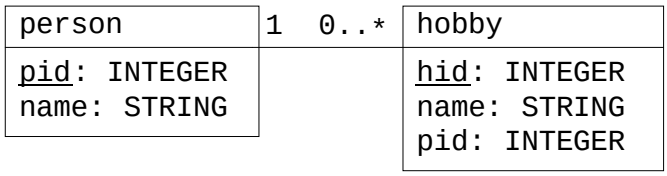
person	
pid	**name**
1	’Wolfgang’
2	’Wolfgang’
3	’Sonja’

hat_hobby	
pid	**hid**
1	1
1	2
1	3
2	2
2	4
3	3
3	5

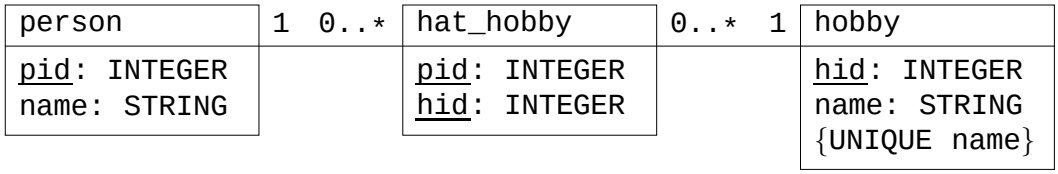
hobby	
hid	**name**
1	’Akkordeon’
2	’Fahrrad’
3	’Flöte’
4	’Judo’
5	’Ballett’



wird zu:

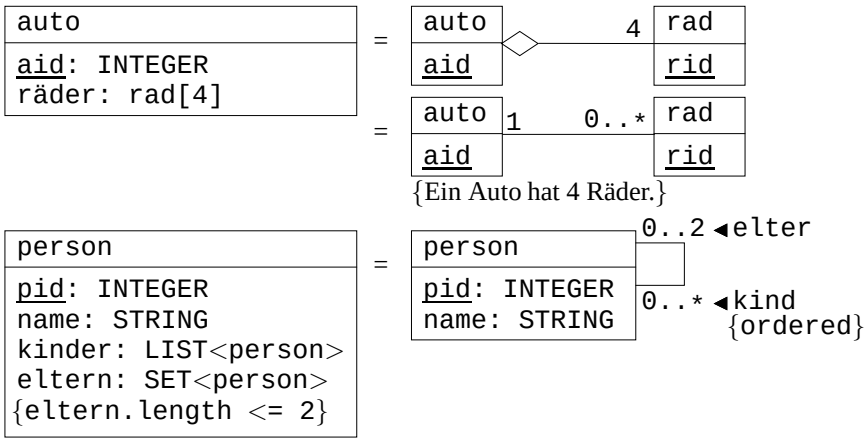


oder alternativ (besser) zu:



Objektorientierte Schemata sind i. Allg. nicht in 1NF. Es gibt auch (nicht-kommerzielle!) RDBS, die NFNF-Relationen unterstützen. Man braucht dann allerdings weitere relationale Operatoren wie `fold` und `unfold` zum Transformieren von NFNF-Relationen in 1NF-Relationen und umgekehrt.

Weitere Beispiele



3.2 Überführung eines ER-Modells in ein relationales Schema

ER-Modelle sind wesentlich ausdruckskräftiger als Relationenschemata, da zwischen Entity-Typen und Beziehungen unterschieden wird.

Daher sollte man immer zunächst ein ER-Modell erstellen und dieses dann nach „Schema F“ in ein Relationenschema transformieren.

3.2.1 „Schema F“

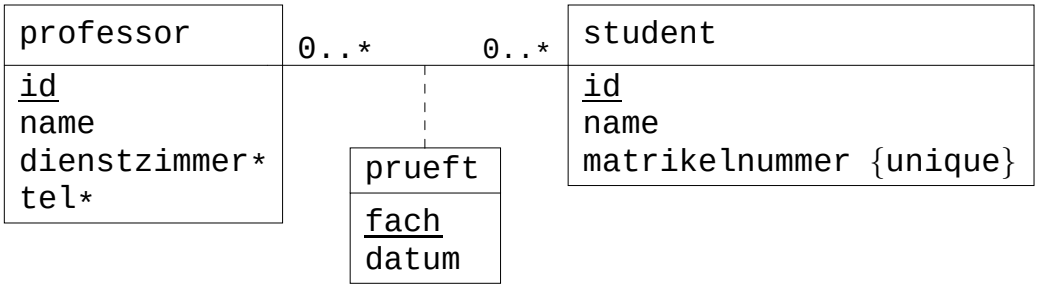
1. Komplexe Domänen müssen normalisiert werden (1 NF). Dies kann und sollte bereits im ER-/UML-Diagramm erfolgen.
2. Jeder Entitytyp e mit Attributen $a_1, \ldots, a_n$ und den zugehörigen Domänen $D_1, \ldots, D_n$ wird auf das Relationenschema $e(a_1 : D_1, \ldots, a_n : D_n)$ abgebildet. Der Primärschlüssel sowie weitere Schlüsselkandidaten (`unique`) werden übernommen.
3. Eine Beziehung (Relationship) r zwischen Entity-Typen $e_1, \ldots, e_n$ wird durch ein Relationenschema $r(\ldots)$ dargestellt, das aus allen Primärschlüssel-Attributen aller Entitytypen $e_1, \ldots, e_n$ besteht. Gleiche Attributnamen

werden dabei geeignet umbenannt. Eigene Attribute der Relationship r werden zum Relationenschema hinzugenommen. Für jede Integritätsbedingung `ordered` wird ein zusätzliches (Nicht-Schlüssel-)Integer-Attribut eingefügt, das die Sortier-Reihenfolge festlegt.

Ein Schlüsselkandidat von r kann aus den Primärschlüssel-Attributen der Ursprungsrelationen zusammen mit den Schlüsselattributen der Beziehung selbst gebildet werden. Dieser Schlüsselkandidat enthält evtl. zu viele Attribute, ist also nicht unbedingt minimal (z. B. wenn es sich um eine 1:n-Beziehung und nicht um eine m:n-Bezeihung handelt). In einem derartigen Fall muss der Schlüsselkandidat auf einen *echten* Schlüssel reduziert werden. Oder ein geeigneter anderer Schlüsselkandidat wird zum Primärschlüssel gemacht. Wie üblich geschieht dies, indem man geeignete funktionale Abhängigkeiten ermittelt. Manchmal (selten) kann es auch sinnvoll sein, für die Beziehungstabelle ein künstliches Schlüsselattribut einzuführen, z. B. weil zu der Beziehung selbst eine Beziehung besteht. *Wenn Sie nicht wissen, warum Sie den künstlichen Schlüssel brauchen, verzichten Sie darauf!*

Für alle übernommenen Primärschlüsselattribute werden Fremdschlüsselbeziehungen zu den Orginalrelationen eingetragen. Es gibt also stets ebenso viele Fremdschlüsselbeziehungen wie beteiligte Relationen.

Beispiel 1



Dieses UML-Diagramm liegt bereits in 1NF vor.

Datenbankschema (erzeugt nach „Schema F“)

```
professor: id, name, dienstzimmer*, tel*          (2. Schritt)
student:   id, name, matrikelnummer {unique}      (2. Schritt)
prueft:    pid, sid, fach, datum                  (3. Schritt)
```

Anmerkung: Der bessere Primärschlüssel von `student` wäre `matrikelnummer`. Ich wollte aber zeigen, wie man mit gleichbenannten Attributen unterschiedlicher Bedeutung umgeht. Im obigen Datenbankschema wurden die Primärschlüs-

sel-Attribute der beiden Entity-Typen `professor` und `student` geeignet umbenannt (da sie gleich benannt sind!) und zur Bezeihungstabelle `prueft` hinzugefügt. Im obigen Schema sind auch zwei *Fremdschlüsselbeziehungen* eingezeichnet. Diese werden im Anschluss an die SQL-`CREATE-TABLE`-Befehle noch genauer erörtert.

SQL-Schema

Für das obige Datenbankschema kann man in SQL drei `CREATE-TABLE`-Befehle angeben. Im Folgenden werden für jedes Attribut geeignete Datentypen angegeben (diese fehlen im UML-Diagramm) sowie zusätzliche Integritätsbedingungen (`PRIMARY KEY`, `UNIQUE`, `FOREIGN KEY`):

```
CREATE TABLE professor
 (id           INTEGER     NOT NULL,
  name         VARCHAR(20) NOT NULL,
  dienstzimmer VARCHAR(5),
  tel          VARCHAR(20),
  PRIMARY KEY (nameid)
 )

CREATE TABLE student
 (id             INTEGER     NOT NULL,
  name           VARCHAR(20) NOT NULL,
  matrikelnummer INTEGER     NOT NULL,
  PRIMARY KEY (id),
  UNIQUE (matrikelnr)
 )
```

Hier wurde neben der Primary-Key-Bedingung noch eine weitere (vom DBMS zu überprüfende) Integritätsbedingung angegeben. Jede Matrikelnummer muss eindeutig (unique) sein, d. h., zwei Studenten dürfen nicht dieselbe Matrikelnummer haben. Dies bedeutet, dass `matrikelnummer` ein Schlüsselkandidat ist.

```
CREATE TABLE prueft
 (pid     INTEGER     NOT NULL,
  sid     INTEGER     NOT NULL,
  fach    VARCHAR(20) NOT NULL,
  datum   DATE        NOT NULL,
  PRIMARY KEY (pid, sid, fach),
  FOREIGN KEY (pid) REFERENCES professor (id),
  FOREIGN KEY (sid) REFERENCES student (id)
 )
```

Bei der Überführung eine Relationship in eine Tabelle werden zwei *Fremdschlüssel*-Integritätsbedingungen definiert (*Fremdschlüssel* = foreign key). Diese besagen, dass ein Tupel „Professor A prüft Student B“ nur existieren kann, wenn es sowohl den Professor A in der Tabelle `professor` als auch den Studenten B in der Tabelle `student` gibt. Das heißt, `student` B kann beispielsweise nur gelöscht werden, wenn zuvor das obige Prüfungstupel ebenfalls gelöscht wurde (*referenzielle Integrität* = alle Fremdschlüsselbedingungen werden erfüllt).

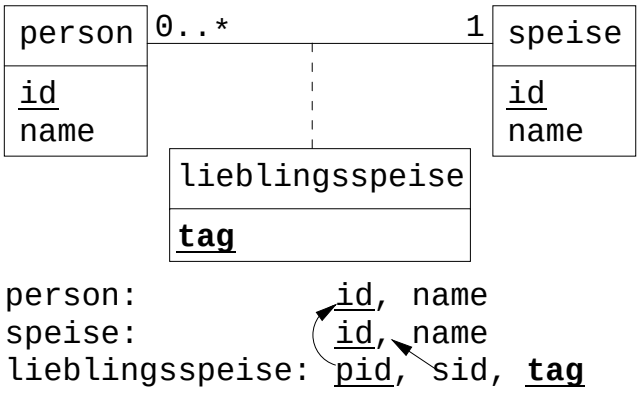
Beispiel 2a

Eine Person hat genau eine Lieblingsspeise und das auch noch an *einem speziellen Tag in der Woche* .

Man beachte, dass nicht `pid`, `sid` der Schlüssel der Beziehungstabelle `lieblingsspeise` ist, sondern nur `pname`. Das liegt daran, dass es sich bei der Beziehung um keine m:n-Beziehung handelt, sondern um eine 1:n-Beziehung. Die Fremdschlüsselbeziehungen werden davon nicht beeinflusst. Wir werden im nächsten Abschnitt sehen, dass wir dieses Schema noch optimieren können.

Beispiel 2b

Eine Person hat an *jedem Tag der Woche* genau eine spezielle Lieblingsspeise.



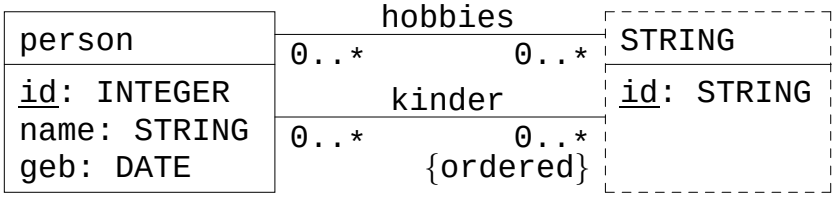
Dieses Schema kann nicht weiter optimiert werden.

Beispiel 3

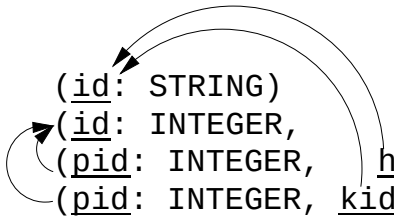
Gegeben sei folgendes Datenmodell:

person	
id: INTEGER	
name: STRING	
geb: DATE	
kinder: LIST<STRING>	{Namen der Kinder}
hobbies: SET<STRING>	{Namen der Hobbies}

Dieses wird im ersten Schritt 1NF-normalisiert (siehe Anmerkung 2 am Ende von Abschnitt 2.5):



Für das 1NF-normalisierte Modell ergibt sich im zweiten und dritten Schritt folgendes Relationenschema:



```
STRING    (id: STRING)                                              (2. Schritt)
person    (id: INTEGER,      name: STRING, geb: DATE)               (2. Schritt)
hobbies   (pid: INTEGER,   hid: STRING)                             (3. Schritt)
kinder    (pid: INTEGER, kid: STRING, kpos: INTEGER)                (3. Schritt)
```

Da die virtuelle Tabelle **`STRING`** alle existierenden Strings enthält, beschreiben die darauf verweisenden Fremdschlüsselbeziehungen genau dieselben Einschränkungen, wie die Angabe der Domäne `STRING` bei den Attributen `hid` und `kid`. Das heißt, diese Tabelle ist samt den auf sie verweisenden Fremdschlüsselbeziehungen überflüssig.

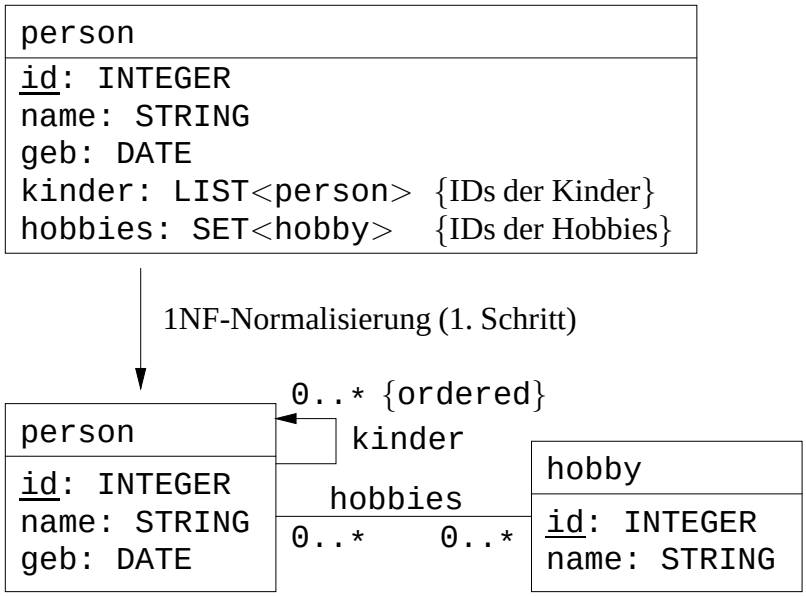
```
person    (id: INTEGER, name: STRING, geb: DATE)                    (2. Schritt)
hobbies   (pid: INTEGER, hid: STRING)                               (3. Schritt)
kinder    (pid: INTEGER, kid: STRING, kpos: INTEGER)                (3. Schritt)
```

Als Hobby- und als Kind-Name kann dementsprechend jeder beliebige String eingetragen werden. Im Falle der Kind-Beziehung ist das sicher unerwünscht. Hier sollten nur Namen (genauer: Identifikatoren) von existierenden Personen angegeben werden können.

Dies kann zum Beispiel erreicht werden, indem in der ursprünglichen Personen-Klasse `kinder: LIST<person>` anstelle von `kinder: LIST<STRING>` geschrieben wird.

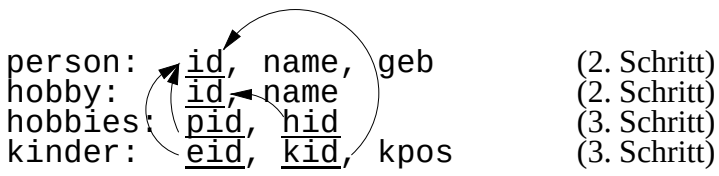
Auch bei den Hobbies macht es Sinn, die Auswahl mit Hilfe einer einer speziellen `hobby`-Klasse (bzw.-Tabelle) zu beschränken. Dadurch kann man zum Beispiel unterschiedliche Schreibweisen für ein und dasselbe Hobby vermeiden. In einer Web-Oberfläche würde nun nicht mehr ein Textfeld, sondern eine Drop-Down-Liste zur Eingabe eines Hobbies angezeigt werden. Allerdings sollte es über die Oberfläche jederzeit auch möglich sein, diese Drop-Down-Liste (d. h. die Tabelle `hobby`) zu erweitern.

Mit diesen Modifikationen erhält man folgendes Datenmodell:



Beachten Sie, dass für die Kinder eine unidirektionale Beziehung verwendet wurde. Die Tatsache, dass für jede Person (bis zu) zwei Eltern bekannt sein können, wurde im Ausgangsmodell nicht modelliert. Daher wurde dies auch im 1NF-normalisierten Modell nicht berücksichtigt. Beachten Sie außerdem, dass die Liste der Kinder geordnet ist. Dies wird im normalisierten Modell durch die Integritätsbedingung `ordered` ausgedrückt.

Für dieses Modell erzeugt der Algorithmus folgendes Schema:



Die Tatsache, dass die Kinder-Beziehung unidirektional ist, hat keinerlei Einfluss auf das resultierende Realtionenschema. Die Integritätsbedingung `ordered` wurde dagegen berücksichtigt: Es wurde das zusätzliche Attribut `kpos` (wie auch schon beim vorhergehenden Schema) eingeführt.

Wolfgangs erstes Kind heißt Sibylle, das zweite Sonja und das dritte Verena. Marianne hat dieselben Kinder. Diese Fakten können nun in der Datenbank gespeichert werden.

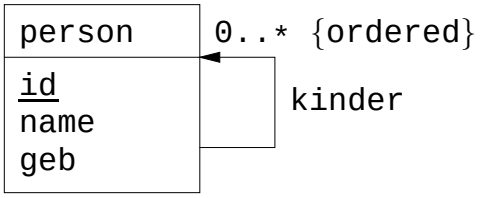
Tabelle **person**

id	**pname**	**geb**
1	'Wolfgang'	1961
2	'Marianne'	...
3	'Sibylle'	...
4	'Sonja'	...
5	'Verena'	...

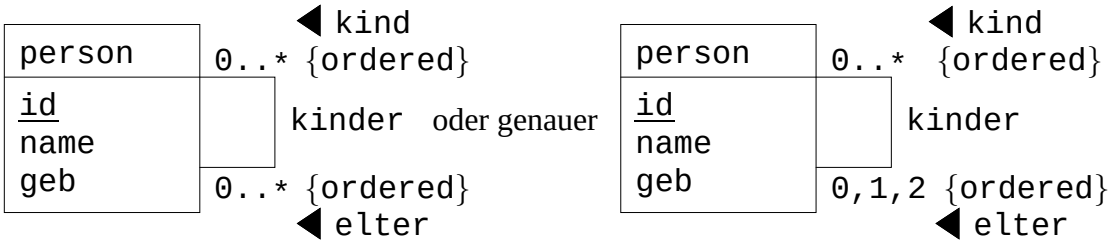
Tabelle **kinder**

eid	**kid**	**kpos**
1	3	1
1	4	2
1	5	3
2	3	1
2	4	2
2	5	3

Eine weitere Verfeinerung des Modells ergibt sich aus der Tatsache, dass ein mengenwertiges Attribut wie `kinder: LIST<person>` eigentlich nur eine Seite einer Beziehung beschreibt. Dies habe ich durch eine unidirektionale Bezeihung ausgedrückt:



Eine binäre Beziehung kann jedoch immer von zwei Seiten betrachtet werden. So auch hier:



Von einer Person sind bis zu zwei Eltern bekannt. (Diese Tatsache wurde im ursprünglichen Modell gar nicht modelliert. Hier war der Präzisionsverlust wohl etwas zu groß.) Ich habe sie geordnet, damit Mutter (Position 1) und Vater (Position 2) unterschieden werden können.

Beachten Sie, dass sich die relationalen Schemata für die unidirektionale und die bidirektionale Variante nur leicht unterscheiden.

unidirektional: `kinder: pid, kid, kpos`

bedirektional: `kinder: pid, kid, ppos, kpos`

Wären die Eltern nicht sortiert, ergäben sich gar keine Änderungen.

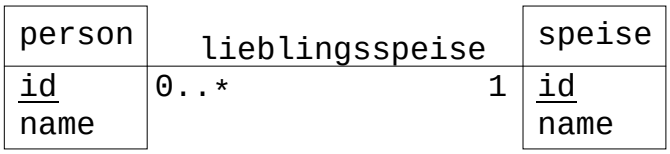
Man kann außerdem die Tatsache, dass es nur zwei Eltern gibt, zur Optimierung des Modells einsetzen. Dies ist eine weitere Verfeinerung des ursprünglichen Modells! Wir werden darauf im nächsten Abschnitt zurückkommen.

3.2.2 Optimiertes „Schema F“

Das vorgestellte Verfahren zur Überführung von ER-Diagrammen in Relationenschemata funktioniert für jedes ER-Diagramm, d. h., für beliebige Entitytypen und beliebige Relationen. Allerdings ist das Verfahren für 1:n-, n:1-, 1:1-Beziehungen und andere spezielle Beziehungen nicht optimal, da man in diesen Fällen die gesonderte „Relationship-Tabelle“ einsparen kann ($\Rightarrow$ Join-Einsparung).

1:n- bzw. n:1-Beziehungen

Bei 1:n- bzw. n:1-Beziehungen kann man zu derjenigen Tabelle, in der Elemente mehrfach referenziert werden, die Primärschlüssel-Attribute derjenigen Tabellen hinzufügen, in der alle Elemente höchstens einmal referenziert werden. Dies ist eine typische Fremdschlüssel-Beziehung.



Datenbankschema

`person: id, name, lid` (ID der Lieblingsspeise)

`speise: id, name`

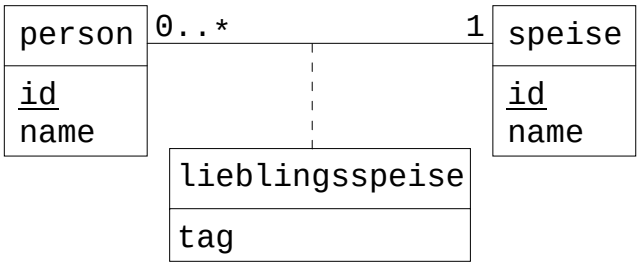
SQL-Schema

```
CREATE TABLE person
 (id   INTEGER     NOT NULL,
  name VARCHAR(20) NOT NULL,
  lid  VARCHAR(20) NOT NULL,
  PRIMARY KEY (id),
  FOREIGN KEY (lid) REFERENCES speise(id)
 )
```

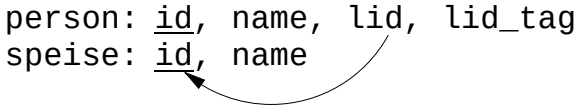
```
CREATE TABLE speise
 (id   INTEGER     NOT NULL,
  name VARCHAR(20) NOT NULL,
  PRIMARY KEY (id)
 )
```

Diese Art der Optimierung funktioniert auch, wenn die Beziehung eigene Attribute hat, aber keine eigenen Schlüsselattribute.

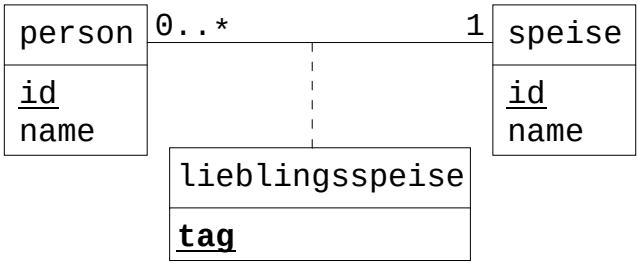
Folgendes Modell kann optimiert werden:



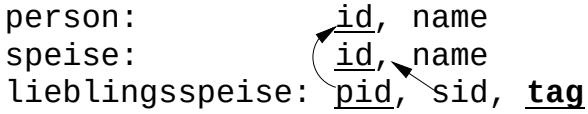
Eine Person hat genau eine Lieblingsspeise an einem speziellen Tag in der Woche.



Folgendes Modell kann dagegen nicht optimiert werden:



Jede Person hat jeden Tag genau eine Lieblingsspeise, wie z. B. z. B. Kasperl und Seppl von Ottfried Preußler: Donnerstags müssen es Würstl mit Kraut sein und am Sonntag brauchen sie einen Pflaumenkuchen (Preußler [1962], Preußler [2006]).

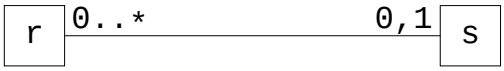


1:1-Beziehungen

1:1-Beziehungen kann man vollständig eliminieren, indem man einfach alle Attribute der beiden zugehörigen Entitytypen in *einer* Tabelle gemeinsam ablegt. Einen der beiden Primärschlüssel (hier z. B. `aid`) verwendet man als Primärschlüssel der gemeinsamen Tabelle. Die anderen Primärschlüsselattribute können komplett entfernt werden, wenn es sich um einen künstlichen Schlüssel handelt (wie hier `pid`), anderenfalls behält man sie bei und definiert sie als `unique(`<*Attribut 1*>`,...,`<*Attribut n*>`)`.

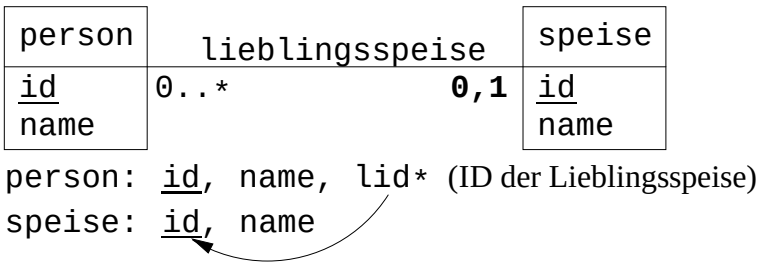
Optionale Beziehungen

0,1:n- bzw. n:0,1-Beziehung



Erweitere `r` um den anderen Fremdschlüssel (wie im Fall 1:n). Der Fremdschlüssel darf dabei auch den Wert `NULL` annehmen (im Gegensatz zum Fall 1:n).

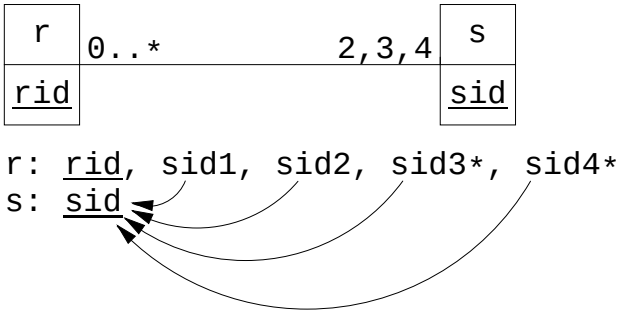
Im Beispiel mit der Lieblingsspeise müsste also in der Tabelle `person` das Attribut `lname` durch `lname*` ersetzt werden, wenn jede Person (an einem speziellen Tag in der Woche) eine Lieblingsspeise haben *kann*, aber nicht unbedingt haben muss.



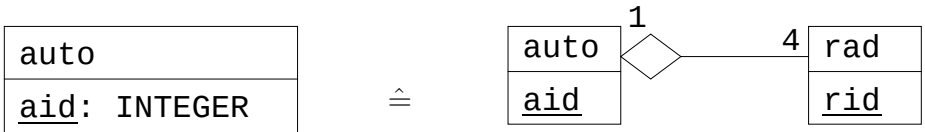
Es gibt Multiplizitäten, die weder beliebig groß sein können, noch gleich `1` oder `0,1` sind: z. B. `4` oder `0..2`. Die soeben beschriebene Optimierungsmethode lässt sich für diese Fälle verallgemeinern. Wenn bekannt ist, dass ein Element einer Klasse `r` zu mindestens *min* und höchstens *max* Elementen einer Klasse `s` Beziehungen hat, d. h., dass eine *min–max:n*-Beziehung vorliegt, dann kann man in die Relation `r` *max* Attribute vom Typ des Primärschlüssels von `s` einfügen und für jedes dieser Attribute eine Fremschlüsselbeziehung zu `s` definieren. (Sollte `s`

mehr als ein Primärschlüssel-Attribut haben, vervielfacht sich die Zahl der zusätzlichen Attribute in `r` entsprechend.) Von den *max* Attributen müssen *min* Attribute als `NOT NULL` deklariert werden, der Rest darf den Wert `NULL` annehmen.

Beispiel



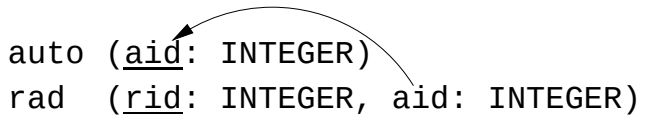
Beispiel: Auto



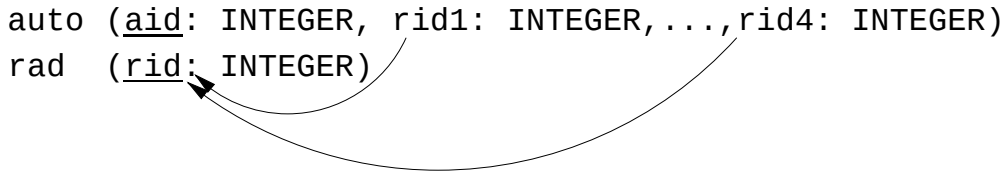
{Ein Auto hat 4 Räder}

Hier sind zwei Transformationen denkbar. Einerseits kann man die Multiplizität `4` als Spezialfall von `0..*` auffassen und entsprechend behandeln.

Dafür ergibt sich folgende Transformation:



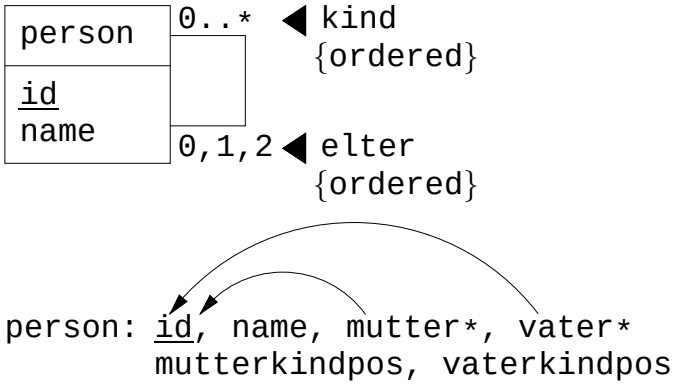
Man kann aber auch für jedes Rad einzeln ein Attribut in der Tabelle Auto angegeben. Da die Zahl der Räder bekannt und sehr klein ist, ist dies möglich. Eine 4er-Beziehung entspricht vier 1er-Beziehungen. Das heißt, man könnte auch folgende Transformation vornehmen.



Man beachte, dass im zweiten Fall die Reihenfolge der Räder durch die Attributreihenfolge festgelegt ist. Um dies auch für den ersten Fall zu erreichen, bräuchte man dort ein zusätzliches Attribut wie z. B. `radpos`.

Beispiel: Personen

Das Personen-Beispiel vom Ende des letzten Abschnitt 3.2.1 kann auf diese Weise optimiert werden.



Im Gegensatz zu dem gemäß „Schema F“ (Abschnitt 3.2.1) entwickelten Schema

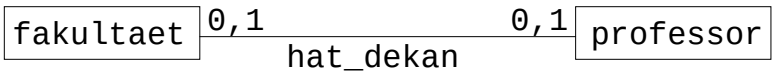
```
person: id, name
kind:   pid, kid, ppos, kpos
```

wird also nur eine Relation benötigt.

0,1:0,1-Beziehung

Erweitere `r` oder `s` um den anderen Fremdschlüssel. Der Fremdschlüssel darf auch den Wert `NULL` annehmen. Welche der beiden Relationen erweitert wird, hängt von allgemeinen Überlegungen ab, wie z. B. „Welche Relationen enthält weniger Elemente?“, „Bei welcher Relation wird seltener ein `NULL`-Wert gesetzt?“ etc.

Beispielsweise ist es im folgenden Fall



besser, die Relation `fakultaet` um einen Fremdschlüssel zu erweitern, da eine Fakultät zu fast jedem Zeitpunkt einen Dekan hat, aber nur sehr wenige Professoren Dekane einer Fakultät sind.

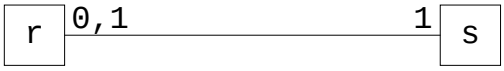
```
fakultaet: fid, dekan*
professor: pid
```

bessere Variante

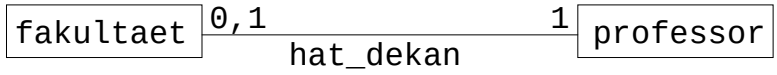
```
fakultaet: fid
professor: pid, dekan_von*
```

schlechtere Variante

0,1:1- bzw. 1:0,1-Beziehung



Erweitere `r` um den anderen Fremdschlüssel. Der Fremdschlüssel darf nicht `NULL` sein (da es zu jedem Tupel in `r` ein Tupel in `s` geben muss). Man beachte allerdings, dass ein `s`-Tupel auch ohne zugehöriges `r`-Tupel existieren kann.




```
fakultaet: fid, dekan        fakultaet: fid
professor: pid               professor: pid, dekan_von*
```

gute Variante — schlechte Variante (vermeiden)

Eine weitere Möglichkeit wäre, `s` um alle Attribute von `r` zu erweitern und für alle diese Attribute den Wert `NULL` zuzulassen. Der Vorteil wäre, dass man nur noch eine an Stelle von zwei Tabellen hätte.

```
professor: pid, ...,    fid*, ...*
           Prof.-Attribute  Fak.-Attribute
                            (NOT NULL nur beim Dekan)
```

Dieses Optinierung sollte man allerdings **vermeiden**, das sie drei Nachteile mit sich bringt.

Der erste Nachteil ist der Präzisionsverlust: Man kann nicht mehr zwischen `NULL`- und `NOT-NULL`-Attributen von `r` (im Beispiel ist dies `fakultaet`) unterscheiden.

Der zweite Nachteil ist, dass alle `s`-Tupel, die in keiner Beziehung zu `r` stehen (im Beispiel sind dies die Professoren, die nicht Dekan der Fakultät sind, also fast alle Professoren), trotzdem die `r`-Attribute enthalten und jeweils `NULL` als Attributewert eintragen müssen:

Tabelle **`professor`**

pid	**name** ...	**fid**	**fname** ...
1	'Scholz'	'I'	'Informatik'
2	'Klever'	NULL	NULL
3	'Kowarschick'	NULL	NULL
4	'Rist'	NULL	NULL
...			

Der dritte und größte Nachteil wäre jedoch, dass Beziegungen zum Entitytyp `r` nicht mehr von Beziehungen zum Entitytyp `s` unterschieden werden könnten. Das

heißt, Fremdschlüsselbeziehungen könnten fehlerhaft sein. Beispielsweise könnte ein Studiengang wie z. B. „Interaktive Systeme“, der einer Fakultät zugeordnet sein muss (Fremdschlüssel!), fälschlicherweise einem beliebigen Professor (z. B. „Kowarschick“) zugeordnet werden.

Vermeiden Sie diese Art der Optimierung!

Unidirektionale Beziehungen

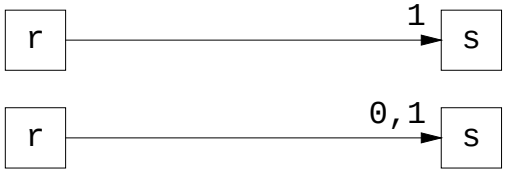
Wie Sie bereits wissen, wird die Unidirektionalität einer Beziehung vom „Schema F“ ignoriert.

Für die Optimierung gelten folgende Regeln:

Die unidirektionale Beziehung

kann nicht optimiert werden, wenn die Vielfachheit der anderen Richtung nicht bekannt ist. Hier wird gemäß „Schema F“ verfahren.

Die folgenden unidirektionalen Beziehungen

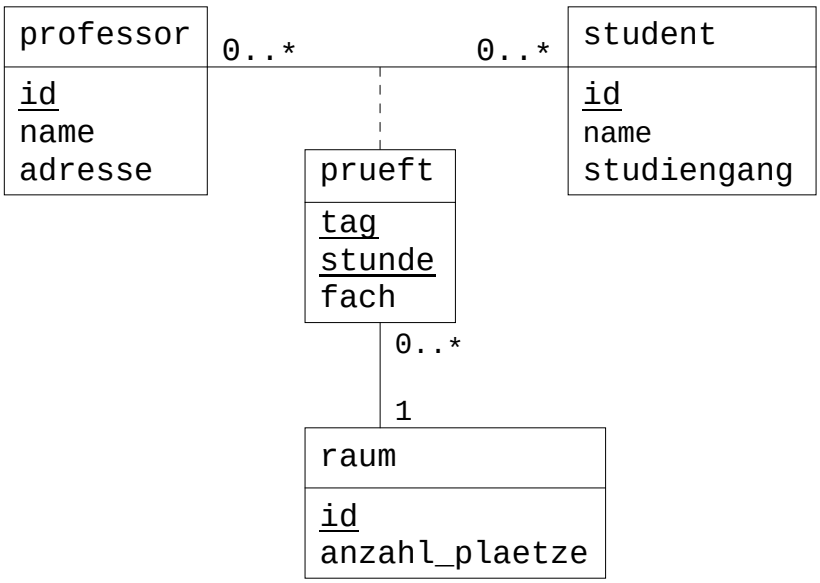


können dagegen wie normale n:1-Beziehungen optimiert werden. Da die Vielfachheit der anderen Richtung unbekannt ist, wird sie mit `0..*` gleichgesetzt.

Beziehungen zu Beziehungen

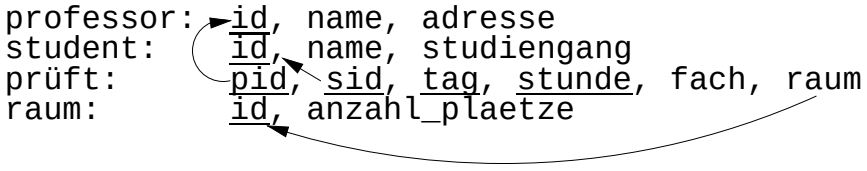
Beziehungen können nicht nur zwischen Klassen bestehen, sondern auch zwischen einer Klasse und einer Beziehung oder sogar zwischen zwei Beziehungen.

Beispiel



Jeder Prüfungsbeziehung zwischen einem Professor und einem Studenten ist genau ein Raum zugeordent. (Die ganze Prüfung kann selbstverständlich mehrere Räume belegen.)

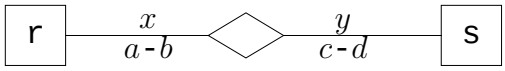
In diesem Fall wird nicht anders verfahren als bisher. Für das obige Beispiel erhält man folgendes Schema:



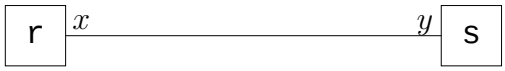
Wenn eine Beziehung nicht in einer eigenen Tabelle untergebracht, sondern in einer anderen Tabelle „versteckt“ wird, müssen natürlich auch Beziehungen zu dieser Beziehung als Beziehungen zu dieser anderen Tabelle realisiert werden. Dieser Fall kommt in der Praxis jedoch kaum vor.

ER → Rel-Matrix

Die verschiedenen Abbildungen einer ER-Beziehung auf ein Relationenschema

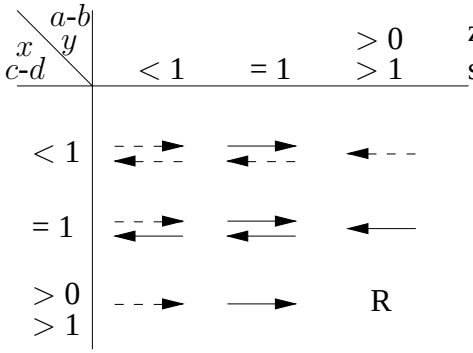


ER



UML

kann man mit Hilfe einer Matrix zusammenfassen:

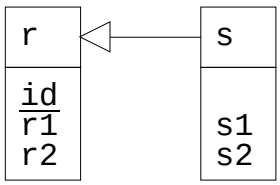


zu s-Tupeln gibt es so viele r-Tupel

- - ► foreign key in r references s
◄ - - foreign key in s references r
——► foreign key in r references s, NOT NULL
◄—— foreign key in s references r, NOT NULL

R Beziehung als eigene Relation (+ Zusatzbedingungen)

3.2.3 Vererbung: IS-A-Beziehungen



IS-A-Beziehungen (s isa r) können entweder durch Hinzunahme aller Attribute (1. Möglichkeit) oder nur der Primärschlüsselattribute (2. Möglichkeit) von r zu s realisiert werden.

1. Möglichkeit

```
r: id,r1,r2
s: id,r1,r2,s1,s2 (s.id NOT IN (SELECT r.id FROM r))
```

2. Möglichkeit
   `r: id,r1,r2`
   `s: id,s1,s2` (FK: `id`→`r:id`,
   d. h. `s.id IN (SELECT r.id FROM r)`)

Im ersten Fall spart man eine Join-Operation, um in `s` auch an die von `r` geerbten Nichtschlüsselattribute zu gelangen. Im zweiten Fall ist die Auflistung aller `r`-Objekte einfacher.

1. Möglichkeit
   Alle Objekte der Art `r`:
   ```
   SELECT id, r1, r2 FROM r
   UNION
   SELECT id, r1, r2 FROM s
   ```
   Alle Informationen über `s`-Objekte:
   ```
   SELECT id, r1, r2, s1, s2 FROM s
   ```
2. Möglichkeit
   Alle Objekte der Art `r`:
   ```
   SELECT id, r1, r2 FROM r
   ```
   Alle Informationen über `s`-Objekte:
   ```
   SELECT id, r1, r2, s1, s2 FROM r JOIN s USING (id)
   ```

Eine dritte Realisierungsmöglichkeit ist es, für eine gesamte IS-A-Hierarchie nur eine einzige Tabelle zu definieren:

```
r: id, r1, r2, s1*, s2*, typ (ENUM)
```

Diese enthält die Attribute der Basisklasse sowie aller Unterklassen. Nichtdefinierte Attributwerte werden mit `NULL` initialisiert. Da hierdurch auch `NOT-NULL`-Attribute den Wert `NULL` annehmen können, kann es in der Datenbank leicht zu Inkonsistenzen kommen. Man sollte also i. Allg. auf diese Realisierungsmöglichkeit verzichten.

Seit SQL:1999 unterstützt SQL einfache Attributvererbung auch direkt:

```
CREATE TABLE r (…);
CREATE TABLE s UNDER r (…);
```

Die Untertabelle `s` erbt alle Attribute und Integritätsbedingungen, insbesondere den Primärschlüssel. `s` kann weitere Attribute und Integritätsbedingungen (mit Ausnahme eines anderen Primärschlüssels) definieren.

Mit `SELECT * FROM r` werden alle Tupel von `r` einschließlich der Tupel von `s` (reduziert auf die geerbten Attribute von `r`) und aller anderen Unterklassen selektiert.

Mit `SELECT * FROM ONLY(r)` werden dagegen ausschließlich die eigenen Tupel von `r`, nicht aber die Tupel von Unterklassen von `r` (wie z. B. `s`) selektiert.

Diese Art der Vererbung hat den Vorteil, dass sowohl der Zugriff auf alle `r`-Objekte, als auch der Zugriff auf alle Attribute eines `s`-Objektes ohne zusätzlichen `JOIN`- oder `UNION`-Operator möglich sind. Die Nachteile sind: Mehrfachvererbung wird nicht unterstützt und standard-konforme Vererbung bieten bislang nicht sonderlich viele DMBS.

PostgreSQL unterstützt Vererbung, allerdings nicht standard-konform mit eigener Syntax und Semantik:

```
CREATE TABLE s (…) INHERITS (r)
```

Hier ist zwar Mehrfachvererbung erlaubt, aber aufgrund von Implementierungsschwächen sollte man die PostgreSQL-Vererbung derzeit besser nicht verwenden.

Beachten Sie, dass in allen vorgestellten Fällen auch Beziehungen von Oberklassen zu Unterklassen vererbt werden, so wie es sein sollte. Das gilt selbst dann, wenn in die Tabelle der Oberklasse zusätzliche „Beziehungsattribute“ existieren.

3.2.4 Beispiel: HSA-DB

Aufgabe

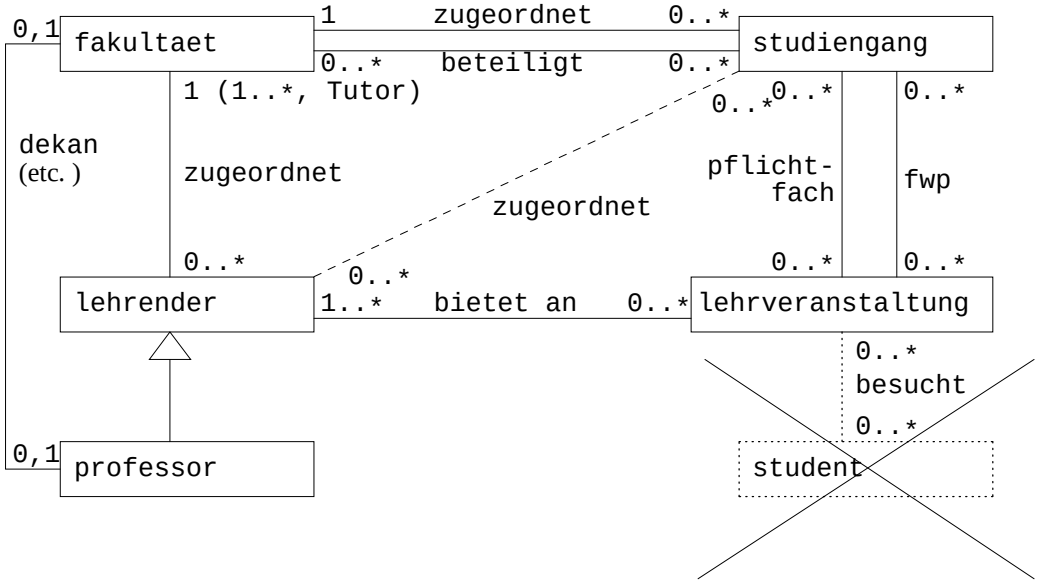
Nehmen Sie an, dass die Web-Seiten der Hochschule Augsburg künftig dynamisch erzeugt werden. Die Inhalte der Seiten werden dazu in einer Datenbank abgelegt.

Entwickeln Sie ein geeignetes Datenmodell für diese Datenbank. Da es sich um eine Übungsaufgabe handelt, soll das Datenmodell nur folgende Informationen zu enthalten: Fakultäten, Studiengänge, Lehrende, Professoren (als spezielle Lehrende), Lehrveranstaltungen (Lehrinhalte, kein Studenplan!) und Beziehungen zwischen den entsprechenden Objekten.

1. Sehen Sie sich die derzeitigen HTML-Seiten an und notieren Sie, welche Arten von Inhalten Sie in der Datenbank speichern müssen.
2. Überlegen Sie sich, welche Klassen (Entity-Typen) zum Speichern dieser Informationen benötigt werden. Geben Sie noch keine Attribute dieser Klassen an.
3. Welche Beziehungen bestehen zwischen Ihren Klassen? Berücksichtigen Sie nicht nur offensichtliche Beziehungen, wie z.B. „Lehrender hält Lehrveranstaltung“, sondern auch Informationen wie „Professor ist Dekan, Studiengangsleiter etc.“. Bedenken Sie, dass jede Beziehung, die Sie in Ihr Modell aufnehmen, später auch in der Datenbank in Form von Daten gespeichert und gepflegt werden muss.
4. Zeichnen Sie ein geeignetes UML-Diagramm.
5. Geben Sie für jede Ihrer Klassen wichtige Attribute an. Überlegen Sie, ob es auch für bestimmte Beziehungen wichtige Attribute geben könnte.
6. Transformieren Sie das Diagramm in ein relationales Datenbankschema.

Ein mögliches Datenmodell

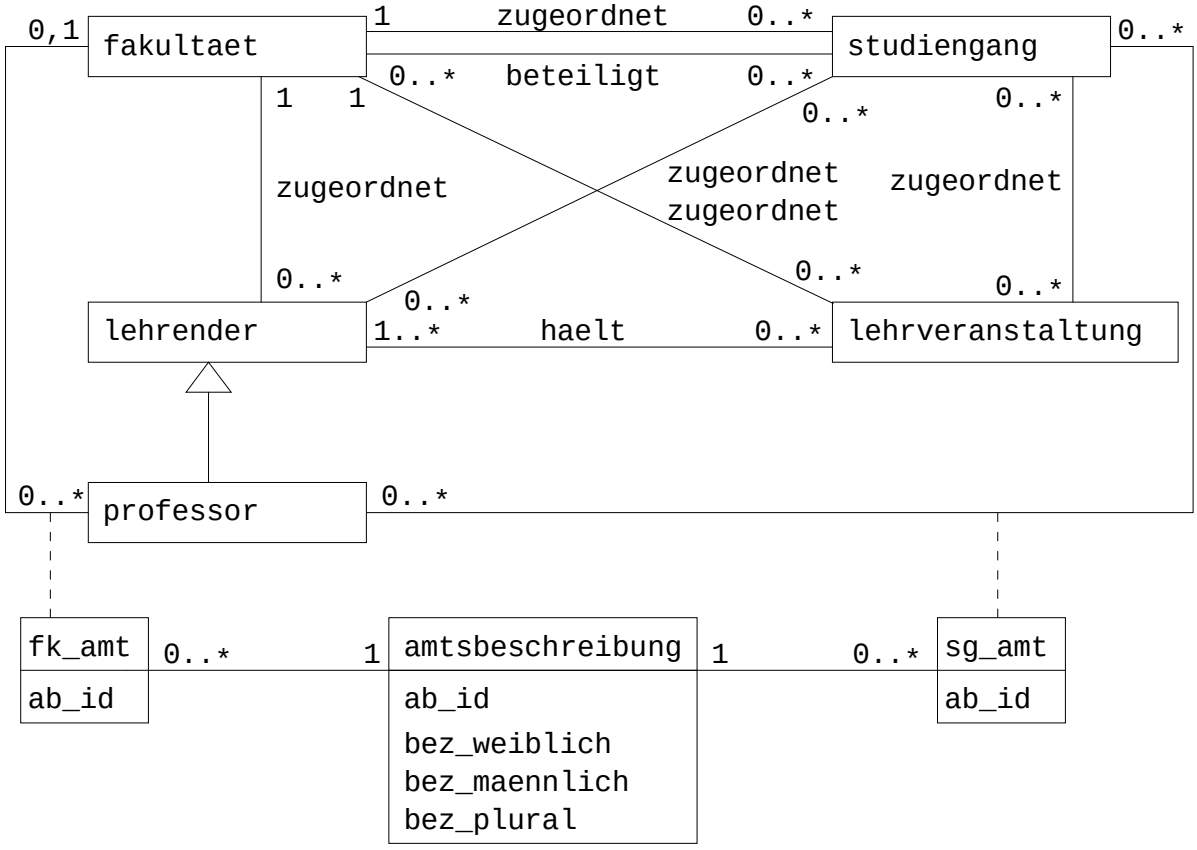
Beim folgenden Datenmodell handelt es sich nur um eine mögliche Lösung.



Dieses Modell hat z. B. den Nachteil, dass jedes Amt (Dekan, Studiendekan, . . .) durch eine eigene Beziehung dargestellt wird. Das heißt, für jedes neue Amt müsste das Modell und damit das Schema geändert werden. Besser wäre es daher, eine Beziehungsklasse `amt` einzuführen. Ein Professor kann jedes Fakultätsamt (wie z. B. Dekan oder Frauenbeauftragte) höchstens an einer Fakultät ausüben. Studiengangsämter (wie z. B. Studienganskoordinator oder PK-Vorsitzender) können von einem Professor dagegen an beliebig vielen Studiengängen ausgeübt werden.

Auf den Entitytyp `student` sollte i. Allg. verzichtet werden. Der Pflegeaufwand steht in keinem Verhältmis zu Informationsgewinn. Sollte allerdings eine Personalisierungskomponente im Web-Auftritt vorgesehen werden, so würde sich das Blatt wenden. In diesem Fall könnte jeder Student seine eigenen Daten verwalten.

Verbessertes Datenmodell

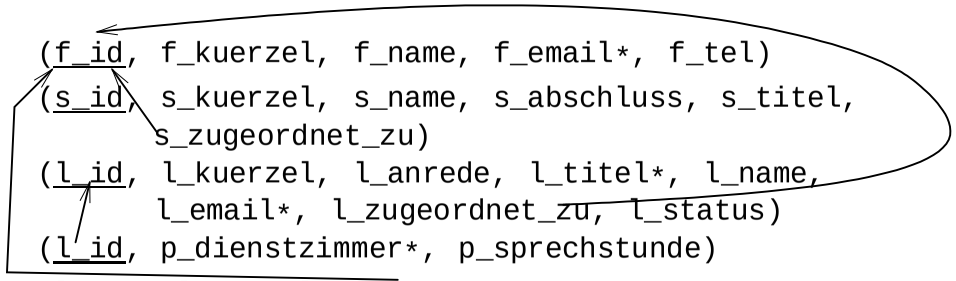


Attribute

Auf die Angabe zahlreicher Attribute wird hier aus Gründen der Übersichtlichkeit verzichtet. Ein paar wenige zusätzliche Attribute (neben den jeweiligen Identifikatoren) sind im nachfolgenden Relationenschema angegeben.

Relationenschema

Entity-Relationen



```
fakultaet           (f_id, f_kuerzel, f_name, f_email*, f_tel)
studiengang         (s_id, s_kuerzel, s_name, s_abschluss, s_titel,
                    s_zugeordnet_zu)
lehrender           (l_id, l_kuerzel, l_anrede, l_titel*, l_name,
                    l_email*, l_zugeordnet_zu, l_status)
professor           (l_id, p_dienstzimmer*, p_sprechstunde)
lehrveranstaltung   (lv_id, lv_titel, lv_zugeordnet_zu)
amtsbeschreibung    (ab_id, ab_bez_weiblich, ab_bez_maennlich, ab_bez_plural)
```

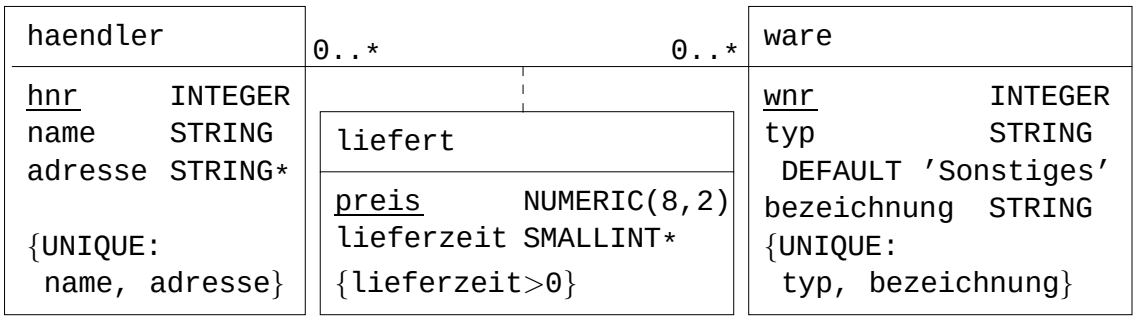
Relationship-Relationen

```
amt_fk              (f_id, l_id, ab_id)
amt_stg             (s_id, l_id, ab_id)
fk_beteiligt_stg    (f_id, s_id)
lehr_zugeordnet_stg (l_id, s_id)
lehr_haelt_lv       (l_id, lv_id)
stg_zugeordnet_lv   (s_id, lv_id)
```

Fremdschlüssel verweisen auf die jeweils gleichnamigen Attribute der obigen Entity-Relationen (l_id verweist i. Allg. auf lehrender; Ausnahme: die Tabellen amt_fk und amt_stg, da laut UML-Diagramm nur Professoren Ämter ausüben können).

3.3 Die Relationale Algebra und SQL

Im Folgenden verwende ich die klassische Lieferanten-DB als „running example“:



Das zugehörige Relationenschema sieht wie folgt aus:

```
haendler
  hnr          INTEGER     NOT NULL
  name         VARCHAR(30) NOT NULL
  adresse      VARCHAR(50)
  PRIMARY KEY (hnr),
  UNIQUE (name, adresse)

ware
  wnr          INTEGER     NOT NULL
  typ          VARCHAR(30) NOT NULL DEFAULT 'Sonstiges'
  bezeichnung VARCHAR(50) NOT NULL
  PRIMARY KEY key (wnr),
  UNIQUE (typ, bezeichnung)

liefert
  hnr          INTEGER     NOT NULL
  wnr          INTEGER     NOT NULL
  preis        NUMERIC(8,2)NOT NULL
  lieferzeit   SMALLINT    CHECK (lieferzeit >= 0)

  PRIMARY KEY (hnr, wnr, preis)
  FOREIGN KEY (hnr) REFERENCES haendler
  FOREIGN KEY (wnr) REFERENCES ware
```

(Online: `http://mmdb.hs-augsburg.de/beispiel/haendler/`)

Beispielsdaten

In den meisten der nachfolgenden Beispiele gehe ich davon aus, dass in der zugehörigen Datenbank folgende Daten enthalten sind. Hinsichtlich der Warenbezeichnungen sind sie etwas veraltet. :-)

haendler

hnr	name	adresse
1	'Maier'	'Königsbrunn'
2	'Müller'	'Königsbrunn'
3	'Maier'	'Augsburg'
4	'Huber'	NULL

ware

wnr	typ	bezeichnung
1	'CPU'	'Pentium IV 3,8'
2	'CPU'	'Celeron 2,6'
3	'CPU'	'Athlon XP 3000+'
4	'RAM'	'SDRAM 1GB'
5	'Sonstiges'	'Eieruhr'
⋮		

liefert

hnr	wnr	preis	lieferzeit
1	1	200.00	1
1	1	194.00	6
1	2	100.00	NULL
1	3	150.00	7
1	4	10.00	1
1	5	5.00	1
2	1	190.00	NULL
2	3	170.00	4
3	1	195.00	2

Definition

Unter einer *relationalen Algebra* versteht man eine Menge von *relationalen Operationen*, die jeweils eine oder mehrere Relationen (Tabellen) wieder auf Relationen abbilden.

Die Algebra-Eigenschaft, dass die Ergebnisse relationaler Operationen immer Relationen sind, wird auch als *Abgeschlossenheit* bezeichnet. Die Abgeschlossenheit hat zur Folge, dass nacheinander beliebig viele relationale Operationen auf eine Tabelle angewendet werden können.

3.3.1 Die Identität

Die einfachste relationale Operation ist die Identität $id : R \to R$, die jede Relation r als Ganzes zurückliefert:

$id(r) = r$ (Damit gilt auch $id(id(r)) = r$ etc.)

In SQL geschieht dies mit Hilfe eines speziellen `SELECT`-Befehls:

```
SELECT DISTINCT * FROM <Tabelle>
```

Auf den Zusatz `DISTINCT` kann man verzichten, wenn die Tabelle mit Sicherheit keine Duplikate enthält.

In SQL2 gibt es auch noch den Befehl `TABLE`; dieser wurde jedoch in SQL3 aufgegeben.

```
TABLE <Tabelle>
```

Beispiele

```
SELECT * FROM haendler
TABLE haendler (SQL2)
SELECT DISTINCT *
  FROM (SELECT typ FROM ware) AS typ_ware
```

Man beachte, dass der `SELECT`-Befehl Duplikate nur dann automatisch entfernt, wenn man `SELECT DISTINCT` schreibt. Anderenfalls (d. h. bei `SELECT` $\hat{=}$ `SELECT ALL`) kann das Ergebnis Duplikate enthalten. Das heißt, SQL realisiert eine *Multimengen-Semantik*. Im Falle von Basis-Relationen (wie z. B. `haendler`) ist die Angabe von `DISTINCT` allerdings nicht notwendig, da derartige Relationen immer ohne Duplikate gespeichert werden. Nur berechnete Relationen können also Duplikate enthalten.

3.3.2 Die Projektion

Die *Projektion* dient dazu, bestimmte *Spalten* aus einer beliebigen (d. h. gespeicherten oder berechneten Tabelle) zu selektieren.

Wenn $r(a_1, \ldots, a_n)$ eine Relation ist und $\{c_1, \ldots, c_k\}$ eine Teilmenge der Attribute $\{a_1, \ldots, a_n\}$, dann ist $\pi_{c_1,\ldots,c_k}(r)$ diejenige Tabelle, die aus r entsteht, wenn man alle übrigen Spalten aus r entfernt. Man beachte: Dabei entstehende Duplikate müssen ebenfalls entfernt werden.

Beispiel

ware

wnr	typ	bezeichnung
1	'CPU'	'Pentium IV 3,8'
2	'CPU'	'Celeron 2,6'
3	'CPU'	'Athlon XP 3800+'
4	'RAM'	'1GB'
5	'Sonstiges'	'Eieruhr'

$\pi_{\texttt{typ}}(\texttt{ware})$

typ
'CPU'
'RAM'
'Sonstiges'

Man kann die obige Definition der Projektion sogar noch erweitern: Jede Spalte der Ergebnisrelation wird mit Hilfe jeweils einer Funktion aus den Attributwerten von beliebig vielen (d. h. 0, 1 oder mehreren) Attributen der Ursprungsrelation *berechnet*: $\pi_{a_1+a_3,5,a_2*(a_4+a_7)}(r)$

Beispiele

Ausgabe der Nettopreise (ohne MwSt):

$\pi_{\texttt{hnr,wnr,round(preis/1.19,2) AS nettopreis,lieferzeit+1}}(\texttt{liefert})$

hnr	wnr	nettopreis	column4
1	1	168.06	2
1	1	161.34	7
1	2	84.03	NULL
1	3	126.05	8
2	1	159.66	NULL
2	3	142.85	5
3	1	163.86	3

Bezeichnung der Händler:

$\pi_{\texttt{name || ' in ' || adresse}}(\texttt{haendler})$

column1
'Maier in Königsbrunn'
'Müller in Königsbrunn'
'Maier in Augsburg'
NULL

Man beachte, dass die Verknüpfung (Konkatenation) zweier Zeichenketten in SQL mit dem Verknüpfungsoperator `||` (und nicht wie sonst mit `+`) erfolgt.

`NULL`-Werte

Wird der Wert `NULL` mit anderen Werten auf irgendeine Weise verknüpft, ist das Ergebnis stets `NULL`, wie z. B. `'Huber'||' in '||NULL = NULL`.

SQL-Syntax

In SQL wird die Projektionsliste direkt nach dem Schlüsselwort `SELECT` angegeben. Die drei zuvor angegebenen Beispiele schreibt man als SQL-Befehle wie folgt:

```
SELECT DISTINCT typ FROM ware;
SELECT hnr, wnr, round(preis/1.19,2),lieferzeit+1
  FROM liefert;
SELECT name || 'in ' || adresse FROM haendler;
SELECT * FROM haendler;                -- * ≙ alle Attribute
```

Störend am letzten Ergebnis ist, dass im Falle des Händlers `'Huber'` nur `NULL` ausgegeben wird. Dieses Problem kann in SQL mit zwei Arten von bedingten Anweisungen gelöst werden: mit der klassischen `CASE`-Anweisung und mit dem `COALESCE`-Operator.

```
SELECT CASE WHEN adresse IS NULL THEN name
            ELSE name || ' in ' || adresse
       END
FROM   liefert;

SELECT COALESCE(name || ' in ' || adresse, name)
FROM   liefert
```

Der `COALESCE`-Operator liefert den ersten Wert ungleich `NULL`, sofern einer exitiert. Anderenfalls ist das Ergebnis `NULL`.

Fehlerhafte Verwendung der `CASE`-Anweisung

Für jeden Händler soll zusätzlich ausgegeben werden, ob er in Königsbrunn wohnt oder nicht.

```
SELECT hnr, name,
       CASE WHEN adresse = 'Königsbrunn' THEN TRUE
                                         ELSE FALSE
       END AS haendler_wohnt_in_koenigsbrunn
FROM   haendler
```

Das Ergebnistabelle dieser Anfrage behauptet auch für Huber, dass dieser nicht in Königsbrunn wohnt, obwohl gar nicht bekannt ist, wo Huber wohnt (`adresse IS NULL`). Er könnte also auch in Könisgbrunn wohnen. Das Problem ist, dass SQL eine dreiwertige Logik unterstützt, die zwischen den Werten `TRUE`, `FALSE` und `UNKNOWN` unterscheidet (siehe Abschnitt 3.3.3). Korrekt müsste die Anfrage also folgendermaßen formuliert werden:

```
SELECT hnr, name,
       CASE WHEN adresse IS NULL         THEN NULL
            WHEN adresse = 'Königsbrunn' THEN TRUE
                                         ELSE FALSE
       END AS haendler_wohnt_in_koenigsbrunn
FROM   haendler
```

Dasselbe Resultat kann man aber auch viel einfacher und kürzer ohne CASE-Anweisung ermitteln:

```
SELECT hnr, name,
       adresse = 'Königsbrunn'
         AS haendler_wohnt_in_koenigsbrunn
FROM   haendler
```

Alias-Namen

Manchmal (z. B. bei Namensgleichheiten) ist es notwendig oder zumindest nützlich Attribute oder sogar ganze Tabellen temporär (d. h. innerhalb einer Anfrage) unzubenennen. Dies geschieht mit Hilfe von so genannten *Tupelbezeichnern* (*range variables*).

Jeder Relation in der FROM-Klausel ist implizit oder explizit ein derartiger Tupelbezeichner zugeordnet. Den Tupelbezeichner kann man sich als Variablennamen vorstellen, der der Reihe nach alle Tupel der zugehörigen Relation durchläuft. In SQL ist allerdings eher die Auffassung üblich, dass der Tupelbezeichner lediglich ein *Alias-Name* für eine Relation ist.

```
SELECT DISTINCT w.typ FROM ware w;
SELECT w.* FROM ware AS w,
SELECT l.hnr l.wnr, l.lieferzeit+1 FROM liefert l;
```

Man kann zusätzlich die Attributnamen umbenennen:

```
SELECT w.t FROM ware AS w(nr, t, bez);
SELECT w.t FROM ware w(nr, t, bez);
```

Das Schlüsselwort AS ist optional und kann daher weggelassen werden.

Defaultmäßig wird der Relationenname selbst als Tupelbezeichner verwendet.

```
  SELECT typ      FROM ware
≡ SELECT ware.typ FROM ware ware
≡ SELECT ware.typ FROM ware
```

Wenn die Attributnamen der Projektionsliste völlig eindeutig sind (wie in unserem bisherigen Beispielen), kann man auf den Tupelbezeichner also auch verzichten.

3.3.3 Die Selektion

Während die *Projektion* dazu dient, bestimmte *Spalten* einer Tabelle zu selektieren (und weiterzuverarbeiten), dient die *Selektion* dazu, bestimmte *Zeilen* einer Tabelle zu *selektieren*. Bei der Selektion wird für jedes Tupel ein boolesches Prädikat (welches i. Allg. von den Attributen der zugehörigen Tabelle abhängt) ausgewertet.

Liefert das Prädikat den Wert `TRUE`, ist das Tupel im entsprechenden Ergebnis enthalten, sonst nicht.

$\sigma_{\texttt{typ='CPU'}}(\texttt{ware})$

```
SELECT * FROM ware WHERE typ = 'CPU'
```

wnr	typ	bezeichnung
1	'CPU'	'Pentium IV 3,8'
2	'CPU'	'Celeron 2,6'
3	'CPU'	'Athlon XP 3000+'

$\sigma_{\texttt{typ='CPU'}\wedge\ \texttt{bezeichnung LIKE 'Athlon\%'}}(\texttt{ware})$

```
SELECT * FROM ware
WHERE typ = 'CPU' AND bezeichnung LIKE 'Athlon%'
```

wnr	typ	bezeichnung
3	'CPU'	'Athlon XP 3000+'

SQL unterstützt folgende booleschen Operatoren, wobei allerdings i. Allg. eine 3-wertige Logik (`TRUE, FALSE, UNKNOWN`) zum Einsatz kommt:

=, <=, >, >=, <> und `[NOT] BETWEEN ... AND ...`,
`IS DISTINCT FROM` (2-wertige Logik) im Gegensatz zu <>,
`OR`, `AND`, `NOT`,
`IS [NOT] TRUE/FALSE/UNKNOWN/NULL`,
`LIKE` (String-Vergleich; _ $\hat{=}$ beliebiges Zeichen, % $\hat{=}$ beliebige Zeichenkette),
`SIMILAR TO` (Vergleich mit regulären Ausdrücken),
<*Zeitintervall*> `OVERLAPS` <*Zeitintervall*>

Daneben werden auch noch Operatoren angeboten, die auf ganzen Tabellen arbeiten: `[NOT] IN`, `MATCH`, `ALL`, `ANY`, `SOME`, `EXISTS`, `UNIQUE`.

3-wertige Logik

Da Tabellen i. Allg. auch den Wert `NULL` enthalten können, realisiert SQL eine so genannte *3-wertige Logik* (three-valued logic).

In dieser Logik gibt es die drei Wahrheitswerte `TRUE` (`t`), `FALSE` (`f`) und `UNKNOWN` (`u`) mit folgenden Wahrheits-Matrizen:

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

Außerdem müssen Arithmetikoperatoren, Vergleichsoperatoren etc. mit `NULL` umgehen können, wobei `NULL` als *unbekannter* und nicht als *undefinierter* Wert interpretiert wird:

`5+NULL=NULL`, `5>NULL` ↔ `UNKNOWN`, `NULL=NULL` ↔ `UNKNOWN` etc.

Anmerkung

Der Wert `NULL` darf laut SQL-Standard nicht in Termen verwendet werden (Ausnahme ... `IS NULL`, ... `IS NOT NULL`). Die Terme `5+NULL` und `NULL=NULL` sind also laut SQL-Standard unzulässig. Es ist aber zulässig `5+c` und `c=c` zu schreiben, auch wenn das Attribut `c` den Wert `NULL` annehmen kann (siehe nachfolgendes Beispiel).

Beispiel

Es gebe drei Attribute `a=3`, `b=4` und `c IS NULL`.

Dann gilt:

```
a>b AND b>c ↔ FALSE
a>b OR  b>c ↔ UNKNOWN
a<b OR  b<c ↔ TRUE
NOT(a=c)    ↔ UNKNOWN
c=c         ↔ UNKNOWN (und nicht etwa TRUE!!!)
```

Achtung

Der Test, ob ein Wert gleich `NULL` ist, muss mit `IS NULL` erfolgen:

```
c = NULL      ↔ UNKNOWN
c IS NULL     ↔ TRUE
c IS NOT NULL ↔ FALSE
```

Beachten Sie, dass `NOT(c=NULL)` ungleich `c IS NOT NULL` ist:

```
NOT(c=NULL)   ↔ UNKNOWN
c IS NOT NULL ↔ FALSE
```

Der Vergleich `c=NULL` ist laut SQL-Standard deswegen, wie bereits gesagt, gar nicht zulässig. Verwenden Sie daher immer den Test `c IS NULL`, auch wenn das DBMS den Test `c=NULL` entgegen den Vorgaben der Standards doch unterstützt.

Achtung

`x IS FALSE` ist nicht das Synonym von `x IS NOT TRUE`.

`x IS NOT TRUE` ist gleichwertig `x IS FALSE OR x IS UNKNOWN`.

`SELECT`-Anweisung

Eine `SELECT`-Anweisung

```
SELECT * FROM r WHERE <Prädikat>
```

liefert ein `r`-Tupel nur dann als Ergebnis, wenn das <*Prädikat*> den Wert `TRUE` liefert. Liefert es `FALSE` oder `UNKNOWN` wird das Tupel entfernt.

Wenn man ausschließlich mit Attributen arbeitet, deren Domänen als `NOT NULL` deklariert wurden, reduziert sich die dreiwertige Logik von SQL auf die klassische zweiwertige Logik.

3.3.4 Vereinigung, Durchschnitt, Differenz

Relationen sind Mengen. Es können daher die Vereinigung, der Durchschnitt oder die Differenz zweier Relationen gebildet werden. Dabei muss allerdings darauf geachtet werden, dass die zugehörigen Relationen dieselben Attributnamen besitzen und die Domänen „kompatibel“ sind.

$r(a_1 : D_1, \ldots, a_n : D_n) \cup s(a_1 : D_1, \ldots, a_n : D_n)$
$r(a_1 : D_1, \ldots, a_n : D_n) \cap s(a_1 : D_1, \ldots, a_n : D_n)$
$r(a_1 : D_1, \ldots, a_n : D_n) \backslash s(a_1 : D_1, \ldots, a_n : D_n)$

In SQL gibt es zu diesem Zweck die Operationen `UNION` (mit Duplikateliminination), `UNION ALL` (ohne Duplikatelimination), `INTERSECT`, `INTERSECT ALL`, `EXCEPT` und `EXCEPT ALL`.

Achtung: Bei `SELECT`-Klauseln gilt `ALL` als Default, bei Mengenoperationen `DISTINCT`:

```
SELECT    = SELECT ALL
UNION     = UNION DISTINCT
INTERSECT = INTERSECT DISTINCT
EXCEPT    = EXCEPT DISTINCT
```

Anmerkung

In TransBase müssen Sie `DIFF` anstelle von `EXCEPT` schreiben.
In Oracle heißt es `MINUS` anstelle von `EXCEPT`.

Beispiele

Alle CPUs und alle Speicherbaustein:

```
(SELECT * FROM ware WHERE typ='CPU')
UNION
(SELECT * FROM ware WHERE typ='RAM')
```

Alle Händler, deren Name *nicht* mit „`M`“ anfängt.

```
(SELECT * FROM haendler)
EXCEPT
(SELECT * FROM haendler WHERE name LIKE 'M%')
```

Daneben gibt es noch die Möglichkeit, nur bestimmte *gemeinsame* Attribute zweier Relationen zu berücksichtigen und die anderen Spalten zu eliminieren.

```
(SELECT * FROM r) UNION ALL CORRESPONDING BY (x,y)
(SELECT * FROM s)
=
(SELECT x,y FROM r) UNION ALL (SELECT x,y FROM s)
```

Wenn `BY (x,y)` weggelassen wird, werden *alle* gemeinsamen Attribute von `r` und `s` verwendet werden.

3.3.5 Kartesisches Produkt

Zwei Relationen man mit Hilfe des kartesischen Produktes ($\times$) verknüpfen.

`haendler` $\times$ `liefert`

hnr	name	adresse	hnr	wnr	preis	lieferzeit
1	'Maier'	'Königsbrunn'	1	1	200.00	1
1	'Maier'	'Königsbrunn'	1	1	194.00	6
1	'Maier'	'Königsbrunn'	1	2	100.00	NULL
1	'Maier'	'Königsbrunn'	1	3	150.00	7
1	'Maier'	'Königsbrunn'	2	1	190.00	NULL
1	'Maier'	'Königsbrunn'	2	3	170.00	4
1	'Maier'	'Königsbrunn'	3	1	195.00	2
2	'Müller'	'Königsbrunn'	1	1	200.00	1
2	'Müller'	'Königsbrunn'	1	1	194.00	6
2	'Müller'	'Königsbrunn'	1	2	100.00	NULL
2	'Müller'	'Königsbrunn'	1	3	150.00	7
2	'Müller'	'Königsbrunn'	2	1	190.00	NULL
2	'Müller'	'Königsbrunn'	2	3	170.00	4
2	'Müller'	'Königsbrunn'	3	1	195.00	2
3	'Maier'	'Augsburg'	1	1	200.00	1
3	'Maier'	'Augsburg'	1	1	194.00	6
3	'Maier'	'Augsburg'	1	2	100.00	NULL
3	'Maier'	'Augsburg'	1	3	150.00	7
3	'Maier'	'Augsburg'	2	1	190.00	NULL
3	'Maier'	'Augsburg'	2	3	170.00	4
3	'Maier'	'Augsburg'	3	1	195.00	2
4	'Huber'	NULL	1	1	200.00	1
4	'Huber'	NULL	1	1	194.00	6
4	'Huber'	NULL	1	2	100.00	NULL
4	'Huber'	NULL	1	3	150.00	7
4	'Huber'	NULL	2	1	190.00	NULL
4	'Huber'	NULL	2	3	170.00	4
4	'Huber'	NULL	3	1	195.00	2

Mit Hilfe des kartesischen Produktes, der Selektion und der Projektion kann nun z. B. bestimmt werden, wer irgendwelche Waren innerhalb von einem Tag liefern kann.

$\pi_{\texttt{h.hnr,h.name,h.adresse}}$
$(\sigma_{\texttt{h.hnr=l.hnr AND l.lieferzeit=1}}(\texttt{haendler h} \times \texttt{liefert l}))$

Ergebnis der Bedingung `l.lieferzeit=1`:

hnr	name	adresse	hnr	wnr	preis	lieferzeit
1	'Maier'	'Königsbrunn'	1	1	200.00	1
2	'Müller'	'Königsbrunn'	1	1	200.00	1
3	'Maier'	'Augsburg'	1	1	200.00	1
4	'Huber'	NULL	1	1	200.00	1

Ergebnis der Bedingung `h.hnr=l.hnr`:

hnr	name	adresse	hnr	wnr	preis	lieferzeit
1	'Maier'	'Königsbrunn'	1	1	200.00	1

Ergebnis der Projektion $\pi_{\texttt{h.hnr,h.name,h.adresse}}$:

hnr	name	adresse
1	'Maier'	'Königsbrunn'

In SQL wird das kartesische Produkt beliebig vieler Relationen einfach dadurch gebildet, dass mehrere Relationen (evtl. zusammen mit Tupelbezeichnern) in die `FROM`-Klausel geschrieben werden:

```
SELECT *
FROM   haendler h, liefert l
```

Aus dem kartesischen Produkt muss mit Hilfe von geeigneten Prädikaten das gewünschte Ergebnis selektiert werden. In der Projektionsliste müssen außerdem die gewünschten Attribute angegeben werden. Die vollständige Anfrage lautet für das obige Beispiel:

```
SELECT h.hnr, h.name, h.adresse
FROM   haendler h, liefert l
WHERE  h.hnr=l.hnr AND l.lieferzeit=1
```

Weitere Beispiele

Welche Waren liefert Maier aus Königsbrunn?

```
SELECT w.wnr, w.typ, w.bezeichnung
FROM   haendler h, liefert l, ware w
WHERE  h.hnr=l.hnr AND l.wnr=w.wnr AND
       h.name='Maier' AND
       h.adresse LIKE '%Königsbrunn%'
```

Wer liefert welche Waren billiger als Maier aus Königsbrunn?

```
SELECT h.hnr, h.name, h.adresse,
       w.wnr, w.typ, w.bezeichnung,
       l.preis, lm.preis
FROM   -- Ware von Maier (ohne Warenbezeichnung und -typ)
       haendler hm, liefert lm
       -- Ware der anderen Lieferanten
       haendler h, liefert l, ware w,
WHERE  -- Lieferant Maier
       hm.name='Maier' AND
       hm.adresse LIKE '%Königsbrunn%' AND
       -- liefert diese Waren:
       hm.hnr = lm.hnr AND
       -- Die anderen
       h.hnr <> hm.hnr AND
       --  liefern dies:
       h.hnr = l.hnr AND l.wnr = w.wnr AND
       -- Dieselbe Ware
       l.wnr = lm.wnr AND
       -- kostet weniger als bei Maier:
       l.preis < lm.preis
```

Man sieht, dass die Verwendung von Tupelbezeichnern unerlässlich ist, wenn man *ein und dieselbe Relation* mehrfach in der `FROM`-Klausel verwenden muss.

3.3.6 Join

Das kartesische Produkt zusammen mit der Selektion reicht – von einem rein mathematischen Standpunkt aus gesehen – aus, um die Verknüpfung von zwei oder mehreren Tabellen zu beschreiben. In der Realität bereiten kartesische Produkte allerdings große Probleme, da die Zwischenergebnisse sehr groß werden können. Die Anzahl der Elemente (*Mächtigkeit*) einer Menge M bezeichnet man mit $|M|$. Für Relationen $r_1, \ldots, r_n$ gilt: $|r_1 \times \ldots \times r_n| = |r_1| \cdot \ldots \cdot |r_n|$

Wenn wir z. B. 100 Händler, 1000 Waren und 50000 `liefert`-Beziehungen haben, besteht das kartesische Produkt der fünf Relationen des letzten SQL-Beispiels aus

$$100 \cdot 50000 \cdot 1000 \cdot 100 \cdot 50000 = 25 \cdot 10^{15}$$

Tupeln. In Worten: 25 Billiarden Tupel. Wenn unser Rechner 10^6 Tupel pro Sekunde abarbeiten kann, benötigt er also zur Beantwortung der gestellten Frage $25 \cdot 10^9$ Sekunden = $25 \cdot 31,71$ Jahre = 792,75 Jahre.

Aus diesem Grund wurden die so genannte Join-Operation eingeführt, die im wesentlichen kartesische Produkte gefolgt von speziellen Selektionen und Projektionen sind. Diese Spezialfälle können meist **sehr** viel effizienter bearbeitet werden.

Prädikat-Join

Die allgemeinste (und i. Allg. ineffizienteste) Join-Operation ist der *Prädikat-Join*. Er ist definiert als ein kartesisches Produkt gefolgt von einer Selektion. Diese Operation kann i. Allg. auch nicht effizienter als ein kartesisches Produkt gefolgt von einer Selektion ausgewertet werden.

$$r \bowtie_p s := \sigma_p(r \times s)$$

Dabei ist p ein Prädikat (wie $r.a < s.b \wedge s.c > 5$), welches sich auf beliebige Attribute von r und s abstützt.

Equijoin

Wenn das Prädikat lediglich je ein Attribut (oder je eine Menge von Attributen) der beiden Relationen r und s auf *Gleichheit* testet, spricht man von einem *Equijoin*:

$$r \bowtie_{r.a=s.b} s := \sigma_{r.a=s.b}(r \times s)$$

Equijoins sind sehr effizient realisierbar und kommen sehr häufig vor (siehe letztes SQL-Beispiel).

Wenn man zum Beispiel den so genannten Sort-Merge-Join-Algorithmus verwendet, beträgt die Equijoin-Komplexität:

$$O(|r| \cdot \log |r| + |s| \cdot \log |s| + |\text{<}\textit{Ergebnis}\text{>}|)$$

Falls das Ergebnis klein ist, kann die letzte Größe vernachlässigt werden. Das Ergebnis ist nur dann groß, wenn ein und derselbe Joinattribut-Wert in sehr vielen Tupeln vorkommt, wie z. B. `geschlecht: 'weiblich'` oder `'männlich'`. Falls jedoch jedes Tupel der kleineren Relation im Durchschnitt nur einen Joinpartner hat, ist das Ergebnis so groß wie die kleinere der beiden Relationen, d. h.:

$$O(|\text{<}\textit{Ergebnis}\text{>}|) = O(\min\{|r|, |s|\})$$

Dies ist wesentlich kleiner als:

$$O(|r| \cdot \log |r|) \text{ und } O(|s| \cdot \log |s|)$$

Insbesondere Schlüsselattribute kommen in einer Relation nicht mehrfach vor. Das heißt, wenn z. B. eine Relation r über ein Schlüsselattribut (von r) mit s gejoint wird, gilt $|\text{<}\textit{Ergebnis}\text{>}| \leq |s|$.

Beispiel

Unter der Annahme, dass die Ergebnisgröße bei der Bestimmung der Zeitkomplexität vernachlässigt werden kann, ergeben sich beispielsweise folgende Laufzeitunterschiede:

Angenommen $|r| = |s| = 10^6$; Verarbeitungsgeschwindigkeit: 10^6 Tupel/s

Nested-Loop-Join (Berechung des gesamten kartesischen Produktes)

```
FOR ALL rtup IN r
  FOR ALL stup IN s
    IF (<rtup und ntup erfüllen Joinbedingung>)
      <Füge rtup ∘ ntup ins Ergebnis ein.> // ∘ = Konkatenation
```

$|r| \cdot |s| = 10^{12}$
$\Rightarrow c_1 \cdot 10^6$ s $= c_1 \cdot 11,57$ Tage dauert die Berechnung des kartesisches Produktes
(wobei c_1 ein unbekannter, aber konstanter und nicht allzu großer Wert ist).

Sort-Merge-Join

```
SORT (r, <r-Join-Attribute>)
SORT (s, <s-Join-Attribute>)
WHILE (!empty(r) && !empty(s))
{
  <Kartesisches Produkt aus obersten r- und s-Tupel mit gleichem Joinpartner bilden.>
  <Kartesisches Produkt ins Ergebnis einfügen.>
  <Oberste r- und s-Tupel mit gleichem Joinpartner aus r und s entfernen.>
}
```

Wenn jedes Element höchstens einen (oder höchstens ein paar wenige) Joinpartner hat, fällt die Ermittelung des kartesischen Produktes in jedem Schritt nicht ins Gewicht. Dann kann man die Dauer der Join-Berechnung folgendermaßen abschätzen:

$|r| \cdot \log|r| + |s| \cdot \log|s| = 10^6 \cdot 6 + 10^6 \cdot 6 = 12 \cdot 10^6$ Operationen
$\Rightarrow 12 \cdot c_2$ Sekunden dauert die Berechnung des Sort-Merge-Joins
(wobei c_2 ein unbekannter, aber konstanter und nicht allzu großer Wert ist).

Aufgabe

Rechnen Sie aus, wie lange die Berechnung der Anfrage „Wer liefert welche Waren billiger als Maier aus Königsbrunn?“ dauert (bei gleicher DB-Größe wie zuvor).

Natural Join

Der *Natural Join* ist ein Equijoin über gleichbenannte Attribute der Relationen r und s, wobei gleichbenannte Attribute *nur einmal* (als *eine* Spalte) im Ergebnis aufgeführt werden. Wenn $r(a_1, \ldots, a_m, c_1, \ldots, c_k)$ und $s(b_1, \ldots, b_n, c_1, \ldots, c_k)$ dann:

$$r \bowtie s := \pi_{a_1,\ldots,a_m,c_1,\ldots,c_k,b_1,\ldots,b_n}(r \bowtie_{r.c_i=s.c_i} s)$$

Diese Join-Operation kann allerdings nicht effizienter als ein Equijoin berechnet werden (wenn man davon absieht, dass gleichbenannte Joinpartner nur einmal und nicht wie beim Equijoin zweimal im Ergebnis enthalten sind). Der wesentliche Vorteil des Natural Joins ist, dass die Syntax kompakter ist als beim Equijoin.

Semijoin

Der Semijoin ist ein Spezialfall des Natural Joins, der nur die Attribute von r als Ergebnis liefert:

$$r \ltimes s := \pi_{a_1,\ldots,a_m,c_1,\ldots,c_k}(r \bowtie s)$$

Der Semijoin ist eine spezielle Selektion: Aus r werden diejenigen Tupel ausgewählt, die in s einen Joinpartner haben.

Der Semijoin kann noch etwas effizienter als der entspechende Equijoin implementiert werden, da die Tupel von s nicht weiterverarbeitet werden müssen.

3.3.7 Explizite Joins in SQL

In SQL ist auch möglich, die eben diskutierten Joins in der `FROM`-Klausel explizit anzugeben. Es hängt allerdings vom DBMS ab, ob es sich an die vorgegebene Joins und die vorgegebene Join-Reihenfolge hält oder diese intern vom Optimierer ändern (= optimieren) lässt.

Kartesisches Produkt: (Cross Join)

Man kann das kartesische Produkt zweier Tabellen explizit erzeugen. Praktische Anwendungen für diese Operation gibt es aber eher selten (siehe aber Abschnitt 3.6.1). Folgendes Beispiel macht dagegen keinen rechten Sinn, ist aber dennoch erlaubt:

```
SELECT hnra, hnrb, a, p, z
FROM   (haendler CROSS JOIN liefert)
       AS hcl(hnra, l, a, hnrb, wnr, p, z)
```

Anmerkung: Bei der *expliziten Angabe* des zu verwendenden Join-Operators wird *häufig der Alias-Operator* `AS` verwendet, um das Join-Ergebnis zu benennen. Hier heißt das, dass sowohl in der `SELECT`-Klausel als auch in der `WHERE`-Klausel nur noch auf die (temporär erzeugte) Tabelle `hcl` zugegriffen werden kann, nicht aber auf die ursprünglichen Tabellen `haendler` und `liefert`.

Natural Join

Wenn die Join-Attribute zweier Relationen gleich benannt sind, kann man die Relation mit Hilfe der Natural-Join-Operation verknüpfen. Beachten Sie, dass jedes Join-Attribut nur jeweils einmal im Ergebnis erscheint.

```
SELECT name, preis
-- SELECT lieferant.name, lieferant.preis
FROM   (haendler NATURL JOIN liefert) AS lieferant;
```

Anmerkung: Wenn es keine gemeinsamen Attribute gibt, degeneriert der Natural Join zum kartesischen Produkt.

Diese Art des Natural Joins sollten Sie unbeding vermeiden. Grund: Nachträgliche Schemaänderungen (neue Attribute oder Attributnamen) können die Bedeutung derartiger Joins verändern. Verwenden Sie daher stets folgende Variante des Natural Joins, bei der die Join-Attribute mittels einer `USING`-Klausel *explizit* angegeben werden. Auf diese Weise kann man verhindern, dass *zufällig* gleichbenannte Attribute als Join-Partner verwendet werden.

```
SELECT name, preis
FROM   haendler JOIN liefert USING (hnr)
```

Diese Anweisung liefert als Ergebnis wiederum nur eine `hnr`-Spalte, da beide Spalten für jedes Tupel immer denselben Wert enthalten.

Prädikat-Join

Wenn man im zweiten der eben beschreibenen Join-Operationen eine `ON`-Klausel an Stelle der `USING`-Klausel verwendet, erhält man einen Prädikatjoin. In der `ON`-Klausel kann ein beliebiges Prädikat angegeben werden.

```
SELECT name, preis
FROM   (haendler JOIN liefert
        ON haendler.hnr = liefert.hnr
       ) AS lieferant;
```

Beachten Sie: Diese Anfrage liefert im Gegensatz zum Natural Join zwei `hnr`-Spalten als Ergebnis, da beliebige Join-Bedingungen angegeben werden können (z. B. `haendler.hnr < liefert.hnr`). Das heißt, in den beiden `hnr`-Spalten können pro Tupel durchaus unterschiedliche Werte stehen.

Equijoin und Semijoin

Für den Equijoin und den Semijoin gibt es keine speziellen SQL-Operatoren. Allerdings kann jeder Equijoin und jeder Semijoin als spezieller Prädikat-Join formuliert werden, indem in die `ON`-Klausel die entsprechenden Attribut-Vergleiche geschrieben werden. Beim obigen Prädikatjoin-Beispiel handelt es sich z. B. um einen auf diese Weise definierten Equijoin!

Outer Joins

Elemente einer Tabelle r, die keinen Joinpartner in einer Tabelle s finden, erscheinen nicht im Ergebnis von $r \bowtie s$. Manchmal benötigt man allerdings alle Elemente der Tabelle r, unabhängig davon, ob es einen passenden Joinpartner gibt oder nicht. Dies ist mit Hilfe der so genannten Outer-Joins möglich.

Der Left Outer Join sorgt dafür, dass die linke Tabelle vollständig erhalten bleibt, der Right Outer Join, dass die rechte Tabelle erhalten bleibt und der Full Outer Join, dass beide erhalten bleiben.

Beispielsweise ist der Natural Left Outer Join wie folgt definiert:

```
SELECT * FROM r NATURAL LEFT JOIN s
≡ (Pseudo-SQL)
( SELECT * FROM r NATURAL JOIN s )
UNION ALL
( SELECT *, NULL, ..., NULL FROM r
  WHERE * NOT IN (SELECT r.* FROM r NATURAL JOIN s) )
```

Beispiel

r

name	**ware**
'Maier'	'Lampen'
'Müller'	'Rechner'

s

name	**tel**
'Maier'	123
'Huber'	234

r ⋈ s

name	**ware**	**tel**
'Maier'	'Lampen'	123

r $\bowtie_{\text{left}}$ s

name	**ware**	**tel**
'Maier'	'Lampen'	123
'Müller'	'Rechner'	NULL

r $\bowtie_{\text{right}}$ s

name	**ware**	**tel**
'Maier'	'Lampen'	123
'Huber'	NULL	234

r $\bowtie_{\text{full}}$ s

name	**ware**	**tel**
'Maier'	'Lampen'	123
'Müller'	'Rechner'	NULL
'Huber'	NULL	234

Union Join

Daneben gibt es noch den Union Join, der überhaupt keine Join-Partner sucht, sondern nur `NULL`s als Join-Partner einsetzt.

```
r UNION JOIN s
name     ware      name    tel
'Maier'  'Lampen'  NULL    NULL
'Müller' 'Rechner' NULL    NULL
NULL     NULL      'Maier' 123
NULL     NULL      'Huber' 234
```

Für diese Join-Art ist mir bislang kein sinnvolles Anwendungsbeispiel eingefallen. Mir ist auch kein DBMS bekannt, das diese Join-Art implementiert hätte (vgl. Türker [2003], Seite 248). Beachten Sie außerdem, dass die obige Operation eigentlich sinnlos ist, da das Ergenis zwei gleich benannte Attribute enthalten würde. Damit wäre aber folgende Anfrage gar nicht mehr eindeutig beantwortbar:

```
SELECT name
FROM   r UNION JOIN s
```

Die Attribute müssen also geeignet umbenannt werden:

```
SELECT rname, sname
FROM   r AS r(rname, ware) UNION JOIN s AS s(sname, tel)
```

Auch wenn dieser Join nicht explizit existiert, so kann er doch sehr einfach mit einer echten `UNION`-Operation nachgebildet werden:

```
SELECT rname, ware, sname, tel
FROM   (SELECT name AS rname, ware,
               NULL AS sname, NULL AS tel
        FROM r
       )
       UNION
       (SELECT NULL AS rname, NULL AS ware,
               sname,         tel
        FROM s
       )
```

Zusammenfassung

SQL unterstützt folgende expliziten Joins:

```
JOIN          = INNER JOIN
LEFT JOIN     = LEFT OUTER JOIN
RIGHT JOIN    = RIGHT OUTER JOIN
FULL JOIN     = FULL OUTER JOIN
                (ON- oder USING-Klausel möglich)

NATURAL JOIN = NATURAL INNER JOIN (vermeiden!)
NATURAL LEFT/RIGHT/FULL JOIN      (vermeiden!)
                (ohne ON- oder USING-Klausel)

CROSS JOIN
UNION JOIN
```

Kartesisches Produkt und Inner Join können syntaktisch problemlos mit einfachen `SELECT-FROM-WHERE`-Anfragen formuliert werden. Die Einführung spezieller Inner-Join-Operatoren erhöht daher nicht die Ausdrucksfähigkeit von SQL. Outer Joins können dagegen als echte SQL-Erweiterungen aufgefasst werden, auch wenn sie mit Hilfe von `UNION`, `NOT EXISTS` und Subqueries (umständlich) nachgebildet werden können.

3.4 Laufzeit-Performanz/Optimierung

3.4.1 Selektion mit Index-Unterstützung

Wenn für ein Attribut `a` einer Tabelle `r` ein Index definiert wurde (`CREATE INDEX` ...), können Selektionsanfragen mit Prädikaten der Art

`a =` <*konstanter Wert*>

sehr schnell beantwortet werden – unabhängig von der Anzahl der Elemente der Tabelle `r`:

```
SELECT * FROM r WHERE a=5
SELECT * FROM r WHERE a=5 OR a=6
```

Der Vorteil eines Indexes ist, dass nicht die ganze Tabelle nach geeigneten Tupeln durchsucht werden muss. Das heißt, durch die Definition von Indexen für einzelne Attribute oder auch für mehrere Attribute gemeinsam, kann der DB-Entwickler Anfrage-Antwortzeiten enorm beschleunigen. Insbesondere ein Primärindex, der

für die Primärschlüssel-Attribute meist sogar automatisch definiert wird, kann die Beantwortung von Anfragen unter Umständen enorm beschleunigen.

Allerdings erkauft man sich dies mit leichten Performanz-Einbußen bei den Update-Operationen. Jede Modifikation am Datenbestand hat immer auch eine Modifikation aller zugehörigen Indexe zur Folge.

Gute DBMS-interne Optimierer nutzen vorhandene Indexe aus, indem sie die Optimierung „Selektion so früh wie möglich“ durchführen, damit die relationalen Operatoren möglichst kleine Tabellen verarbeiten.

Zum Beispiel kann die Anfrage

```
SELECT name, adresse
FROM   haendler
WHERE  adresse = 'Königsbrunn'
```
(Nenne mir die Namen aller Königsbrunner Händler)

auf zwei Arten beantwortet werden:

$\pi_{\texttt{name,adresse}}(\sigma_{\texttt{adresse='Königsbrunn'}}(\texttt{haendler}))$
$\sigma_{\texttt{adresse='Königsbrunn'}}(\pi_{\texttt{name,adresse}}(\texttt{haendler}))$

Die erste Variante („frühe Selektion“) ist wesentlich effizienter, wenn für das Attribut `adresse` ein Index definiert wurde. Im zweiten Fall entfernt nämlich der Projektions-Operator zunächst von *jedem* Tupel der Relation `haendler` die Händlernummer `hnr` bevor die Selektion einen Großteil der so modifizierten Tupel einfach löscht.

Wenn ein Index die Sortierreihenfolge der Attribute erhält (wie z. B. B-Bäume oder UB-Bäume, nicht aber Hash-Indexe), können auch Selektions-Anfragen mit Ungleichungsprädikaten effizienter beantwortet werden. Zum Beispiel:

```
SELECT * FROM t WHERE 5<a AND a<=10
SELECT * FROM t WHERE a BETWEEN 6 AND 10
SELECT * FROM t WHERE a>7
```

Im ersten Fall ist das Ergebnis der Anfrage vermutlich klein, im dritten Fall dagegen viel größer. In allen Fällen sorgt der Index jedoch dafür, dass keine unnötigen Tupel bearbeitet werden. Der Vorteil, den der Einsatz eines Indexes mit sich bringt, ist umso größer, je größer die Differenz zwischen Tabellengröße und Ergebnisgröße ist.

Noch eine weitere Anfrageart kann von Indexen, die die Sortierung erhalten, profitieren: Vergleiche mit dem `LIKE`-Operator.

```
SELECT * FROM haendler WHERE adresse LIKE 'König%'
SELECT * FROM haendler WHERE adresse LIKE '%brunn'
```

Im ersten Fall werden alle Händler gesucht, die in einem Ort wohnen, dessen Namen mit `König` beginnt. Im anderen Fall werden die Händler gesucht, die in einem Ort wohnen, der auf `brunn` endet.

Der erste Fall, in dem das Ende des Suchstrings „gestutzt“ wird (englisch: end truncation), wird von einem Index, der die Sortierreihenfolge erhält, unterstützt: Alle gesuchten Wörter liegen im Index direkt hintereinander. Die zweite Anfrage kann dagegen von diesem Index nicht profitieren. Hier müssen die Adressen **aller** Händler der Reihe nach untersucht werden.

Join-Optimierung

In SQL definiert man die Join-Reihenfolge i. Allg. nicht explizit, sondern überlässt es den Optimierern, eine möglichst optimale Reihenfolgen-Optimierung des kartesischen Produktes vorzunehmen. Dies ist allerdings sehr schwierig, da es $O(n!)$ ($> O(n^c)$ und $O(c^n)$ für jede natürliche Zahl c) mögliche Joinvarianten für n Relationen gibt ($5! = 120$, $10! = 3\,628\,800$, $20! = 2432902008176640000$). Es gibt Methoden, eine sehr gute Joinreihenfolge mit der Komplexität $O(2^n)$ ($2^5 = 32$, $2^{10} = 1024$, $2^{20} = 1\,048\,576$) zu bestimmen. Für $n > 10$ ist auch das noch zu viel – vor allem, wenn Anfragen interaktiv ausgewertet werden sollen. Dafür gibt es heuristische Verfahren (z.B. so genannte Greedy-Algorithmen) zur Joinoptimierung mit der Komplexität $O(n^2)$ ($10^2 = 100$, $20^2 = 400$). Allerdings können diese Verfahren normalerweise nicht auf beliebige, sondern nur auf spezielle zu joinende Relationen angewendet werden. (Garcia-Molina et al. [2002])

Auch für die Join-Optimierung gilt: Je kleiner die Tabellen sind, die ein Join-Operator verarbeiten muss, desto schneller erfolgt der Join. Daher ist die Optimierungs-Regel „Selektion so früh wie möglich“ auch hier gültig. Dies gilt meist selbst dann, wenn die Selektions-Operation nicht durch einen Index beschleunigt wird, da Join-Operationen teurer als Selektions-Operationen sind.

Beispiel

```
SELECT r.*
FROM   r, s, t
WHERE  r.a=s.a AND s.b=t.b AND t.c=5
```

Der langsamste Relationale-Algebra-Ausdruck wäre der folgende: kartesisches Produkt plus späte Selektionen:

$$\pi_{\texttt{r.*}}(\sigma_{\texttt{r.a=s.a} \wedge \texttt{s.b=t.b} \wedge \texttt{t.c=5}}((\texttt{r} \times \texttt{s}) \times \texttt{t}))$$

Schneller wird die Anfrage beantwortet, wenn Optimierer hocheffiziente Equijoin-Algorithmen anstelle des kartesischen Produktes einsetzt:

$$\pi_{\texttt{r.*}}(\sigma_{\texttt{t.c=5}}((r \bowtie_{\texttt{r.a=s.a}} \texttt{s}) \bowtie_{\texttt{s.b=t.b}} \texttt{t}))$$

Allerdings erfolgt hier die Selektion viel zu spät. So müssen die Tabellen `r`, `s` und `t` vollständig gejoint werden, obwohl nur ein geringer Teil davon benötigt wird. Viel besser ist eine frühe Selektion. Da bei der Selektion nur ein Attribut von `t` berücksichtigt wird, kann die Selektions-Operation direkt auf `t` angewendet werden:

$$\pi_{\texttt{r.*}}((\texttt{r} \bowtie_{\texttt{r.a=s.a}} \texttt{s}) \bowtie_{\texttt{s.b=t.b}} (\sigma_{\texttt{t.c=5}}(\texttt{t})))$$

Dieser Ausdruck ist immer noch nicht optimal, da `r` und `s` vollständig gejoint werden. Um kleine Zwischenergebnisse zu erhalten, ist es i. Allg. besser, zunächst `s` mit der verkleinerten Tabelle `t` zu joinen. Für genauere Aussagen müssten allerdings über alle drei Tabellen statistische Informationen vorliegen. Ohne zusätzliche statistische Informationen sieht der folgende relationale Ausdruck am erfolgversprechensten aus.

$$\pi_{\texttt{r.*}}(\texttt{r} \bowtie_{\texttt{r.a=s.a}} (\texttt{s}_{\texttt{s.b=t.b}} \bowtie (\sigma_{\texttt{t.c=5}}(\texttt{t}))))$$

Wenn dem Optimierer noch effizientere Algorithmen für die Berechnung eines Semijoins zur Verfügung stehen, könnte er die beiden Join-Operationen ($\bowtie$) durch Semijoin-Operationen ($\ltimes$) ersetzen. In beiden Fällen werden nämlich die Attribute des Join-Partners für die weiteren Berechnungen nicht mehr benötigt.

3.5 Aggregation und Gruppierung

SQL bietet neben den Standard-Relationale-Algebra-Operatoren auch noch Aggregation und Gruppierung. Diese sind für die Praxis sehr wichtig (bedeuten aber für die zugrundeliegende Theorie und damit für die Anfrageoptimierung deutlich mehr Aufwand).

3.5.1 Aggregation ohne Gruppierung

Wie lautet der Durchschnittspreis des Artikels Nr. 3?

```
SELECT AVG(preis)
FROM   liefert
WHERE  wnr = 3
```

Wieviele verschiedene Artikel werden derzeit angeboten?

```
SELECT COUNT(*)
FROM   ware
```

Oder genauer (wenn *lieferbare* Waren gemeint sind):

```
SELECT COUNT(*)
FROM   (SELECT DISTINCT wnr FROM liefert)
```

Oder kürzer:

```
SELECT COUNT(DISTINCT wnr)
FROM   liefert
```

In allen diesen Fällen wird eine Menge von Tupeln auf einen einzigen Wert abgebildet. Diesen Vorgang nennt man *Aggregation*.

SQL unterstützt folgenden Aggregationsfunktionen:

`COUNT`: Zähle Elemente

`SUM`: Bilde (numerische) Summe

`AVG`: Bilde (numerischen) Durchschnitt

`MIN`: finde Minimum

`MAX`: finde Maximum

Achtung

`COUNT(*)`: Zähle alle

`COUNT(preis)`: Zähle alle, deren Preise ungleich `NULL` sind

Bei allen Aggregationsfunktionen kann man bestimmen, ob Duplikate berücksichtigt werden sollen (`ALL` = Defaultwert) oder nicht (`DISTINCT`). Für die Minimum- und Maximum-Berechnung macht dies allerdings keinen Unterschied.

Aggregatfunktionen können nicht geschachtelt werden:

```
SELECT AVG(SUM(x)) ...
```

ist Nonsense.

Anstatt dessen muss man einen komplexeren SQL-Ausdruck verwenden.

```
SELECT AVG(s) FROM (SELECT SUM(x) as s ...)
```

3.5.2 Aggregation mit Gruppierung

Tabellen können vor der Aggregation in Gruppen eingeteilt werden. Die Aggregation wird dann für jede Gruppe separat durchgeführt. Im Gegensatz zur Aggregation ohne Gruppierung liefert die Aggregation mit Gruppierung i. Allg. mehr als ein Element als Ergebnis (und zwar für jede Gruppe eines).

Beispiel

liefert

hnr	wnr	preis
1	1	50
2	1	100
3	1	150
1	2	50
3	2	100

Wie lautet der Durchschnittspreis der einzelnen Waren?

Hier müssen die Daten also nach den Waren gruppiert werden:

```
SELECT   wnr, AVG(preis)
FROM     liefert
GROUP BY wnr;
```

Ergebnis

wnr	AVG(preis)
1	100
2	75

Man beachte: Wenn eine GROUP-BY-Klausel verwendet wird, dürfen Attribute, die nicht in der Gruppierungsliste stehen, nur *mit* Aggregatsfunktion (COUNT, AVG, SUM etc.) in die Projektionsliste geschrieben werden.

Also nicht:

```
SELECT   wnr, preis
FROM     liefert
GROUP BY wnr;
```

Sondern:

```
SELECT   wnr, AVG(preis)
FROM     liefert
GROUP BY wnr;
```

Man kann auch Bedingungen für ganze Gruppen angeben: Alle Gruppen mit mehr als drei Elementen, mit einem Durchschnittspreis größer als ein bestimmter Wert etc. Dazu gibt es die HAVING-Klausel, deren Attribute (genauso wie in der Projektionsliste) wie folgt aussehen müssen: GROUP-BY-Attribute werden *ohne* Aggregationsfunktion aufgerufen, alle anderen Attribute *mit* Aggregationsfunktion.

Wie lautet der Durchschnittspreis für jeden Artikel billiger als 1000 Euro, für den es mindestens drei verschiedene Preise (< 1000 Euro) gibt?

```
SELECT   wnr, AVG(preis) AS "Durchschnittspreis"
FROM     liefert
WHERE    preis < 1000        -- AND wnr < 100 (erlaubt)
GROUP BY wnr
HAVING   COUNT (DISTINCT preis) >= 3;
                        -- AND wnr < 100 (auch hier erlaubt)
```

Man beachte, dass die Bedingung `preis < 1000` nur in der `WHERE`-Klausel stehen darf, da sie gruppenunspezifisch ist. Die Bedingung `COUNT(DISTINCT preis) >= 3` darf dagegen nur in der `HAVING`-Klausel stehen, da sie eine gruppenspezifische Aggregatsfunktion beinhaltet. Bedingungen zu Gruppierungsattributen wie `wnr < 100` dürfen in beide Bedingungsklauseln eingefügt werden. Hier entscheidet der Geschmack. Es gibt allerdings DBMS, bei denen es effizienter ist, diese Bedingung in die `WHERE`-Klausel zu schreiben, da diese vor der Gruppierung ausgewertet wird, die `HAVING`-Klausel jedoch erst danach.

3.6 Unteranfragen

Die `SELECT`-, `FROM`-, `WHERE`- und `HAVING`-Klauseln sowie die Update-Befehle (`INSERT`, `DELETE`, `UPDATE`) dürfen vollständige `SELECT`-Klauseln enthalten. Derartige Anfragen heißen Unteranfragen (Subqueries). Ich gehe im Folgenden davon aus, dass alle Tabellen in Unteranfragen keine `NULL`-Werte enthalten. Die Behandlung von `NULL`-Werten in diesem Zusammenhang würde den Vorlesungsrahmen sprengen.

Projektionslisten mit Unteranfragen

Jede Unteranfrage in einer Projektionsliste muss stets genau ein Element als Ergebnis liefern.

Beispiel

```
SELECT
    (SELECT wert FROM autor WHERE docid=5),
    (SELECT wert FROM titel WHERE docid=5)
FROM dummy;
```

Damit verlassen wir den Bereich der reinen relationalen Algebra. Die relationalen Ausdrücke können nur noch etwas unkonventionell formuliert werden:

$\pi_{\pi_{\texttt{wert}}(\sigma_{\texttt{docid=5}}(\texttt{autor})),\pi_{\texttt{wert}}(\sigma_{\texttt{docid=5}}(\texttt{titel}))}(\texttt{dummy})$

Hier sind `autor` und `tabelle` zwei Tabellen mit Primärschlüssel `docid` und einem weiteren Attribut `wert`. Die Tabelle `dummy` enthält lediglich ein einziges

Element für dessen Inhalt wir uns nicht weiter interessieren. Es wird aus syntaktischen Gründen benötigt, da in SQL immer eine `FROM`-Klausel vorhanden sein muss. In Oracle gibt es zu diesem Zweck standardmäßig die Dummy-Tabelle `dual`, in PostgreSQL kann – nicht-standard-konform – die `FROM`-Klausel ganz entfallen.

Ein sinnvolleres Beispiel

Wie viele unterschiedliche Waren liefern die einzelnen Händler?
Lösung mit `GROUP-BY`-Klausel (vgl. Abschnitt 3.5.2):

```
SELECT   hnr, name, adresse, COUNT(DISTINCT wnr)
FROM     haendler LEFT JOIN liefert USING (hnr)
GROUP BY hnr     -- SQL 2008: name und adresse dürfen fehlen
```

Lösung ohne `GROUP-BY`-Klausel:

```
SELECT hnr, name, adresse,
       (SELECT COUNT(DISTINCT wnr)
        FROM liefert l
        WHERE l.hnr = h.hnr
       ) AS anzahl
FROM   haendler h
```

Man spricht hier von *korrelierten Unteranfragen* (*correlated subqueries*), da die Unteranfrage in der `SELECT`-Klausel auf Variablen der übergeordneten Anfrage (hier `h.hnr`) Bezug nimmt.In diesem Fall muss die Unteranfrage für jeden Lieferanten einzeln ausgewertet werden. Eine *nicht-korrelierte Unteranfrage* braucht dagegen nur ein einziges Mal ausgewertet zu werden und ist daher zu bevorzugen (sofern man die Wahl zwischen korrelierten und nicht.korrelierten Subqueries hat).

Ein trickreiches Beispiel

Jede Ware soll mit einer laufenden Nummer versehen werden und zwar in der alphabethischen Reihenfolge der Bezeichnungen:

```
SELECT   (SELECT COUNT(*)+1
          FROM   ware w2
          WHERE  w2.bezeichnung < w1.bezeichnung
         ) AS nr,
         w1.bezeichnung
FROM     ware w1
ORDER BY nr
```

Diese Anfrage ist allerdings ziemlich langsam, da im Prinzip das halbe kartesische Produkt `ware` $\times$ `ware` ermittelt werden muss. Auch die `GROUP-BY`-Variante ist nicht performanter:

```
SELECT   COUNT(*) AS nr, w1.bezeichnung
FROM     ware w1 CROSS JOIN ware w2
WHERE    w2.bezeichnung <= w1.bezeichnung
GROUP BY w1.bezeichnung
FROM     ware w1
ORDER BY nr
```

Erst in SQL:2003 wurden die so genannten Window-Funktionen eingeführt, die derartige Anfragen deutlich performanter erledigen. Eine Window-Funktion ist eine spezielle Aggregatsfunktion, die komplexere Gruppierungen ermöglicht als die klassischen Aggregatsfunktionen. Diese Gruppierungen werden direkt in der `FROM`-Klausel angegeben. Die obige Anfrage könnte man mit Hilfe einer Window-Funktion folgendermaßen formulieren:

```
SELECT   COUNT(*) OVER (ORDER BY bezeichnung) AS nr,
         bezeichnung
FROM     ware
ORDER BY nr
```

oder

```
SELECT   RANK() OVER (ORDER BY bezeichnung) AS nr,
         bezeichnung
FROM     ware
ORDER BY nr
```

`FROM`-Klauseln mit Unteranfragen

Beispiel

```
SELECT DISTINCT *
FROM   (SELECT * FROM haendler) AS h
```

Dies ist ein relativ sinnloses Beispiel. Gerade im Zusammenhang mit Aggregation (siehe Abschnitt 3.5) werden Unteranfragen in `FROM`-Klauseln allerdings häufiger benötigt.

Ein sinnvolleres Beispiel

```
SELECT MAX(avgpreis)
FROM (SELECT AVG(preis)
      FROM liefert
      WHERE wnr = 1
      GROUP BY hnr
     ) AS p(avgpreis)
```

liefert den maximalen Durchschnittspreis der Ware mir Warennummer 1. Die folgende Anfrage ist dagegen nicht erlaubt:

```
SELECT   MAX(AVG(preis))
FROM     liefert
WHERE    wnr = 1
GROUP BY hnr
```

`WHERE`/`HAVING`-Klauseln mit Unteranfragen – Die `IN`-Bedingung

Die aus der Mengenlehre bekannte ist-Element-von-Beziehung ($\in$) gibt es auch in SQL. Dabei können sowohl die ist-Elemente-von-, als auch die ist-kein-Element-von-Beziehung auf mehrere Arten ausgedrückt werden.

```
∈: IN      = SOME  MATCH
∉: NOT IN  <> ALL
```

Beispiel

Welche Händler liefern den Artikel mit Warennummer 3?

```
SELECT h.hnr, h.name
FROM   haendler h
WHERE  h.hnr IN
       (SELECT l.hnr
        FROM liefert l
        WHERE l.wnr = 3
       )
```

Oder in (verallgemeinerter) Relationaler-Algebra-Notation:

$\pi_{\texttt{h.hnr, h.name}}(\sigma_{\texttt{hnr IN } \pi_{\texttt{l.hnr}}(\sigma_{\texttt{l.wnr}=3}(\texttt{liefert l}))}(\texttt{haendler h}))$

Hier handelt es sich um eine *nicht-korrelierte Unteranfrage*, da die Unteranfrage nicht Bezug auf Variablen der übergeordneten Anfrage nimmt.

Man könnte die Anfrage allerdings auch mit Hilfe von *korrelierten Unteranfragen* formulieren (was man so weit als möglich vermeiden sollte):

```
SELECT h.hnr, h.name
FROM   haendler h
WHERE  3 IN
       (SELECT l.wnr
        FROM liefert l
        WHERE l.hnr = h.hnr
       )
```

Noch schlimmer:

```
SELECT h.hnr, h.name
FROM   haendler h
WHERE  EXISTS
       (SELECT *
        FROM liefert l
        WHERE l.hnr = h.hnr AND l.wnr = 3
       )
```

Schließlich kann man die Anfrage auch ganz einfach als (Semi-)Join formulieren (mit zusätzlichem `DISTINCT`-Operator):

```
SELECT DISTINCT h.hnr, h.name
FROM            haendler h, liefert l
WHERE           h.hnr = l.hnr AND l.wnr = 3
```

bzw.

```
SELECT DISTINCT h.hnr, h.name
FROM            haendler JOIN liefert l USING (hnr)
WHERE           l.wnr = 3
```

Am effizientesten auszuwerten sind i. Allg. die beiden letzten sowie die erste Anfrage, da man alle drei mit Hilfe eines Semijoins realisieren kann. Die Anfragen mit korrelierter Unteranfrage werden dagegen von den meisten DBMS i. Allg. wesentlich weniger effizient ausgewertet. Diese Möglichkeiten wurden auch eher als *abschreckende* Beispiele gebracht. Immerhin sieht man daran, dass die Verknüpfung von Tabellen auch ohne Join möglich ist, wenn Unteranfragen unterstützt werden. Man muss also als Programmierer immer aufpassen, ob es im Falle von (korrelierten) Unteranfragen nicht auch einfachere – und evtl. effizientere Lösungen gibt.

Noch'n Beispiel

Welche Händler liefern zumindest einige derjenigen Waren, Händler 3 liefert (nur Lieferantennummern):

```
SELECT DISTINCT l.hnr
FROM            liefert l
WHERE           l.wnr IN
                (SELECT l.wnr
                 FROM liefert l
                 WHERE l.hnr = 3
                )
```

Man beachte, dass `l.wnr` außerhalb der Unteranfrage eine andere Bedeutung hat als innerhalb. Man könnte zur Verdeutlichung außerhalb auch überall `l1` und innerhalb überall `l2` schreiben.

Andererseits könnte man in diesem Fall – da es sich um eine nicht-korrelierte Unteranfrage handelt – auch auf alle fett geschriebenen Tupelbezeichnungen (Aliasnamen) völlig verzichten. Ich tendiere allerdings dazu, immer eindeutige Tupelbezeichner zu verwenden, da man diese – als menschlicher „Parser" – besser, d. h. fehlerfreier lesen kann.

Tupel als Werte

Prädikate wie das `IN`-Prädikat können auch Tupel als Werte verarbeiten:

```
...WHERE ROW('Maier', 'Königsbrunn') IN
         (SELECT name, adresse FROM haendler)
```

Wann Join, wann Unteranfrage?

In Abschnitt 3.10.3 werden Sie ein Beispiel sehen, in dem die Verwendung des `IN`-Operators anstelle einer Join-Operation zwingend erforderlich ist. Der `UPDATE`-Operator erlaubt nämlich nicht, eine zu modifizierende Tabelle mit einer anderen Tabelle zu joinen. Allgemein gilt, dass man Unteranfragen verwenden muss, wenn man den Inhalt einer Tabelle modifizieren und dabei die zu modifizierende Tabelle mit einer anderen Tabelle verknüpfen will.

Der `IN`-Operator wird außerdem häufig verwendet, wenn man am Vergleich mit einer Vielzahl von Elementen interessiert ist:

```
SELECT wnr, typ, bezeichnung
FROM   ware
WHERE  typ IN ('CPU','RAM','Sonstiges');
```

anstelle von

```
SELECT wnr, typ, bezeichnung
FROM   ware
WHERE  typ = 'CPU' OR
       typ = 'RAM' OR
       typ = 'Sonstiges';
```

Ein weiteres Anwendungsgebiet von Unteranfragen ist die Vermeidung von falschen Duplikaten.

Beispiel

Wie lauten die Namen aller Händler, die CPUs liefern (z. B. zur Erstellung von Namenskärtchen)? Beachten Sie, dass hier die Händlernummer `hnr` entgegen unseren sonstigen Gepflogenheiten nicht in der Projektionsliste aufgeführt werden soll.

Folgende Anfrage liefert zu viele Tupel (falsche Duplikate):

```
SELECT haendler.name
FROM   haendler JOIN liefert USING (hnr)
                JOIN ware    USING (wnr)
WHERE  typ = 'CPU'
```

Folgende Anfrage liefert zu wenige Tupel (`'Maier'` nur einmal):

```
SELECT DISTINCT haendler.name
FROM            haendler JOIN liefert USING (hnr)
                         JOIN ware    USING (wnr)
WHERE           typ = 'CPU'
```

Folgende Anfrage liefert genau richtig viele Tupel: zweimal `'Maier'`, aber `'Müller'` und `'Huber'` nur je einmal:

```
SELECT haendler.name
FROM   haendler
WHERE  hnr IN
       (SELECT hnr
        FROM liefert JOIN ware USING (wnr)
        WHERE typ = 'CPU'
       )
```

Bei der Join-Variante muss man `hnr` zusätzlich ausgeben, um dieses Ergebnis zu erhalten:

```
SELECT DISTINCT hnr, name
FROM            haendler JOIN liefert USING (hnr)
                         JOIN ware    USING (wnr)
WHERE           typ = 'CPU'
```

Wenn man die Ausgabe des Identifikators vermeiden will, muss man einen zusätzlichen `SELECT`-Befehl einfügen. Folgende Query liefert exakt dasselbe Ergebnis wie die obige Anfrage mit dem `IN`-Operator:

```
SELECT name
FROM (SELECT DISTINCT hnr, name
      FROM   haendler JOIN liefert USING (hnr)
                      JOIN ware    USING (wnr)
      WHERE  typ = 'CPU'
     ) AS aux(hnr, name)
```

Von den zuvor beschriebenen Ausnahmen abgesehen, **sollten im Allgemeinen Joins eingesetzt werden**. Joins sind meist einfacher zu verstehen, können häufig effizienter berechnet werden und außerdem besteht nicht die Gefahr, dass man aus Versehen korrelierte Subqueries erzeugt.

Der MATCH-Operator

Wie bereits erwähnt wurde, bedeutet <*Wert*> MATCH <*Tabelle*> dasselbe wie <*Wert*> IN <*Tabelle*> mit einer Ausnahme: Wenn <*Wert*> gleich NULL ist, liefert IN als Ergebnis UNKNOWN und MATCH liefert TRUE.

```
5 IN (1, 2, 4, NULL)
```

Daneben gibt es noch den Operator MATCH UNIQUE:

```
<Wert> MATCH UNIQUE <Tabelle>
```

Dieser Test liefert nur dann TRUE, wenn die Tabelle genau ein Element besitzt – und zwar <*Wert*> – oder wenn <*Wert*> gleich NULL ist. Im Zusammenhang mit NULL-Werten gibt es noch weitere Match-Varianten, die hier aber nicht diskutiert werden sollen.

Der MATCH-Operator wird derzeit von PostgreSQL nicht unterstützt.

Die ALL- und SOME-Operatoren

Es gibt die Möglichkeit, einen Wert mit allen (ALL) oder einigen Elementen – d. h. mindestens einem Element (SOME, ANY) zu vergleichen.

Billiger als alle:

```
preis <= ALL (SELECT preis FROM liefert WHERE ...)
```

Billiger als irgendeiner (nicht der teuerste):

```
preis < SOME (SELECT preis FROM liefert WHERE ...)
```

Es gibt ALL und SOME (= ANY) für die Operatoren =, <, <=, >, >= und <>.

Wie bereits gesagt wurde, gilt IN = =SOME und NOT IN = <>ALL.

Achtung: In Umgangsenglisch kann ANY nicht nur irgendeiner/s, sondern auch *jeder* bedeuten (Get those parts whose price is less than that of **any** blue part.)

Aus diesem Grund ist es besser immer SOME anstelle ANY zu verwenden.

Der EXISTS-Operator

Dieser Operator dient dazu, festzustellen, ob eine Tabelle leer ist oder nicht.

```
EXISTS <Tabelle>
```

Verwendungsmöglichkeiten – insbesondere zur Realisierung des Allquantors – werden noch ausführlich diskutiert.

Der `UNIQUE`-Operator

Dieser Operator kann genutzt werden, um festzustellen, ob in einer Tabelle Duplikate vorkommen (`FALSE`) oder nicht (`TRUE`):

```
UNIQUE <Tabelle>
```

Achtung: `UNIQUE` funktioniert nur mit der SQL-Multimengensemantik, nicht aber mit der Mengensemantik der relationalen Algebra (da Mengen niemals Duplikate enthalten).

Auch dieser Operator ist derzeit in PostgreSQL nicht implementiert!

Beispiel

Welche Artikel werden von *mehreren* Händlern zum billigsten Preis angeboten?

```
SELECT w.wnr, w.name
FROM   ware w
WHERE  NOT UNIQUE
       (SELECT w.wnr -- oder l1.wnr oder 'Wert ist egal'
        FROM   liefert l1
        WHERE  l1.wnr = w.wnr AND
               l1.preis <=ALL (SELECT l2.preis
                               FROM   liefert l2
                               WHERE  l2.wnr = w.wnr
                              )
       )
```

Andere Lösung (mit Gruppierung und Aggregation; siehe Abschnitt 3.5)

```
SELECT   w.wnr, w.name
FROM     liefert l1 JOIN ware w USING (wnr)
WHERE    l1.preis <=ALL (SELECT l2.preis
                         FROM   liefert l2
                         WHERE  l2.wnr = l1.wnr
                        )
GROUP BY l1.wnr
HAVING   COUNT(l1.hnr) > 1
```

Man beachte, dass man dieselbe Anfrage auch mit reiner Mengensemantik beantworten kann, d. h. ohne den Operator `UNIQUE` und ohne Aggregation. Allerdings ist dies deutlich aufwändiger, da man dazu zwei `EXISTS`-Operatoren benötigt: Es gibt einen Händler `h1`, der zum billigsten Preis liefert, und es gibt einen zweiten Händler `h2` $\neq$ `h1`, der ebenfalls zum billigsten Preis liefert (SQL-Query selbst formulieren!).

3.6.1 Allquantifizierte Fragen

Division

Der Divisionsoperator $\div$ dient dazu, *allquantifizierte* Fragen (. . . gilt für alle . . .) zu beantworten. Er ist im wesentlichen die Umkehroperation des kartesisches Produktes ($(r \times s) \div s = r$).

Der Divisionsoperator $\div$ ist wie folgt definiert:

Es sei $A = \{a_1, \ldots, a_n\}$ die Menge derjenigen Attribute von r die nicht zu s gehören. Dann gilt:

$$r \div s := \pi_A(r) \backslash \pi_A((\pi_A(r) \times s) \backslash r)$$

Beispiel

Im Folgenden wird eine stark vereinfachte Version unserer Standard-Beispiel-Datenbank verwendet:

`liefert`

hnr	**wnr**
1	1
1	2
1	3
2	1
2	2

`ware`

wnr
1
2
3

Folgende allquantifizierte Anfrage soll beantwortet werden:

Welcher Händler liefert *alle* Waren?

Dafür kann der Divisions-Operator $\div$ eingesetzt werden, da die Division von `liefert` durch `ware` alle diejenigen Händler (genauer: deren Nummern) liefert, für die *alle* Waren aus der Tabelle `ware` in der Tabelle `liefert` enthalten sind. Das ist nur bei Händler `1` der Fall.

Wie funktioniert die Division nun genau?

Zunächst bestimmen wir diejenigen Attribute von `liefert`, die **nicht** in `ware` enthalten sind.

$A = \{\texttt{hnr}, \texttt{wnr}\} \backslash \{\texttt{wnr}\} = \{\texttt{hnr}\}$

Damit können wir ermitteln, wer überhaupt als Lieferant aller Waren in Frage kommt.

Es gibt zwei Händler:

$\pi_{\texttt{hnr}}(\texttt{liefert})$

```
SELECT DISTINCT hnr
FROM            liefert
```

hnr
1
2

Wenn jeder Händler alles liefern würde, hätten wir folgende Tabelle:

$\pi_{\texttt{hnr}}(\texttt{liefert}) \times \texttt{ware}$

```
SELECT hnr, wnr
FROM   (SELECT DISTINCT hnr
        FROM            liefert
       ),
       ware
```

hnr	wnr
1	1
1	2
1	3
2	1
2	2
2	3

Allerdings liefert Händler 2 Ware 3 nicht:

$(\pi_{\texttt{hnr}}(\texttt{liefert}) \times \texttt{ware}) \setminus \texttt{liefert}$

```
SELECT hnr, wnr
FROM   (SELECT DISTINCT hnr
        FROM            liefert
       ),
       ware
EXCEPT
SELECT hnr, wnr
FROM   liefert
```

hnr	wnr
2	3

Das heißt, Händler 2 liefert nicht alles:

$\pi_{\texttt{hnr}}((\pi_{\texttt{hnr}}(\texttt{liefert}) \times \texttt{ware}) \backslash \texttt{liefert})$

```
SELECT hnr              -- genauer: SELECT DISTINCT hnr
FROM (SELECT hnr, wnr
      FROM   (SELECT DISTINCT hnr
              FROM            liefert
             ),
             ware
      EXCEPT
      SELECT hnr, wnr
      FROM   liefert
     )
```

hnr
2

Wenn man von der Liste aller Händler diejenigen abzieht, die *nicht* alles liefern, bleiben nur diejenigen Lieferanten übrig, die alles liefern. Hier heißt das, dass nur Händler 1 alles liefert:

$\pi_{\texttt{hnr}}(\texttt{liefert}) \backslash \pi_{\texttt{hnr}}((\pi_{\texttt{hnr}}(\texttt{liefert}) \times \texttt{ware}) \backslash \texttt{liefert})$

```
SELECT hnr
FROM   liefert
EXCEPT
SELECT hnr         -- SELECT DISTINCT hnr ist hier unnötig
FROM (SELECT hnr, wnr
      FROM   (SELECT DISTINCT hnr
              FROM            liefert
             ),
             ware
      EXCEPT
      SELECT hnr, wnr
      FROM   liefert
     )
```

hnr
1

Realisierung in SQL

Der Divisionsoperator könnte relativ effizient realisiert werden (auch wenn die oben vorgestellte Definition wegen der Verwendung des kartesischen Produktes

äußerst ineffizient ist). Er wird dennoch von SQL nicht unterstützt. Das heißt, man muss ihn – z. B. analog zur Definition des Divisionsoperators – durch andere Operatoren nachbilden. Die Verwendung des Cross-Joins ist allerdings zu teuer. Effizienter ist es, z. B. den `EXISTS`-Operator von SQL einzusetzen.

Man beachte, dass folgende Beziehung zwischen *Für alle ...* (*For all*) und *Es gibt ein ...* (*exists*) gilt:

> „Für alle x gilt ...“
> ist gleichwertig zu
> „Es gibt kein x, für das ... nicht gilt“ (doppelte Verneinung!).
>
> $\forall x p(x) \Leftrightarrow \neg\exists x \neg p(x)$
>
> (Es gilt auch: $\exists x p(x) \Leftrightarrow \neg\forall x \neg p(x)$, aber das nützt uns hier nichts)

Der `EXISTS`-Operator überprüft, ob in einer Menge Elemente enthalten sind. Genauer: Der Ausdruck

```
EXISTS (<SELECT-Anweisung>)
```

liefert `TRUE` genau dann, wenn das Ergebnis der `SELECT`-Anweisung nicht leer ist, d. h., wenn *es* in dieser Tabelle wenigstens ein Element `gibt`.

Die Anfrage

```
SELECT r.a
FROM   r
WHERE  EXISTS (SELECT * FROM s WHERE r.b=s.c)
```

kann wie üblich nur als unkonventionaler relationaler Algebra-Ausdruck geschrieben werden:

$\pi_{\texttt{r.a}}(\sigma_{\texttt{EXISTS}(\sigma_{\texttt{r.b=s.c}}(\texttt{s}))}(\texttt{r}))$

Beispiel

Die Anfrage „Welcher Händler liefert *alle* Waren?“ kann folgendermaßen formuliert werden:

```
SELECT h.hnr
FROM   haendler h
WHERE  <h.hnr liefert jede Ware>
≡
SELECT h.hnr
FROM   haendler h
WHERE  <Es gibt keine Ware, die h.hnr nicht liefert>
```

Das heißt, wir müssen zunächst die Frage klären, welche Waren `h.hnr` *nicht* liefert.

```
SELECT l.wnr FROM liefert l WHERE l.hnr = x
```

sagt uns, welche Waren der Lieferant x liefert. Durch Differenzbildung erfahren wir, welche Ware er nicht liefert:

```
(SELECT w.wnr FROM ware w)
EXCEPT
(SELECT l.wnr FROM liefert l WHERE l.hnr = x)
```

Nun können wir den `NOT-EXISTS`-Operator von SQL einsetzen, um unsere ursprüngliche Frage zu beantworten.

```
SELECT h.hnr
FROM   haendler h
WHERE  NOT EXISTS
       ((SELECT w.wnr FROM ware w)
        EXCEPT
        (SELECT l.wnr FROM liefert l
         WHERE l.hnr = h.hnr
        )
       )
```

Hier liegt wieder eine korrelierte Unteranfrage vor.

Andere Möglichkeiten der Realisierung in SQL

Die Anfrage

```
SELECT l.hnr
FROM   liefert l
WHERE  l.hnr = x AND
       l.wnr = y
```

liefert ein Ergebnis genau dann wenn Händler x die Ware y liefert. Anderenfalls ist das Ergebnis die leere Tabelle. Das heißt, wir können die Waren, die x *nicht* liefert, auch mit Hilfe einer korrelierten Unteranfrage ermitteln:

```
SELECT w.wnr
FROM   ware w
WHERE  NOT EXISTS
       (SELECT * FROM liefert l
        WHERE  l.hnr = x AND
               l.wnr = w.wnr
       )
```

Und damit können wir unsere ursprüngliche Frage nach den Lieferanten *aller* Waren mittels zwei verschachtelter korrelierter Unteranfragen formulieren:

```
SELECT h.hnr
FROM   haendler h
WHERE  NOT EXISTS
       (SELECT *
        FROM   ware w
        WHERE  NOT EXISTS
               (SELECT *
                FROM   liefert l
                WHERE  l.hnr = h.hnr AND
                       l.wnr = w.wnr
               )
       )
```

Diese Art der Formulierung der Division in SQL ist sehr umständlich und auf den ersten Blick unverständlich. Außerdem ist die Auswertung vermutlich ineffizient (sofern es sich bei dem DBMS-Optimierer nicht um ein Wunderwerk handelt). Meistens geht es jedoch einfacher, wenn man die Aggregatfunktion `COUNT` verwendet. Man zählt einfach, wieviele Waren es gibt, und vergleicht dies mit der Zahl der Waren, die ein Lieferant liefern kann:

```
SELECT h.hnr
FROM   haendler h
WHERE  (SELECT COUNT(w.wnr)
        FROM   ware w
       )
       =
       (SELECT COUNT(DISTINCT l.wnr)
        FROM   liefert l
        WHERE  l.hnr = h.hnr
       )
```

Dritte Möglichkeit der Realisierung in Nicht-Standard-SQL

In einigen DBMS (z. B. in TransBase von TransAction Software GmbH) ist es auch möglich, Mengen zu vergleichen: `=`, `<>`, `SUBSET` etc., auch wenn dies im SQL-Standard nicht vorgesehen ist. Damit kann die Frage, welcher Lieferant alle Waren liefert, relativ „natürlich“ formuliert werden:

```
SELECT h.hnr
FROM   haendler h
WHERE (<Alle Waren, die h liefert>) = (<Alle Waren, die es gibt>)
```

Das heißt:

```
SELECT h.hnr
FROM   haendler h
WHERE  (SELECT DISTINCT l.wnr FROM liefert l
        WHERE l.hnr = h.hnr
       )
       =
       (SELECT w.wnr FROM ware w)
```

3.7 Sortierung

SQL realisiert nicht die Mengensemantik der Relationalen Algebra, sondern eine Multimengensemantik. Das heißt, Duplikate werden nicht automatisch entfernt. Dies ist z. B. für Aggregation notwendig. Allerdings realisiert SQL keine Listensemantik. Das heißt, die Reihenfolge der Ergebnistupel einer Anfrage ist nicht festgelegt und kann sich bei denselben Daten und denselben Anfragen von DBMS zu DBMS, ja sogar von Anfrage-Zeitpunkt zu Anfrage-Zeitpunkt unterscheiden. Wenn es auf die Reihenfolge der Ergebnistupel ankommt, kann man daher ans Ende des äußersten `SELECT`-Statements immer eine `ORDER-BY`-Klausel anhängen:

Beispiel

Sortiere die Händler nach ihren Adressen (= Wohnort) und innerhalb der Wohnorte alphabetisch:

```
SELECT   h.hnr, h.name, h.adresse
FROM     haendler h
ORDER BY adresse, name (nicht h.adresse, h.name,
                        dies gilt als „deprecated“ in SQL3)
```

Es können beliebig viele Attribute der Ergebnis-Projektionsliste in der `ORDER-BY`-Klausel angegeben werden.

Jedes Attribut kann dabei aufsteigend (`ASC`, default) oder absteigend (`DESC`) sortiert werden:

```
SELECT   h.hnr, h.name, h.adresse
FROM     haendler
ORDER BY name DESC
```

3.8 Die SELECT-Anweisung: Zusammenfassung

Eine SELECT-Anfrage sieht wie folgt aus (die Nummerierung bezeichnet dabei die zeitliche Abfolge der Abarbeitung):

```
1     [WITH      ...]
6     SELECT     <Projektionsliste>
2     FROM       <Relationen + Joins>
3     [WHERE     <Bedingung>]
4     [GROUP BY <Gruppierungsattribute>]
5     [HAVING    <Gruppenbedingung>]
7     [UNION | EXCEPT | INTERSECT    SELECT ...]
8     [ORDER BY <Sortierungsattribute>]
9     [OFFSET <Anzahl>] [FETCH NEXT <Anzahl> ROWS ONLY]
                                                  (SQL:2008)
9     [LIMIT <Anzahl>] [OFFSET <Anzahl>]   (Nicht-Standard-SQL)
```

Beachten Sie: Da die Sortierung nach der Projektion erfolgt, muss jedes Sortierattribut in der Projektionsliste auftauchen. Da die WHERE- und HAVING-Klauseln vor der SELECT-Klausel (= Projektionsliste) ausgeführt werden, kann man in den WHERE- und HAVING-Klauseln auf die Abkürzungen, die in der Projektionsliste mit AS eingeführt werden nicht zugreifen. In der ORDER-BY-Klausel geht das dagegen schon. Und es ist manchmal auch notwendig. Die Anfrage

```
SELECT   r.a, s.a
FROM     r, s
WHERE    r.a < s.a
ORDER BY r.a,s.a
```

ist in SQL3 nicht mehr erlaubt. Korrekt muss es wie folgt heißen:

```
SELECT   r.a AS ra,
         s.a AS sa
FROM     r, s
WHERE    r.a < s.a
ORDER BY ra,sa
```

3.9 Views

In SQL ist es möglich, so genannte *Views* (*virtuelle* oder *berechnete Relationen*) zu definieren.

```
CREATE VIEW
  lieferant (name, adresse, wnr, preis, dauer) AS
SELECT h.name, h.adresse,
       l.wnr, l.preis, l.lieferzeit
FROM   haendler h, liefert l
WHERE  h.hnr = l.hnr
```

Eine View kann in `SELECT`-Anweisungen wie eine normale Relation (stored table) eingesetzt werden.

```
SELECT AVG(preis)
FROM   lieferant
WHERE  name='Maier'
```

Insbesondere kann eine View auch zur Definition anderer Views eingesetzt werden. Falls allerdings eine View wieder gelöscht wird (z. B. `DROP VIEW lieferant`), werden alle anderen Views und alle Integritätsbedingungen, die sich auf diese View abstützen, ungültig. Die `DROP`-Operation ist daher nur erlaubt, wenn keine derartigen Abhängigkeiten bestehen. Man kann dies verhindern, indem man das Löschen entweder abbricht, wenn noch Abhängigkeiten bestehen (`DROP VIEW <`*name*`> RESTRICT`), oder alle abhängigen Views und Integritätsbedingungen automatisch mitlöscht (`DROP VIEW <`*name*`> CASCADE`).

Ein nachträgliches Ändern einer Views (`ALTER VIEW`) ist im Gegensatz zu Tabellen (`ALTER TABLE`) in SQL nicht möglich.

VIEWs können seit SQL3 auch *rekursiv* definiert werden. Damit ist SQL berechnungsvollständig (turingmächtig) geworden. Zum Beispiel kann mit rekursiven VIEWs das Pfadproblem gelöst werden: Welche Pfade führen – evtl. über mehrere Zwischenknoten – von A nach B? Das Pfadproblem tritt in der Praxis sehr häufig auf (Zugverbindungen, Flugverbindungen, Internetverbindungen, Vorfahren/Nachfahren/Verwandschafts-Beziehungen etc.).

Beispiel Vorfahren

Es sei `eltern(kind,elter)` eine Relation, in der zu jeder Person `kind` ein Elternteil `elter` angegeben ist.

```
eltern
kind           elter
'Sibylle'      'Wolfgang'
'Sibylle'      'Marianne'
'Marianne'     'Jakob'
'Marianne'     'Franziska'
'Franziska'    'Theresia'
⋮
```

Alle Eltern

```
SELECT kind, elter
FROM   eltern
```

Alle Großeltern

```
SELECT e1.kind, e2.elter
FROM   eltern e1 JOIN eltern e2 ON e1.elter = e2.kind
```

Alle Urgroßeltern

```
SELECT e1.kind, e3.elter
FROM        eltern e1
       JOIN eltern e2 ON e1.elter = e2.kind
       JOIN eltern e3 ON e2.elter = e3.kind
```

Alle Vorfahren

Alle Vorfahren können nur mit Hilfe rekursiver Views ermittelt werden. Die Methode, alle obigen Anfrageergenisse mit `UNION` zu vereinigen, funktioniert nur, wenn man die Anzahl der in den Daten maximal enthaltenen Generationen kennt.

```
CREATE RECURSIVE VIEW
  vorfahren (person, vorfahre) AS
SELECT kind, elter FROM eltern)          -- Start
UNION
  (SELECT v.person, e.elter              -- Rekursion
   FROM      vorfahren vi JOIN eltern e
          ON v.vorfahre = e.kind
  )
```

Diese View beschreibt folgende Vorfahren-Tabelle:

```
vorfahren
person          vorfahre
'Sibylle'       'Wolfgang'      1. Generation
'Sibylle'       'Marianne'
'Marianne'      'Jakob'
'Marianne'      'Franziska'
'Franziska'     'Theresia'

'Sibylle'       'Jakob'         2. Generation
'Sibylle'       'Franziska'
'Marianne'      'Theresia'

'Sibylle'       'Theresia'      3. Generation
                                4. Generation
```

4. Generation leer $\Rightarrow$ Ende der Rekursion

Verfahren, um rekursive Views effizient zu berechnen sind heute gut erforscht, allerdings auf dem Gebiet der so genannten *deduktiven* Systeme. Deduktive Sprachen und DBMS wurden vor allen in den 80er- und 90er-Jahren erforscht, waren aber nicht sehr erfolgreich – und leider werden die dabei gewonnenen Erkenntnisse bislang kaum in andere Gebiete (wie z. B. RDBS) übertragen.

PostgreSQL, Oracle, DB2, Firebird und vermutlich noch andere DBMS unterstützen rekursive Tabellenausdrücke (allerdings meist mit gewissen Einschränkungen).

Benannte Anfragen (temporäre Views)

```
WITH <Anfragename>[(<Spaltennamensliste>)]
AS   (<SELECT-Anweisung>)
<SELECT-Anweisung>
```

Beispiel

```
WITH r(a,b)
AS (SELECT …)
SELECT r1.a, r2.b
FROM   r r1, r r2
WHERE  r1.b=r2.a
```

Temporäre Views dürfen auch rekursiv definiert werden:

```
WITH RECURSIVE <Anfragename>[(<Spaltennamensliste>)]
AS   (SELECT …)
[<Traversierungsklausel>]
[<Zyklusklausel>]
SELECT …
```

Beispiel

```
WITH RECURSIVE vorfahren(person, vorfahre)
AS   (SELECT kind AS person, elter AS vorfahre
      FROM   eltern
     )
     UNION
     (SELECT kind AS person, vorfahre
      FROM   eltern, vorfahren
      WHERE  elter = person
     )
SELECT person, vorfahre FROM vorfahre
```

Anmerkung: In PostgreSQL werden derzeit keine rekursiven Views sondern nur rekursive temporäre Views unterstützt.

Mit Kontrolle der Rekursionstiefe:

```
WITH RECURSIVE vorfahren(person, vorfahre, stufe)
AS   (SELECT kind AS person, elter AS vorfahre, 1
      FROM   eltern
     )
     UNION
     (SELECT kind AS person, vorfahre, stufe+1
      FROM   eltern, vorfahren
      WHERE  elter = person
     )
SELECT person, vorfahre
FROM   vorfahren
WHERE  stufe < 5
```

Problem: Optimierung von rekursiven Anfragen

Ein Problem der rekursiven Views ist, dass man die Ergebnisgröße eines Ergebnisses nur sehr schwer abschätzen kann ($\Rightarrow$ automatische Optimierung ist sehr schwer).

Beispiel

Wenn man n Netzknoten (Bahnhöfe, Flughäfen, Router etc.) hat und man berechnen will, von welchem Knoten zu welchem Knoten Verbindungen existieren, enthält das Ergebnis zwischen n und n^2 Tupel:

n Jeder Knoten ist nur mit sich selbst verbunden. (z. B. $n = 100.000 = 10^5$)

n^2 Jeder Knoten ist mit jedem verbunden. ($n = 10^5 \Rightarrow n^2 = 10^{10}$)

Leider lässt sich anhand statistischer Daten vor der eigentlichen Berechnung keine genauere Abschätzung durchführen. Statistische Abschätzungen sind aber ein gängiges Mittel zur Optimierung.

Die folgelden Tabellen sind statistisch vollkommen gleichwertig (gleichviele Tupel, jeder Wert kommt in jeder Spalte gleich häufig vor):

`start`	`ziel`
1	1
2	2
3	3
4	4
⋮	
n	n

`start`	`ziel`
1	2
2	3
3	4
⋮	
$n-1$	n
n	1

Allerdings enthält die transitive Hülle, d. h. die Tabelle aller möglichen Verbindungen im ersten Fall nur n Tupel, im zweiten Fall jedoch n^2 Tupel.

Fazit: Rekursive Views bereiten bei der Abschätzung von Kosten den Optimierern wesentlich mehr Probleme als nicht-rekursive.

3.9.1 Das View-Update-Problem

Views wurden nicht erfunden, um rekursive Definitionen zu ermöglichen oder eine Verwandtschaft zu den deduktiven Systemen herzustellen, sondern um bestimmten Benutzergruppen nur eine eingeschränkte Sicht (View!) auf die Datenbank zu erlauben. So könnte z. B. für die Inventarabteilung eine View erzeugt werden, die es den Mitarbeitern dieser Abteilung ermöglicht, Name, Zimmernummer und Telefonnummer aller Mitarbeiter, nicht aber deren Gehalt und Familienstand zu erfragen. Dazu müssen für die Mitarbeiter der Inventarabteilung eine oder mehrere Datenbank-Benutzergruppen erzeugt werden, die lediglich das Recht haben, diese View zu lesen, nicht aber die Orginaltabelle.

Ein nächster Schritt wäre, diesen Mitarbeitern auch gewisse Updates zu erlauben. In diesem Fall müssten die Befehle `INSERT`, `DELETE` und `UPDATE` auch für

Views ermöglicht werden. Doch das ist nur sehr eingeschränkt möglich. Wenn z. B. eine View mit Hilfe zweier Relationen aufgebaut wurde, ist das Einfügen eines Tupels in die View in SQL nicht erlaubt, da i. Allg. nicht feststeht, welche ein(?) oder zwei(?) Tupel in die zugehörigen Relationen eingefügt werden sollen.

In SQL sind View-Updates nur unter ganz bestimmten Voraussetzungen erlaubt (*Updatable Views*):

1. Die View basiert auf genau einer Relation oder einer Updatable View (kein `JOIN`, `UNION`, `INTERSECT`, `EXCEPT`; keine Subqueries).
2. `SELECT DISTINCT` wurde nicht verwendet.
3. Es gibt keine `GROUP-BY`- oder `HAVING`-Klausel.
4. Es sind nicht alle `WHERE`-Klauseln erlaubt.
5. Jede Spalte der `SELECT`-Klausel ist ein Aliasname für eine Tabellenspalte (keine Arithmetik etc.).
6. Alle Spalten, für die keine Defaultwerte existieren (wie z. B. der Primärschlüssel), sind über die View zugänglich.

3.10 Modifikation des Datenbestandes

Die relationale Algebra sagt nichts darüber aus, wie man einen Datenbestand erzeugt und pflegt. Sie geht einfach davon aus, dass eine Menge von Relationen bekannt ist. SQL bietet dagegen nicht nur die Möglichkeit, einen Datenbestand per Anfragesprache zu untersuchen (`SELECT`), sondern auch ihn zu manipulieren (`INSERT`, `DELETE`, `UPDATE`).

3.10.1 Insert

Es gibt zwei Möglichkeiten, Daten in eine Datenbank via `INSERT` einzufügen.

1. Direkt:

```
INSERT INTO haendler(hnr, name)
VALUES      (5, 'Müller')

INSERT INTO haendler(hnr, name, adresse)
VALUES      (6, 'Schmidt', 'München'),
            (7, 'Schmitt', NULL)
```

```
INSERT INTO haendler (hnr, name, adresse)
VALUES
  ((SELECT COALESCE(MAX(hnr)+1, 1) FROM haendler),
   'Schmidt', 'München'
  ),
  ((SELECT COALESCE(MAX(hnr)+2, 2) FROM haendler),
   'Schmitt', NULL
  )
```

Anmerkung

Nicht aufgeführte Attribute werden mit einem Defaultwert (siehe Abschnitt 3.10.4) oder `NULL` initialisiert, wenn ein derartiger Wert existiert bzw. erlaubt ist.

Anmerkung 2

Man sollte stets die Attributnamen explizit angeben:

```
INSERT INTO haendler (hnr, name, adresse)
VALUES      (4, 'Huber', 'München')
```

an Stelle von:

```
INSERT INTO haendler
VALUES      (4, 'Huber', 'München')
```

Der Grund dafür ist: Durch Modifikationen des Relationenschemas kann sich die Anzahl und oder Reihenfolge der Attribute ändern. Dies kann zur Folge haben, dass zugehörige `INSERT`-Befehle ohne explizite Attributnamen Daten unter falschen Attributenamen erfassen.

2. Per `SELECT`-Statement:

```
INSERT INTO haendler (...)
SELECT … FROM …
```

Achtung: Neben dem `INSERT`-Befehl bieten die meisten DBMSs noch die Möglichkeit, externe Daten (z. B. in einem festen ASCII/ISO-Latin/UTF-Format) direkt in die Datenbank einzulesen (spool in) oder direkt in eine externe Datei auszugeben (spool out). Der Spoolvorgang ist allerdings nicht standardisiert. Das gilt auch für das Einfügen von `BLOB`-Objekten (auch wenn es mit Hilfe von normalen `INSERT`-Befehlen möglich ist `BLOB`s zu erzeugen). Hier bieten allerdings ODBC und JDBC Abhilfe, da diese Standards relativ einheitliche Verfahren zum Einfügen, Modifizieren und Lesen von `BLOB`s bieten.

3.10.2 Delete und Drop

Daten können aus einer Datenbank auch wieder gelöscht werden:

Alle Maiers und Mairs und Maierhubers sollen gelöscht werden:

```
DELETE
FROM   haendler
WHERE  name LIKE 'Mai%r'
```

Alle Tupel sollen gelöscht werden:

```
DELETE FROM haendler
```

Achtung: Solange in der Tabelle `liefert` noch Fremdschlüssel auf Händler-Tupel verweisen, können diese nicht gelöscht werden. Da heißt, man muss entweder zunächst alle `liefert`-Tupel löschen oder ein automatisches Löschen der zu einem Händler zugehörigen `liefert`-Tupel erzwingen. Letzteres errreicht man durch die Angabe von `ON DELETE CASCADE` bei der Definition der zugehörigen Fremdschlüsselbeziehung in der `liefert`-Tabelle:

```
CREATE TABLE liefert
(hnr        INTEGER      NOT NULL,
 wnr        INTEGER      NOT NULL,
 preis      NUMERIC(8,2) NOT NULL,
 lieferzeit SMALLINT     CHECK (lieferzeit >= 0),

 PRIMARY KEY (hnr, wnr, preis),
 FOREIGN KEY (hnr) REFERENCES haendler (hnr)
                   ON DELETE CASCADE
                   ON UPDATE CASCADE,
 FOREIGN KEY (wnr) REFERENCES ware (wnr)
)
```

Um die ganze Tabelle (und nicht nur deren Inhalt) zu entfernen, muss man Folgendes schreiben:

`DROP TABLE haendler` $\equiv$ `DROP TABLE haendler RESTRICT`
(Gegenteil von `CREATE TABLE`)

Wie beim `CREATE-TABLE`-Befehl gibt es die Möglichkeit, Löschungen zu kaskadieren. Mit

```
DROP TABLE haendler CASCADE
```

wird nicht nur die Tabelle gelöscht, sondern auch alle zugehörigen Untertabellen (Vererbung) sowie alle Views, Constraints und Trigger, die von dieser Tabelle abhängen.

3.10.3 Update

Bestehende Datensätze können modifiziert werden.

Alle Maiers erhöhen ihre Preise um 10 % und ihre Lieferfristen um einen Tag:

```
UPDATE liefert
SET    preis = preis*1.10, lieferzeit = lieferzeit+1
WHERE  hnr IN
       (SELECT hnr
        FROM   haendler
        WHERE  name = 'Maier'
       )
```

Beachten Sie, dass hier der `IN`-Operator und eine Unteranfrage anstelle eines Joins verwendet wird. Das muss so sein, da man die zu modifizierende Tabelle nicht mit einer anderen Tabelle joinen kann.

Achtung: Ein Update kann auch auf Schlüsselattributen erfolgen, meist allerdings unter Performanzeinbußen.

3.10.4 Allgemeine Anmerkungen

Bei `UPDATE`- und `INSERT`-Befehlen können die zwei Sonderwerte `NULL` und `DEFAULT` angegeben werden. `DEFAULT` bezeichnet dabei den Defaultwert des aktuellen Datentypes.

Der Programmierer kann den Defaultwert einer Tabellenspalte bei der Definition der entsprechenden Tabelle

```
CREATE TABLE haendler
(name VARCHAR(20) DEFAULT '???', ...)
```

oder mit Hilfe von Domänendefinition

```
CREATE DOMAIN namestring AS VARCHAR(20) DEFAULT '???'

CREATE TABLE haendler (name namestring, ...)
```

festlegen.

Modifikation am Datenbestand sind nur unter bestimmten Bedingungen möglich:

1. Der Benutzer muss die entsprechenden Rechte besitzen.
2. Durch die Modifikation werden keine Integritätsbedingungen verletzt.
3. Die zugehörige Transaktion wird durch `commit` oder Autocommit beendet (und nicht etwa durch `rollback` verworfen).

4. Der Commit wird erfolgreich beendet. Insbesondere heißt das, dass keine der Integritätsbedingungen, deren Überprüfung bis zum Commitzeitpunkt verzögert wurden, auf einen Fehler läuft.
5. Kein Deadlock bei Tabellenzugriff.
6. Die Datenbank wird später (im Katastrophenfall) nicht durch einen Recoveryvorgang auf einen älteren Zustand zurückgesetzt.

Modifikationsoperationen erfolgen zweistufig:

Erst wird alles berechnet, was geändert werden soll, dann werden die Änderungen durchgeführt. Wäre dem nicht so, so wären die Ergebnisse folgender Operationen nicht vorhersagbar, da sich der Durchschnittspreis mit jedem Tupelupdate verändern würde:

```
DELETE FROM liefert
WHERE preis >= (SELECT AVG(preis) FROM liefert)

UPDATE liefert
SET    preis = 0.9*preis
WHERE  preis >= 2*(SELECT AVG(preis) FROM liefert)
```

3.11 Transaktionen

In SQL werden nicht nur ACID-Transaktionen (Abschnitt 1.6), sondern insgesamt vier verschiedene Arten von Transkationen unterstützt. Je nach gewählter *Isolationsebene* können verschiedene Probleme auftreten bzw. vermieden werden.

Lost update

Von zwei Änderungen, die gleichzeitig durchgeführt werden, geht eine verloren (da die eine Änderung die andere überschreibt).

Transaktion A	Transaktion B
lese a	
	lese a
	ändere a
ändere a	
	schreibe a
schreibe a	

Dirty Read

Eine Transaktion liest einen Wert, der von einer anderen, noch nicht erfolgreich beendeten Transaktion geändert wurde. Bricht die andere Transaktion ab, so ist

dieser Wert ungültig.

Transaktion A	Transaktion B
lese a	
ändere a	
	lese (geändertes) a
breche Transaktion ab	

Non-Repeatable Read

Eine Transaktion, die zweimal auf dasselbe Tupel zugreift, erhält dabei eventuell unterschiedliche Werte.

Transaktion A	Transaktion B
lese a	
	lese a
ändere a	
	lese (geändertes) a

Phantom Read

Eine Transaktion liest (gespeicherte) Werte, während eine andere Transaktion diese modifiziert, löscht oder neue einfügt.

Transaktion A	Transaktion B
lese a	
	lese b
ändere a abh. von Anzahl Elemente in b	
	füge Element in b ein
	schreibe b
schreibe a	

Mit Hilfe des Befehls

```
SET TRANSACTION <access mode> ISOLATION LEVEL <level>
```

wobei

```
<access mode> = READ ONLY oder READ WRITE
<level> = READ UNCOMMITED oder
          READ COMMITED oder
          REPEATABLE READ oder
          SERIALIZABLE
```

wird festgelegt, ob die darauf folgenden Transaktionen nur lesend oder auch schreibend auf die Daten zugreifen und welche der oben beschriebenen Probleme vermieden werden sollen. Dabei gilt folgender Zusammenhang:

Isolationsebene	Lost Update	Dirty Read	Non-Repeatable Read	Phantom Read
keine Transaktion	ja	ja	ja	ja
`READ UNCOMMITTED`	nein	ja	ja	ja
`READ COMMITTED`	nein	nein	ja	ja
`REPEATABLE READ`	nein	nein	nein	ja
`SERIALIZABLE` (= ACID)	nein	nein	nein	nein

Eine Transaktion wird entweder explizit durch

```
START TRANSACTION
COMMIT AND CHAIN (aktuelle TA beenden und neue starten)
ROLLBACK AND CHAIN (aktuelle TA abbrechen und neue starten)
```

oder implizit durch Aufruf eines SQL-Befehls (`SELECT`, `INSERT`, `UPDATE`, `DELETE` und andere) gestartet.

Eine Transaktion wird laut Standard durch eine der beiden Befehle `COMMIT` oder `ROLLBACK` beendet.

Die Befehle

```
COMMIT (= COMMIT WORK = COMMIT AND NO CHAIN )
bzw.
COMMIT AND CHAIN
```

versuchen die Ergebnisse einer Transaktion dauerhaft zu schreiben. Dabei kann der Versuch erfolgreich sein oder auch nicht (z. B. weil Benutzerrechte fehlen oder Integritätsbedingungen vereltzt werden). Im letzteren Fall ist das Ergebnis dasselbe, als wenn eine Transaktion explizit mit

```
ROLLBACK (= ROLLBACK WORK = ROLLBACK AND NO CHAIN )
bzw.
ROLLBACK AND CHAIN
```

abgebrochen worden wäre. Sämtliche Änderungen der Transaktion werden rückgängig gemacht (mit Ausnahme von – je nach Isolationsebene – eventuellen Dirty-Read-, Non-Repeatable-Read- oder Phantom-Read-Effekten).

Viele Systeme (wie z. B. ODBC und JDBC) unterstützen darüberhinaus einen so genannten „Auto Commit“. Dabei werden SQL-Befehle, die nicht explizit (z. B. nach `START TRANSAKTION`) innerhalb einer Transaktion ausgeführt werden,

automatisch direkt nach dem Abarbeiten „committed“. Dieses Verhalten ist jedoch nicht standardkonform.

Neben dem vollständigen Rollback einer ganzen Transaktion ist es auch möglich, so genannte Sicherungspunkte zu setzen und mittels `ROLLBACK TO SAVEPOINT` nur die Aktionen, die nach dem Sicherungspunkt erfolgten, rückgängig zu machen.

```
INSERT INTO ...;
SAVEPOINT mein_sicherungspunkt_1;
INSERT INTO ...;
SELECT SUM(...) into summe from ...;
IF summe > ... THEN ROLLBACK;
ELSE IF summe > ... THEN
     ROLLBACK TO SAVEPOINT mein_sicherungspunkt_1;
ELSE
     COMMIT;
ENDIF;
```

In SQL gibt es zwei Zeitpunkte, zu denen die Erfüllung von Integritätsbedingungen überprüft werden kann:

1. Direkt nach Ausführung eines SQL-Befehls (`IMMEDIATE`).
2. Bei Beendigung der aktuellen Transaktion durch `COMMIT` (`DEFERRED`).

Eine Integritätsbedingung (`CONSTRAINT`) kann entweder mit der Option `NOT DEFERRABLE` (Default) oder mit der Option `DEFERRABLE` versehen werden. Im ersten Fall wird die Bedingung stets sofort nach der Ausführung eines entsprechenenden SQL-Befehls überprüft. Im zweiten Fall kann jederzeit mit Hilfe des Befehls `SET CONSTRAINT` der Zeitpunkt des Tests (neu) festgelegt werden.

Beispiel 1

```
CONSTRAINT mein_fk
FOREIGN KEY (a) REFERENCES a(id)
DEFERRABLE INITIALLY IMMEDIATE
```

Die Bedingung `mein_fk` wird zunächst sofort am Ende eines entsprechenden SQL-Befehls durchgeführt. Mit Hilfe eines der folgenden Befehle

```
SET CONSTRAINTS ALL DEFERRED;
SET CONSTRAINTS mein_fk DEFERRED;
```

kann dieses Verhalten jedoch jederzeit geändert werden.

Beispiel 2

```
START TRANSACTION;

CREATE TABLE a
(id INTEGER NOT NULL,
 b INTEGER NOT NULL,
 CONSTRAINT p_a
   PRIMARY KEY (id)
);

CREATE TABLE b
(idINTEGER NOT NULL,
 a INTEGER NOT NULL,
 CONSTRAINT p_b
   PRIMARY KEY (id),
 CONSTRAINT f_b_a
   FOREIGN KEY (a) REFERENCES a(id)
   DEFERRABLE INITIALLY DEFERRED
);

/* Kann erst hier definiert werden, aber nicht direkt in TABLE a. */
ALTER TABLE a ADD
   CONSTRAINT f_a_b
   FOREIGN KEY (b) REFERENCES b(id)
   DEFERRABLE INITIALLY DEFERRED;

COMMIT WORK;

/* Funktioniert nicht, wenn Auto-Commit aktiv! */
INSERT INTO a VALUES (1,2);
INSERT INTO b VALUES (2,1);

/* Funktioniert nicht! */
START TRANSACTION;
INSERT INTO a VALUES (1,2);
COMMIT WORK;
START TRANSACTION;
INSERT INTO b VALUES (2,1);
COMMIT WORK;
```

/* Funktioniert! */
```
START TRANSACTION;
INSERT INTO a VALUES (1,2);
INSERT INTO b VALUES (2,1);
COMMIT WORK;
```

/* Funktioniert! */
```
START TRANSACTION;
DELETE FROM a;
DELETE FROM b;
COMMIT WORK;
```

/* Funktioniert nicht! */
```
SET CONSTRAINTS f_a_b IMMEDIATE;
START TRANSACTION;
INSERT INTO a VALUES (1,2);
INSERT INTO b VALUES (2,1);
COMMIT WORK;
```

/* Analog mit:*/
```
SET CONSTRAINTS f_a_b, f_b_a IMMEDIATE;
SET CONSTRAINTS ALL IMMEDIATE;
```

/* Funktioniert! */
```
SET CONSTRAINTS ALL DEFERRED;
START TRANSACTION;
INSERT INTO a VALUES (1,2);
INSERT INTO b VALUES (2,1);
COMMIT WORK;
```

Kapitel 4

Datenbank-Management-Systeme und Multimedia

4.1 SQL-MM-Standards

SQL2 sieht keine Unterstützung von Multimedia vor. Das hat sich allerdings mit SQL3 geändert. In einer Reihe von Standards werden wichtige Multimedia-Erweiterungen von SQL definiert:

- Large Objects – LOBs (ISO:2003-2 [2007], SQL:2008-2 [2008])
- XML (ISO:2006-14 [2006], SQL:2008-14 [2008])
- SQL/CLI = Call-Level Interface = Basis für ODBC (ISO:2003-3 [2007], SQL:2008-3 [2008])
- Embedded SQL, insbesondere SQLJ (ISO:2003-10 [2007], SQL:2008-10 [2008])
- SQL/JRT – Java-Routinen können innerhalb von SQL aufgerufen werden (ISO:2003-13 [2003], SQL:2008-13 [2008])
- JDBC = Java-SQL-Schnittstelle (Ellis et al. [2001], Andersen [2006])
- SQL/MED = Management of External Data – z.B. Bilder, Videos etc. ((ISO:2003-9 [2003], SQL:2008-9 [2008])
- Framework = Basis-Definitionen (SQL/MM:2007-1 [2007])
- Full-Text = Volltext-Management (SQL/MM:2003-2 [2003])
- Spatial = 3D-Objekt-Management (SQL/MM:2006-3 [2006])
- Still image = Bild-Management (SQL/MM:2003-5 [2003])
- Data mining (SQL/MM:2006-6 [2006])
- History = Versionierung (SQL/MM:2007-7 [2007], noch nicht verabschiedet)

Für viele weitere MM-Themenkomplexe wie Video/Streaming, Audio etc. gibt es derzeit leider noch keine Standards. Doch auch für die aktuellen Standards gibt es aufgrund deren Komplexität erst ziemlich wenige praxistaugliche Implementierungen. Häufig gibt es dagegen irgendwelche proprietären Ersatzlösungen.

4.2 Anforderungen an ein Multimedia-DBMS

Welche Eigenschaften sollte ein gutes Multimedia-Datenbank-Management-System (MMDBMS) erfüllen?

1. Es sollte ein vollwertiges DBMS sein:
   - Persistenz (d. h. dauerhafte Speicherung der Daten)
   - Mehrbenutzerfähigkeit
   - Integritätssicherung (Konsistenz)
   - Transaktionen (Robustheit im Fehlerfall)
   - Recovery (Robustheit im Katastrophenfall)
   - Performanz
   - Ad-hoc-Anfragen
   - etc.
2. Es sollte Multimedia-Objekte (auch Methoden!) unterstützen: Text (insbesondere HTML/XML), Bild, Audio, Video (Streaming!), komplexe *multi*-mediale Objekte ($\Rightarrow$ Interaktion, Synchronisation), Hyperlinks.
3. Es sollte – für MM-Objekte – Informations-Retrieval-Fähigkeiten bieten:
   - Attributsuche (einfach, da DBMS dies bereits unterstützen)
   - Inhaltssuche (i. Allg. sehr schwierig), insbesondere Volltextsuche (nicht so schwierig)
   - navigierende Suche/Linkverfolgung (nicht so schwierig, wird aber derzeit nicht standardmäßig unterstützt)

Vorteile von Multimedia-DBMS

1. Saubere Verwaltung großer Mengen an Multimediadaten. (Zum Beispiel erzeugen Abbrüche von langen Kopiertransaktionen keine Dateileichen.)
2. Schneller Zugriff auf einzelne Objekte.
   (Gegenbeispiel: Ext2-/FAT-Directory mit 10 000 Objekten.)
3. Optimierte Ablage von MM-Objekten. Verwaltung von DVD-Archiven oder Blue-Ray-Archiven (Jukeboxes), Bandarchiven (veraltet), Plattenfarmen etc., Strukturierung der Daten durch Indexe (B-Bäume, 3D-Indexe etc.).

4. Datenbankeigenschaften: Mehrbenutzerbetrieb, Datensicherheit, Konsistenz (z. B. referenzielle Integrität)[1] etc.
5. Effizientes Information Retrieval.
6. Und anderes mehr.

4.2.1 Relationale DBMS und Multimedia

Viele DBMS unterstützen Multimedia-Features auf die ein oder andere Weise:

- Die meisten Datenbanken können BLOBs (Binary Large Objects) und CLOBs (Character Large Objects) verwalten. Leider sind viele Lösungen immer noch nicht standard-konform.
- Volltextsuche wird von vielen Datenbanken unterstützt.
- Oracle (Oracle [2002]) und andere unterstützen Video-Streaming.
- Datenbanken auf DVDs und DVD-Jukeboxes werden unterstützt (z. B. TransBase, TransAction [2002]).
- Es ist möglich WWW-Seiten mit Datenbank-Unterstützung zu erstellen (z. B. mit Ruby on Rails oder JDBC).
- Es gibt spezielle WWW-Datenbankserver (z. B. von Oracle).
- Es gibt zahlreiche NoSQL-DBMS für Spezialaufgaben.

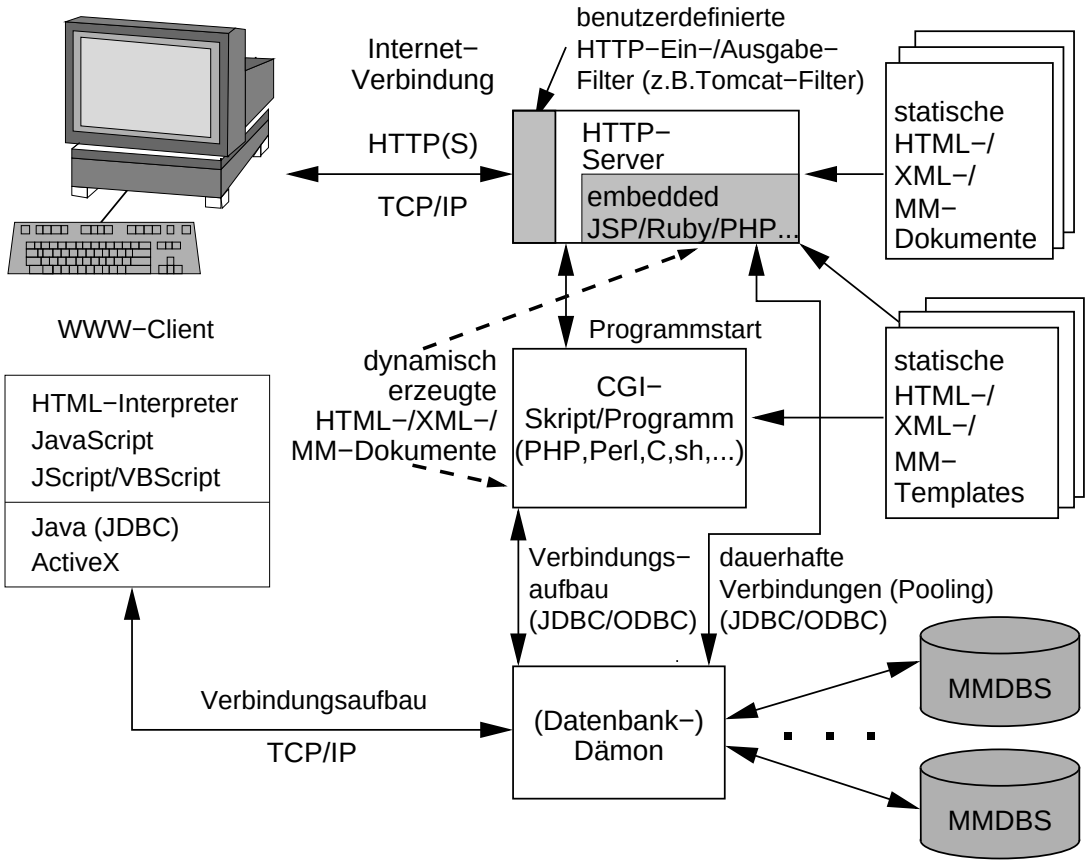
Leider sind diese Techniken und Spracherweiterungen meist proprietärer Natur. Das heißt, jeder Hersteller kocht zurzeit sein eigenes Süppchen.

4.3 Kopplung WWW ↔ MMDBS

Es gibt mehrere Möglichkeiten, über WWW mit einem (MM)DBS zu kommunizieren:

1. Vom Server aus via Perl, PHP, ASP, Python, Tcl, C/C++, ... (meist ODBC).
2. Vom Java-Server aus via Java, JSP, XSP, ... (meist JDBC).
3. Vom Rails-Server aus via integrierte DB-Zugriffsfunktionen.
4. Direkt via HTTP über Spezial-HTTP-Server (z.B. Oracle Web Server).
5. Vom Client aus via JAVA (JDBC), ActiveX (ODBC) o. Ä.
6. Et cetera.

[1] Die referenzielle Integrität ist eine der wesentlichsten Schwachstellen der WWW: URLs zeigen häufig ins Nirwana.



ad 1: Wenn die Skript-Sprache nicht in den Server integriert ist, muss für jeden DB-Zugriff ein eigener CGI-Prozess gestartet werden. Aufgrund des daraus resultierenden Overheads wird CGI heute kaum noch verwendet.

Um auf Daten in einem MMDBMS zugreifen zu können, muss sich der Nutzer jeweils authentifizieren. Dies gilt auch für einen HHTP-Server und seine Hilfsprogramme. Um nicht für jedes dynamisch generierte Dokument den zeitaufwendigen Authentifizierungsvorgang durchlaufen zu müssen, werden heute meist so genannte *Connection Pools* eingesetzt. Beim Start des Servers werden einige Verbindungen zum Datenbankserver aufgebaut und in einen derartigen Pool gelegt. Sobald eine Anfragebearbeitung Zugriff auf eine Datenbank benötig, holt Sie sich eine schon bestehende Verbindung aus dem Pool. Wenn die Anfrage abgearbeitet wurde, wird die Verbindung wieder in den Pool zurückgegeben. Ein Pool-Manager sorgt dafür, dass (sofern möglich) stets genug freie Verbindungen im Pool vorhanden sind. Wenn die Zahl der freien Verbindungen zu groß wird, schließt der Pool-Manager einfach ein paar davon.

4.3.1 JDBC

JDBC (**J**ava **D**ata**B**ase **C**onnectivity) ist eine standardisierte Schnittstelle, um von Java-Programmen aus mit einer oder mehreren Datenbanken zu kommunizieren. Die verschiedenen JDBC-Spezifikationen und -Schnittstellen-Dokumentationen können direkt bei Oracle heruntergeladen werden: `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`.

JDBC unterstützt den Verbindungsaufbau, die Übermittlung von SQL-Statements und die schrittweise Abarbeitung von Anfrageergebnissen.

Ein großes Problem von JDBC ist direkte Übermittlung von SQL-Befehlen an die jeweilige Datenbank. Da sich der SQL-Sprachumfang von DBMS zu DBMS unterscheidet, können JDBC-Programme normalerweise nicht problemlos von einem DBMS auf ein anderes umgestellt werden. JDBC versucht dieses Problem mittels so genannter *SQL-Escapes* abzumildern.[2] Zum Beispiel kann man Datumstrings in der Form `{d '1998-09-13'}` schreiben. Der jeweilige JDBC-Treiber setzt diese Schreibweise in die (häufig nicht SQL-konforme) Schreibweise des zugehörigen DBMS um. Viele der SQL-Escapes vermindern allerdings die Portabilität anstatt sie zu erhöhen, da von den DB-Herstellern nicht verlangt wird, dass sie alle SQL-Escapes einheitlich unterstützen.

Um das Portabilitätsproblem zumindest ein wenig in den Griff zu bekommen, fordert SUN von den Datenbankherstellern für JDBC-Treiber, dass ihre DBMS mindestens den SQL92 Entry Level unterstützen. Nur wenn diese „Mindestausstattung“ vorhanden ist und viele weitere Bedingungen erfüllt werden, kann der SUN-Konformitätstest mit Erfolg absolviert werden und der JDBC-Treiber das Gütesiegel „JDBC compliant“ erhalten (siehe `http://java.sun.com/products/jdbc/driverdevs.html`). Allerdings gibt es auch Treiber ohne dieses Siegel.

Einen weiteren Schritt zu mehr Portabilität stellen die (sehr vielen) verschiedenen boolesche Methoden dar, die eine Auskunft über die Fähigkeiten des aktuellen DBMS geben:

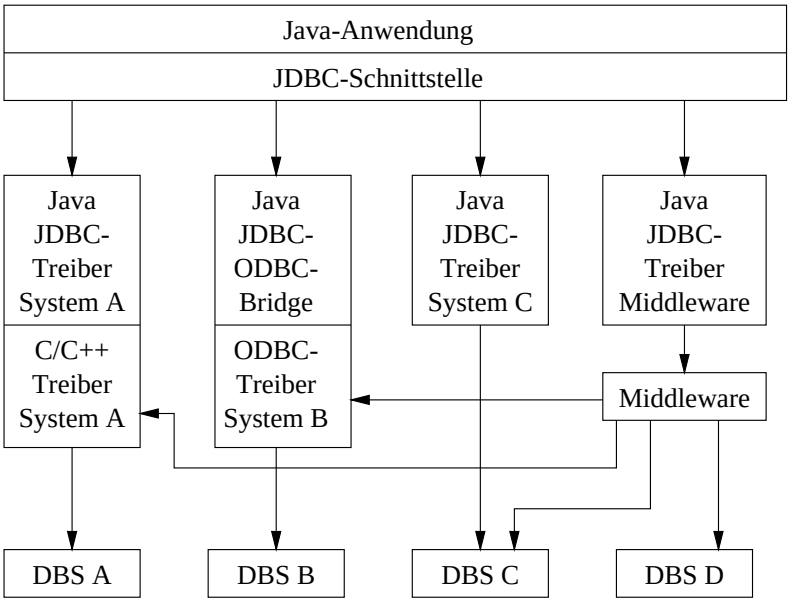
```
supportsANSI92EntryLevelSQL()
supportsSubqueriesInComparisons()
...
```

[2] Idee und Syntax stammen von ODBC (Open Database Connectivity) von Microsoft.

JDBC-Treiber

Es gibt verschiedene Arten von JDBC-Treiber. Zum Beispiel:

1. datenbankspezifische Java-Erweiterungen von datenbankspezifischen C/C++/. . .-Treibern (< 100 % Java, plattformabhängig, DB-abhängig)
2. datenbankunspezifische Java-Erweiterungen von ODBC-Treibern, so genannte JDBC-ODBC-Bridges (< 100 % Java, plattformabhängig, relativ DB-unabhängig)
3. datenbankspezifische Java-Treiber (100 % Java, plattform*un*abhängig, DB-abhängig)
4. so genannte Middleware (oft Java, plattform*un*abhängig; die Middleware selbst muss nicht in Java geschrieben sein – in diesem Fall läuft die Middleware auf speziellen Plattformen).



ad 1: Diese Lösung kann vom Datenbankhersteller schnell implementiert werden, ist allerdings nicht plattformunabhängig.

ad 2: Diese Lösung kann von Drittanbietern schnell implementiert werden, ist aber nur eingeschränkt plattformunabhängig (abhängig von der Verfügbarkeit von ODBC-Treibern für viele Plattformen).

ad 3: Die Implementierung ist für den Datenbankhersteller aufwändiger. Allerdings ist der Treiber plattformunabhängig.

ad 4: Die Implementierung ist für Drittanbieter sehr aufwändig (wenn nicht auf vorhandene Schnittstellen zurückgegriffen werden kann). Der Treiber selbst ist plattformunabhängig, die Middleware dagegen meist plattformabhängig.

Servlets

Javaprogramme mit JDBC-Kopplung können nicht nur auf Clients laufen (so genannte *Applets*), sondern auch auf Servern (so genannte *Servlets*).

Allerdings muss der Server fähig sein Servlets auszuführen. Beispiele:

- Server, die in Java implementiert wurden (Tomcat, Jetty, Jigsaw von W3C etc.) können automatisch Servlets ausführen.
- Für den Apache gibt es diverse Möglichkeiten, Anfragen zur Bearbeitung an einen Java-Server (wie z.B. Tomcat) weiterzuleiten.
- Microsoft IIS kann mit Hilfe des Java SDKs ebenfalls Servlets ausführen.

Eine einfache JDBC-Anwendung

Im Wesentlichen muss jedes JDBC-Programm folgende Anweisungen ausführen:

<*Laden eines JDBC-Treibers*>
<*Öffnen einer Datenbank*>

```
While (!<Ende>)
{
 <Senden einer SQL-Anweisung>
 <Auslesen und Bearbeiten der Ergebnisse>
}
```
<*Datenbank schließen*>

Beispiel

```
import java.io.*;
import java.sql.*;

class Test
{
 public static void main(String[] args)
 throws java.lang.ClassNotFoundException,
        java.sql.IOException,
        java.sql.SQLException
 {
   // Einen oder mehrere JDBC-Treiber dynamisch, d. h. zur Laufzeit laden:
   // Class.forName("openlink.jdbc.Driver");
   // Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   // Class.forName("transbase.jdbc.Driver");
   Class.forName("org.postgresql.Driver");
```

```
// Eine Datenbankverbindung aufbauen
// (der passende Treiber wird automatisch ermittelt):
Connection conn =
  DriverManager.getConnection
    ("jdbc:postgresql://localhost:5432/haendler/",
     "kowa",    // username
     "geheim"   // password
    );

// Nun kann für diese Verbindung ein Statement-Objekt erzeugt werden:
Statement stmt = conn.createStatement();

// Für die Laufzeitumgebung kann man das Ausgabe-Encoding anpassen:
PrintStream out
  = new PrintStream(System.out, true, "UTF-8");

// SQL-Anfragen werden über Statementobjekte an die Datenbank geschickt;
// das Ergebnis dieser Anfragen sind Ergebnisobjekte:
ResultSet rs = stmt.executeQuery
               ("SELECT * FROM haendler h " +
                "WHERE h.name = 'Maier'"
               );
ResultSetMetaData rsmd = rs.getMetaData();

// Als Ergebnis wird ein einfacher Text ausgegeben.
// Alternativ könnte man auch HTML oder XML erzeugen.
// Die Ausgabe fängt mit den Spaltennamen (Metadaten!) an:
for (int i=1; i<=rsmd.getColumnCount(); i++)
  out.print((i==1 ? "" : ", ") + rsmd.getColumnName(i));
out.println();

// Nun können die Ergebnistupel ausgegeben werden:
while (rs.next())   // Solange noch ein Tupel da ist ...
{                   // (funktioniert auch bei leeren Ergebnissen)
  for (int i=1; i<=rsmd.getColumnCount(); i++)
    out.print((i==1 ? "" : ", ") + rs.getObject(i));
  out.println();
};
```

```
    // Nun heißt es aufräumen:
    rs.close();          // Ergebnisobjekt schließen
    stmt.close();        // Statementobjekt schließen
    conn.close();        // Datenbankverbindung schließen
  }
}
```

Unter `http://mmdb.hs-augsburg.de/beispiel/` finden sich weitere JDBC-Beispiele, wie z. B. ein einfacher Web-Auftritt für die Hochschule Augsburg.

4.4 LOBs

Normale SQL-Datentypen wie `binchar` oder `varchar` sind längenbeschränkt (z. B. 64 KByte). Um große Datensätze (LOBs = Large Objects) in Datenbanksystemen zu speichern, wurden deshalb die so genannten **BLOB**s (*Binary Large Objects*) bzw. **CLOB**s (*Character Large Objects*) eingeführt.

Allerdings dürfen auch BLOBs und CLOBs nicht beliebig groß sein. So darf ein `LONGBLOB` (diese Bezeichnung ist nicht standard-konform) in MySQL nicht größer als 4 GB sein. Auch in Oracle (vor Version 10g) dürfen LOBs nicht größer als 4 GB sein. Oft ist auch noch die Gesamtgröße der Datenbank beschränkt. Für die meisten Anwendungen sind dies keine echten Beschränkungen, aber im Videobereich, bei der Speicherung von medizinischen Aufnahmen (Tomografien etc.) und ähnlichen Anwendungen kann es schnell zu Engpässen kommen. Basierend auf der 64-Bit-Technologie sollten im Laufe der Zeit die meisten DBMS deutlich größere LOBs unterstützen (z. B. Oracle 10g: bis 128 TB).

Datenbankoperationen auf LOBs:

- Einfügen, Löschen, Ersetzen
- Test auf `NULL` (`WHERE picture IS NOT NULL`)
- Bestimmung der Größe in Bytes
- Selektion des ganzen LOBs oder eines Teilbereichs (hier leisten B-Bäume gute Dienste)
- Verketten von zwei LOBs (Konkatenation)
- Berechnung eines Hashwertes (z.B. md5)

Gleichheitstests werden manchmal nicht unterstützt, alle LOBs gelten dann als verschieden (wichtig für `distinct` und Gruppierung). Primär- und Sekundär-Indexe können evtl. nicht angelegt werden. LOBs können meist nicht direkt

über die Ad-hoc-Query-Interfaces eingelesen und ausgegeben werden. Die Ein- und Ausgabe erfolgt entweder mit Hilfe eines proprietären Spool-Mechanismuses oder über JDBC, ODBC, ESQL (*Emdedded SQL* = in 3GL-Sprache eingebetteter SQL-Code) etc.

Beispiel: LOBs in PostgreSQL

LOBs werden wie andere Datentypen auch als Attribut-Typen angegeben. Man muss allerdings bei jedem DBMS damit rechnen, dass die Bezeichnung nicht standard-konform ist. In PostgreSQL kann beispielsweise die wohlbekannte Händlertabelle um ein Text-Attribut `anmerkung` und eine Binär-Attribut `logo` erweitert werden. Dabei ist zu beachten, dass PostgreSQL nicht verhindert, dass beliebige Binär-Daten und nicht nur Bilddaten im Attribut `logo` gespeichert werden.

```
CREATE TABLE haendler
 (hnr         INTEGER NOT NULL,
  name        VARCHAR(20) NOT NULL,
  adresse     VARCHAR(20),
  anmerkung   TEXT, /* ≈ CLOB; max 1 GByte */
  logo        BYTEA /* ≈ BLOB; max 2 GByte */

  CONSTRAINT pk_haendler PRIMARY KEY (hnr),
  CONSTRAINT unique_name_address UNIQUE (name, adresse)
 );
```

LOBs und JDBC

In JAVA (JDBC) werden LOBs mit Hilfe von Binary Streams bzw. Character Streams ein- und ausgelesen.

Im folgenden Beispiel werden eine Text- und eine Bild-Datei in die zuvor definierte `haendler`-Tabelle eingefügt. Das vollständige Beispiel (d. h. die Klassen `InsertBLOB`, `SelectBLOB` und `DeleteEntry` sowie Hilfs-Dateien) finden sich unter `http://mmdb.hs-augsburg.de/beispiel/`.

```
import java.sql.*;
import java.io.*;
class InsertBLOB
{
 public static void main(String[] args)
 throwsjava.lang.ClassNotFoundException,
       java.io.FileNotFoundException,
       java.io.UnsupportedEncodingException,
       java.sql.SQLException
```

```
{
 Connection conn = null;
 Class.forName("org.postgresql.Driver");

 try
 {
  conn
    = DriverManager.getConnection
      ("jdbc:postgresql://localhost:5432/haendler/",
       "kowa",
       "geheim"
      );

  PreparedStatement stmt = null;

  File anmfile = new File("wkanmerkung.txt");
  InputStreamReader anmstream
    = new InputStreamReader
            (new FileInputStream(anmfile), "UTF-8");

  File logofile = new File("wklogo.jpg");
  FileInputStream logostream
    = new FileInputStream(logofile);

  stmt
    = conn.prepareStatement
      ("INSERT INTO " +
       "  haendler(hnr, " +
       "           name, adresse, anmerkung, logo) " +
       "VALUES ((SELECT max(hnr)+1 FROM haendler), " +
       "        ?, ?, ?, ?)"
      );
  stmt.setString(1, "Kowa EDV-Beratung");
  stmt.setString(2, "Königsbrunn");
  stmt.setCharacterStream
    (3, anmstream, (int)anmfile.length());
  stmt.setBinaryStream
    (4, logostream, (int)logofile.length());

  stmt.executeUpdate();
 }
```

```
  finally
  {
   if(conn != null)
     conn.close();
  }
 }
}
```

Embedded SQL

In ESQL (Embedded SQL) fängt jeder SQL-Befehl mit `EXEC SQL` an und hört mit `;` auf. C- (oder C++- oder COBOL- oder sonstige) Variablen, auf die in einem SQL-Statement zugegriffen wird, müssen zunächst in einer `DECLARE SECTION` definiert werden. In SQL-Statements kann auf derartige Variablen zugegriffen werden, indem man sie mit einem Doppelpunkt versieht.

```
// Definition von Variablen
EXEC SQL BEGIN DECLARE SECTION;
blobdesc mein_logo;
EXEC SQL END DECLARE SECTION;

mein_logo.mode = FILENAME;
mein_logo.loc.filename = "wklogo.jpg";

EXEC SQL
  INSERT INTO haendler(hnr, name, adresse, logo)
  VALUES ((SELECT max(hnr)+1 FROM haendler),
          'Kowa EDV', 'Königsbrunn', :mein_blob
         );
```

Dieses Programm wird zunächst mit Hilf eines DBMS-spezifischen Precompiler in reinen C-/C++/...-Code übersetzt. Anschließend kann das Programm mit einem Standard-Compiler in Maschinen-Code übersetzt werden.

Für JAVA gibt es den Embedded-SQL-Standard *SQLJ*. Eingebettete SQL-Anweisungen sehen hier folgendermaßen aus:

```
#sql  <SQL-Anweisung>
```

LOBs laut SQL3-Standard

Folgende LOBs sollen laut SQL3-Standard von einem DBMS unterstützt werden:

```
BLOB
BLOB(<length>)
BINARY LARGE OBJECT
BINARY LARGE OBJECT(<length>)
CLOB
CLOB(<length>)
CHARACTER LARGE OBJECT
CHARACTER LARGE OBJECT(<length>)
```

Bei CLOBs kann im Gegensatz zu BLOBs noch ein „Character Set“ und auch noch eine benutzerdefinierte oder DBMS-spezifische „Collation“ angegeben werden; insbesondere gibt es eine sprachspzezifische Variante (`NCLOB` = `NATIONAL CLOB`). Die Collation legt die Sortierreihenfolge fest: z.B. ob „ö“ bei der Sortierung wie „o“ oder „oe“ zu behandeln ist:

```
CLOB(2G) CHARACTER SET UTF8
CLOB(2G) CHARACTER SET UTF8 COLLATE utf8_german_ci
```

Achtung: CLOB(2G) bedeutet nicht, dass nur ein Text im Umfang von höchstens 2 GByte = 2.147.483.648 Bytes in einem zugehörigen Attribut gespeichert werden kann, sondern dass bis zu 2 G = 2.147.483.648 Characters in diesem Attribut abgelegt werden können. Bei UTF-8 beispielsweise kann jedes Zeichen bis zu 3 Byte groß sein.

Operationen auf LOBs

Explitzite Definition eines LOB-Wertes:

```
X'005AFED102F' -- BLOB-Wert in Hex-Schreibweise
'Ein Text mit Umlauten äöü ...' -- CLOB-Wert = String
```

LOBs können mit `=` und `<>` verglichen werden. Dies kann jedoch sehr teuer sein und ist daher nicht Bestandteil von „SQL3 Core SQL“, sondern nur von „SQL3 Enhanced SQL“.

LOBs können konkateniert werden:

```
X'AF' || X'FE' = X'AFFE'
'Wolfgang' || 'Kowarschick' = 'WolfgangKowarschick'
```

Aus LOBs können Substrings extrahiert werden:

```
SUBSTRING (X'543210' FROM 3) = X'10' -- ab dem 3. Byte!
SUBSTRING ('Wolfgang' FROM 3) = 'lfgang'
```

Mit OVERLAY können Teilstrings im LOB durch andere ersetzt werden.

Mit TRIM können bestimmte Zeichen (wie z. B. Leerzeichen) vom Anfang (LEADING), Ende (TRAILING) oder beiden Seiten (BOTH) eines LOBs entfernt werden.

Mit POSITION kann man die Position eines (meist kurzen) LOBs in einem (meist längeren) LOB ermitteln.

Mit BIT_LENGTH, OCTET_LENGTH und CHAR_LENGTH kann die Länge eines LOBs ermittelt werden. Für CLOBs können OCTET_LENGTH und CHAR_LENGTH unterschiedliche Werte liefert, da ein Character aus mehreren Octets (= Bytes) bestehen kann.

Das LIKE-Prädikat funktioniert auch für LOBs (und hat auch dieselben Probleme hinsichtlich der Performanz).

Weitere Operation sind für LOBs nicht definiert. LOBs dürfen weder Teil eines Schlüssels, noch einer GROUP-BY- oder ORDER-BY-Klausel sein.

4.5 Inhaltssuche, insbesondere Volltextsuche

Eine wichtige Aufgabe von MMDBMS ist es, Medien, deren Inhalte bestimmte Eigenschaften haben, möglichst schnell zu finden:

- Finde alle Texte, die die Wörter „Datenbank“ und „Web“ enthalten (Volltextsuche).
- Finde alle Bilder, die zu einem gegebenen Bild ähnlich sind (Image Retrieval, siehe z.B. Teynor et al. [2005], http://lmb.informatik.uni-freiburg.de/people/teynor/demo.de.html)
- Finde alle Videos, die einen Sonnenuntergang enthalten und zeige die entsprechende Szene an.
- Finde das Musikstück, das der Benutzer in sein Mikrofon pfeift/summt (z. B. http://www.midomi.com/, http://www.musicline.de/de/melodiesuche).

Während die allgemeine Inhaltssuche noch nicht das Stadium der Forschung verlassen hat, ist die Volltextsuche (einschließlich der Suche in XML-Datenbeständen) ausgereift und produktiv einsetzbar. Deshalb werde ich mich im Weiteren auf die Volltextsuche beschränken.

4.5.1 Precision und Recall

Das Ziel jeder Volltextmaschine ist es, Anfragen korrekt und vollständig zu beantworten. Das heißt:

$$\text{recall} = \frac{|\text{relevant objects returned}|}{|\text{objects returned}|} \approx 1 \text{ (es gilt immer } \leq 1)$$

$$\text{precision} = \frac{|\text{relevant objects returned}|}{|\text{relevant objects in DB}|} \approx 1 \text{ (es gilt immer } \leq 1)$$

Die Recall wird umso schlechter, je mehr „Falsch-positiv-Ergebnisse“ die Anfrage als Ergebnis liefert (Fehler 1. Art), d. h. je mehr relevante Ergebnisse fälschlicherweise als relevant betrachtet werden.

Die Precision wird umso schlechter, je mehr „Falsch-negativ-Ergebnisse“ die Anfrage als Ergebnis liefert (Fehler 2. Art), d. h. je mehr relevante Ergebnisse fälschlicherweise als nicht-relevant betrachtet werden.

Leider sinkt fast immer die *precision* mit wachsendem *recall* und umgekehrt. Da heißt, obwohl es viele Volltext-Suchtechniken gibt, sind hochwertige Antworten häufig die Ausnahme. Und das wird auch so bleiben, solange die Volltext-Engines nicht intelligent genug sind, die erfassten Texte semantisch (= inhaltlich) zu verstehen.

4.5.2 Volltextsuche laut SQL/MM-Standard

Gemäß dem SQL/MM-Full-Text-Standard (SQL/MM:2003-2 [2003]) gibt einen weiteren Datentyp: `FULLTEXT`. Ihm ist defaultmäßig eine Sprache (z.B. „German“) zugeordnet. Dieser Datentyp unterstützt die Volltextsuche.

```
CREATE TABLE haendler
 (hnr         INTEGER NOT NULL,
  name        VARCHAR(20) NOT NULL,
  adresse     VARCHAR(20),
  anmerkung   FULLTEXT,

  CONSTRAINT pk_haendler PRIMARY KEY (hnr),
  CONSTRAINT unique_name_address UNIQUE (name, adresse)
 );
```

Mit Hilfe der `FULLTEXT`-Methoden `CONTAINS` und `SCORE` können Volltext-Attribute auf das Vorkommen von **Pattern** hin durchsucht werden.

```
SELECT * FROM haendler
```

```
WHERE  anmerkung.CONTAINS(’ "Wolfgang Kowa%" ’) = 1

SELECT * FROM haendler
WHERE  anmerkung.SCORE(’ "Wolfgang Kowarschick" ’) > 0.75
```

Ein Text besteht aus einzelnen Zeichen (Character), die zu größeren Einheiten zusammengefasst werden können:

- Buchstabenfolge (Characters)
- einzelnes Wort
- Wortfolge (Phrase)
- Satz
- Satzfolge
- Paragraf
- Folge von Paragrafen

Ein *Pattern* ist eine Zeichenfolge, nach der gesucht wird. Der Standard unterstützt folgende Patterns:

- Wort, Wort mit Wildcards, Wortstamm
- Phrase (= geordnete Folge von Wörtern, wie zuvor definiert), Phrase mit Wildcards, Phrasen aus Wortstämmen
- eine Menge von Wörtern und Phrasen (ungeordnet)
- Patterns, die durch boolesche Operationen (`&`, `|`, `not`) verknüpft werden
- eine Menge von Patterns
- Nachbarschafts-Pattern (`NEAR`)
- diverse Arten von Thesaurus-Pattern (Synonymwörterbuch-Suche)
- Phoentische Pattern („klingt wie“)
- Fuzzy Pattern (unscharfe Suche)

Implizit ist jedem Wort eine Sprache zugeordnet (für die Wortstammbildung).

Beispiele

Wort:	’ "Datenbank" ’ ’ "Häuser" ’
Wort mit Widcard:	’ "Daten%" ’
Phrase:	’ "Datenbank Technologie" ’
Phrase mit Wildcard:	’ "SQL 200% Standard" ’ ’ "SQL % Standard" ’
Phrasen-/Wörtermenge:	’ ("Datenbank Technologie", "WWW") ’
Wortstamm:	’ STEMMED FORM OF GERMAN "Häuser" ’ = ’ ("Haus", "Häuser", "Hauses") ’
Phrasenstamm:	’ STEMMED FORM OF "grünes Haus" ’ = ’ ("grünes Haus", "grüne Häuser", ...) ’ (GERMAN ist Default)
Boolesche Operatoren:	’ "Wolfgang" & "Kowarschi%k" & not("MMProg") ’
Sprachfestlegung:	’ ENGLISH "die" & GERMAN "Würfel" ’
Nachbarschaft:	’ "Datenbank" IN THE SAME SENTENCE AS "Web" ANY ORDER ’ ’ "Datenbank" NEAR "Web" WITHIN 2 SENTENCES IN ORDER ’
Thesaurus:	’ THESAURUS "computer science" EXPAND SYNONYM TERM OF "list" ’ = Synonym-Suche für list: array, sequence, ... ’ THESAURUS "computer science" EXPAND SYNONYM TERM OF "rule of thumb" ’ = Synonym-Suche für rule of thumb: heuristics
Phonetisches Pattern:	’ SOUNDS LIKE "ganz" ’
Fuzzy Pattern:	’ FUZZY FORM OF "Hauptbahnhof" ’

Stammbildung

Unter *Stammbildung* (*stemming*) versteht man die Reduktion eines Wortes auf ihren Wortstamm. Um dies zu erreichen, muss für jede unterstütze Sprache eine passende Abbildung $w_{\text{Wortstamm,<}\textit{Sprache}\text{>}} : \texttt{Wort} \rightarrow \texttt{Wort}$ zur Verfügung stehen.

Zum Beispiel würde $w_{\text{Wortstamm,deutsch}}$ folgende Abbildungen vornehmen:

geht, ging, gegangen, gehend . . . → gehen

Datenbank, Datenbanken → Datenbank
Haus, Häuser, Hauses → Haus

Die Wortstammbildung ist allerdings nicht trivial und vor allem sprachabhängig. Zum Beispiel ist nicht klar, welchen Stamm das Wort „Essen“ hat: „Essen“ (Stadt in Deutschland), „essen“ (Substantivierung) oder „Esse“ (Pluralbildung). Um dies zu entscheiden, müsste man die Sätze, für deren Wörter die Wortstämme ermittelt werden sollen, semantisch (= inhaltlich) analysieren.

Zum Beispiel ist folgende Wortstammbildung nur möglich, wenn man den *Sinn* des Satzes versteht:

In den Essen in Essen ist das Essen super!
⇒
Wortstämme: *Esse, Essen, essen, super* (ohne Stop-Wörter)

Thesaurus

Ein *Thesaurus* ist ein Synonymwörterbuch. Bei der Thesaurussuche werden Synonyme automatisch berücksichtigt.

Beispiel für Synonyme: `kaufen`, `erstehen`, `käuflich erwerben`, ...

Dabei können auch Ober- bzw. Unterbegriffe berücksichtigt werden. Zum Beispiel ist `Säugetier` ein Oberbegriff für `Hund`, `Katze`, `Maus` etc.

Sehr oft werden fachspezifische Thesauri angelegt. Diese können auch Fachwörtersynonyme aus mehreren Sprachen enthalten, da Wissenschaftler Fachbegriffe häufig nicht übersetzen.

Beispeil „Mathematik“:

`Körper`	`field`	(nicht etwa: `body`)
`Verband`	`lattice`	(nicht etwa: `bandage`)
`Halbachse`	`Ellipsenhalbachse`	`semi axis`

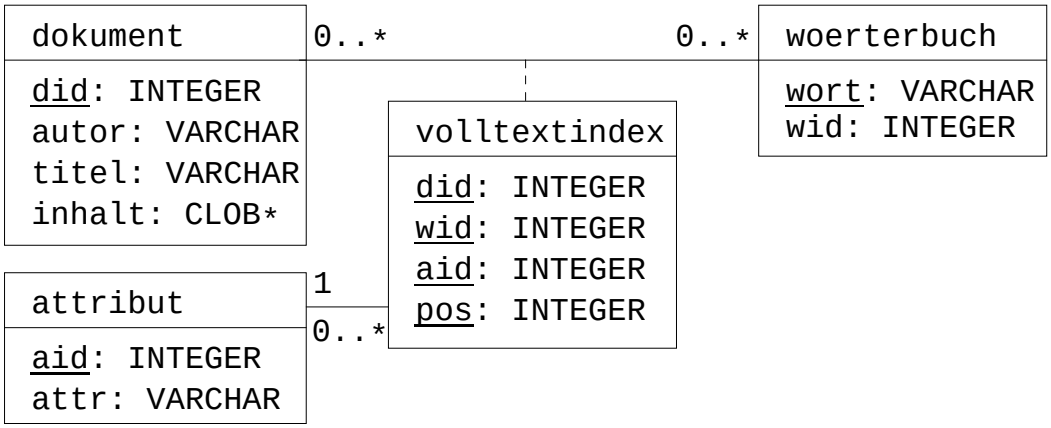
Aber auch Thesauri sind nicht gegen Fehltreffer gefeit. Wie soll ein allgemeiner Thesaurus, ohne die Suchabfrage semantisch zu analysieren, entscheiden, ob in einem Text *anschaffen* im Sinne von Kaufen und nicht im Sinne von *befehlen* oder gar im Sinne von *anschaffen gehen* verwendet wird.

4.5.3 Eine selbstprogrammierte Volltext-Suche

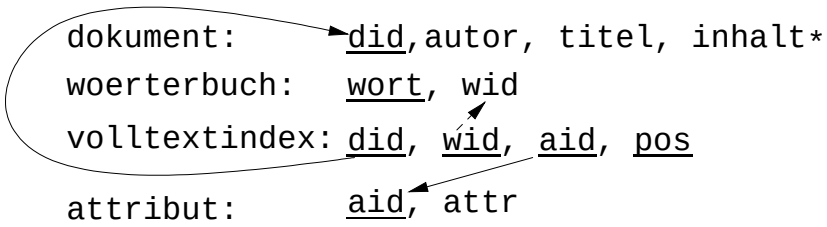
Es ist nicht sonderlich schwer eine Volltext-Suche selbst zu implementieren. Im Prinzip braucht man dafür nur zwei Tabellen: ein Wörterbuch und eine Index-Tabelle, die angibt, welche Wörter die einzelnen Texte enthalten. Wenn man im Index noch speichern will, in welchem Attribut ein Wort vorkommt, sollte man aus Effizienz noch eine dritte Tabelle einführen, die jedem Attribut einen eindeutigen Identifikator zuordnet.

Beispiel

Modell



Schema



Beispielsdaten

dokument

did	**autor**	**titel**	**inhalt**
1	'Wolfgang Kowarschick'	'Skript zur Vorlesung MMDB'	'Vorlesung MM-Datenbanksysteme im WiSe 2009/2010 ...'
2	'Wolfgnag Kowarschick'	'Skript zur Vorlesung MMProg'	'Vorlesung MM-Programmierung im WiSe 2009/2010 ...'

volltextindex

did	wid	attr	pos
1	1	1	1
1	2	1	2
1	3	2	1
1	4	2	2
1	5	2	3
1	6	2	4
1	5	3	1
1	7	3	2
1	8	3	3
1	9	3	4
1	10	3	5
1	11	3	6
1	12	3	7
2	1	1	1
2	2	2	2
2	3	2	1
2	4	2	2
2	5	2	3
2	13	2	4
2	4	3	1
2	7	3	2
2	14	3	3
2	9	3	4
2	10	3	5
2	11	3	6
2	12	3	7

woerterbuch

wort	wid
'wolfgang'	1
'kowarschick'	2
'skript'	3
'zur'	4
'vorlesung'	5
'mmdb'	6
'mm'	7
'datenbanksysteme'	8
'im'	9
'wise'	10
'2009'	11
'2010'	12
'mmprog'	13
'programmierung'	14

attribut

attr	aid
'autor'	1
'titel'	2
'inhalt'	3

Die Tabelle `dokument` ist ein Beispiel für eine Tabelle mit Text-Attributen, für die ein Volltextindex angelegt werden soll.

Das Wörterbuch enthält alle Wörter, die im Index vorkommen können und ordnet jedem dieser Wörter einen Identifikator zu. Beachten Sie, dass zwei verschiedene Wörter denselben Identifikator haben können, d.h., dass unterschiedliche Wörter als Synonyme behandelt werden können.

Die Tabelle `attribut` ordnet jedem Attribut der Tabelle `dokument` einen Idetifikator zu. Genaugenommen handelt es sich hierbei um eine Schema-Information, d. h. um eine Meta-Information.

Im eigentlichen Index steht welches Wort (identifiziert durch seinen Idetifikator!) in welchem Attribut an welcher Position gespeichert ist. Diese Tabelle kann sehr umfangreich werden.

Anmerkung: Man könnte anstelle der absoluten Position `pos` oder auch zusätzlich dazu weitere Positionsinformationen speichern, wie `paragrafnummer`, `satznummer`, `wortnummer` etc.

Nun können beliebige, auch komplexere Volltextanfragen mit Hilfe von normalen SQL-Anfragen formuliert werden.

Beispiele

```
CREATE VIEW dokindex AS
SELECT did, aid, pos, wid,
       wort, attr, autor, titel, inhalt
FROM   dokument NATURAL JOIN volltextindex
                NATURAL JOIN woerterbuch
                NATURAL JOIN attribut
;
```

Finde alle Dokumente vom Autor `Kowarschick`:

```
SELECT did, autor, titel, inhalt
FROM   dokindex
WHERE  wort = 'kowarschick' AND attr = 'autor'
;
```

Finde alle Dokumente, die `Vorlesung` **oder** `Datenbank%` enthalten:

```
SELECT did, autor, titel, inhalt
FROM   dokindex
WHERE  wort = 'vorlesung' OR wort LIKE 'datenbank%'
;
```

Finde alle Dokumente, die `Vorlesung` **und** `Datenbank%` enthalten:

```
SELECT DISTINCT d1.did, d1.autor, d1.titel, d1.inhalt
FROM   dokindex d1, dokindex d2
WHERE  d1.did = d2.did AND
       d1.wort = 'vorlesung' AND d2.wort LIKE 'datenbank%'
;
```

Finde alle Dokumente, die `Vorlesung` und `Kowa%`, aber nicht `MMProg` enthalten.

```
SELECT DISTINCT d1.did, d1.autor, d1.titel, d1.inhalt
FROM   dokindex d1, dokindex d2
WHERE  d1.did = d2.did AND
       d1.wort = 'vorlesung' AND d2.wort LIKE 'kowa%' AND
       NOT EXISTS
       (SELECT *
        FROM   dokindex d3
        WHERE  d1.did = d3.did AND d3.wort = 'mmprog'
       )
;
```

Finde alle Dokumente, die `Michael Lutz` (als Phrase) enthalten:

```
SELECT DISTINCT d1.did, d1.autor, d1.titel, d1.inhalt
FROM   dokindex d1, dokindex d2
WHERE  d1.did = d2.did AND d1.aid = d2.aid AND
       d1.wort = 'michael' AND d2.wort LIKE 'lutz' AND
       d1.pos+1 = d2.pos
;
```

Finde alle Dokumente, die `MM % % WiSe` (als Phrase) enthalten:

```
SELECT DISTINCT d1.did, d1.autor, d1.titel, d1.inhalt
FROM   dokindex d1, dokindex d2
WHERE  d1.did = d2.did AND d1.aid = d2.aid AND
       d1.wort = 'mm' AND d2.wort LIKE 'wise' AND
       d1.pos+3 = d2.pos
;
```

Häufigskeitsverteilung der Wörter:

```
SELECT   wid, wort, COUNT(*) as anzahl
FROM     dokindex
GROUP BY wid, wort
ORDER BY anzahl DESC, wort ASC
;

SELECT   aid, attr, wid, wort, COUNT(*) as anzahl
FROM     dokindex
GROUP BY aid, attr, wid, wort
ORDER BY aid ASC, anzahl DESC, wort ASC
;
```

4.5.4 Volltextsuche in PostgreSQL

Seit Version 8.3 bietet PostgreSQL eine proprietäre Volltextsuche an. Dabei werden Wortsuche, Endtrunkierung (seit 8.4), Wortstamm-Suche, sowie Synonym- und Thesaurus-Suche unterstützt. Ein Thesaurus kann nicht nur zwischen Wortsynonymen, sondern auch zwischen Phrasensynonymen unterscheiden. Außerdem ist es möglich Ober- und Unterbegriffe zu definieren.

Eine Phrasensuche wird eigenartigerweise bislang nicht unterstützt.

In PostgreSQL ist es geschickt, für jeden Index ein zugehöriges Attribut zu erzeugen:

```
ALTER TABLE dokument ADD COLUMN fulltext TSVECTOR;

UPDATE dokument
SET fulltext =
    to_tsvector('german',
                autor || ' ' ||
                titel || ' ' ||
                COALESCE(inhalt, '')
               )
;
```

Dem neuen Attribut `fulltext` wird der proprietäre Datentyp `TSVECTOR` zugeordnet. Ein `TSVECTOR` ist eine Liste von Wort/Positions-Paaren:

```
lutz':6 'rolf':3 'soch':4 'michael':5 'martin':2 ...
```

So ein Vektor darf höchstens 1 MByte groß sein und nicht mehr als 16.384 Wort/Positionspare enthalten.

Derartige Paare können mit Hilfe der proprietären Funktion `to_tsvector` aus beliebigen Zeichenketten erzeugt werden. Als erstes Argument wird `to_tsvector` i. Allg. eine so genannte *text search configuration* übergegeben, die festlegt, wie die Wörter des zweiten Arguments zu modifizieren sind, bevor sie in den Vektor eingefügt werden.

```
SELECT to_tsvector('simple', autor) FROM dokument
=>
lutz':6 'rolf':3 'socher':4 'michael':5 'märtin':2 ...

SELECT to_tsvector('german', autor) FROM dokument
=>
lutz':6 'rolf':3 'soch':4 'michael':5 'martin':2 ...
```

Mit `simple` gibt man an, dass Groß- in Kleinbuchstaben umgewandelt werden sollen, wohingegen `german` zusätzlich erzwingt, dass typische deutsche Endungen entfern sowie Umlaute modifiziert werden. Man kann auf eine ganze Anzahl von vordefinierten Konfigurationen zugreifen sowie eigene Konfigurationen defineren, wobei man auch Synoymwörterbücher und Thesauri einbinden kann.

Im zweiten Argument von `to_tsvector` muss derjenige String übergeben werden, der in einen Vektor transformiert werden soll. Im obigen Beispiel werden die drei Attribute `autor`, `titel` und `inhalt` zu einem String verküpft (konkateniert). Damit das Ergebnis nicht `NULL` wird, wenn der Inhalt `NULL` sein sollte, wird der `COALESCE`-Operator eingesetzt:

```
autor || ' ' || titel || ' ' || COALESCE(inhalt, '')
```

Mit dem neuen Attribut *fulltext* und den darin gespeicherten Vektordaten wäre eine Volltextsuche schon möglich. Allerdings sollte man diese Suche mit einem speziellen Volltext-Index beschleunigen:

```
CREATE INDEX fulltext_index
ON dokument USING gin(fulltext);
```

Von PostgreSQL werden zwei Arten von Volltext-Indexen unterstützt:

GiST: fehlerhate Matches sind wegen des Hashings möglich ($\Rightarrow$ Ergebnis sollte überprüft werden)

GIN: keine fehlerhaftem Matches, weniger performant, Gewichtungen werden nicht berücksichtigt

Als letztes sollte man einen Trigger definieren, der das Volltextattribut bei jedem `INSERT` oder `UPDATE` eines Tuppels automatisch berechnet bzw. aktualisiert.

Dafür steht die proprietäre Funktion `tsvector_update_trigger` zur Verfügung. Man kann eine entsprechende Funktion auch selbst definieren.

```
CREATE TRIGGER fulltext_update
BEFORE INSERT OR UPDATE
ON dokument
FOR EACH ROW EXECUTE PROCEDURE
  tsvector_update_trigger
    (fulltext, 'pg_catalog.german', autor, titel, inhalt);
;
```

Für Volltext-Abfragen gibt es den nicht-standard-konformen Operator `@@`, der links einen `TSVECTOR` und rechts eine Suchanfrage (`TSQUERY`) erwartet. Das heißt, der linke Wert muss mit Hilfe von `to_vector` erzeugt werden und der rechte mit der Funktion `to_query`. In beiden Fällen muss dieselbe *text search*

configuration verwendet werden! Das Ergebnis dieses Operators ist `TRUE`, genau dann wenn die Anfrage mit dem Vektor matcht.

Für die folgenden Beispiele (siehe auch `http://mmdb.hs-augsburg.de/beispiel/`) gebe es zwei Volltextindex-Attribute: das oben definierte Attribut `fulltext` sowie das Attribut `fulltext_authorr`, das für das Attribut `author` einen Index der Art `simple` enthält.

Finde alle Dokumente vom Autor `Kowarschick`:

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  fulltext_autor @@
         to_tsquery('simple', 'Kowarschick')
;
```

Finde alle Dokumente, die `Vorlesung` **oder** `Datenbank%` enthalten:

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  fulltext @@
         to_tsquery('german', 'Vorlesung | Datenbank:*')
;
```

Finde alle Dokumente, die `Vorlesung` **und** `Datenbank%` enthalten:

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  fulltext @@
         to_tsquery('german', 'Vorlesung & Datenbank:*')
;
```

Finde alle Dokumente, die `Vorlesung` und `Kowa%`, aber nicht `MMProg` enthalten.

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  fulltext @@
         to_tsquery('german',
                    'Vorlesung & Kowa:* & !MMProg')
;
```

Finde alle Dokumente, die `Michael Lutz` (als Phrase) enthalten:

Funktioniert derzeit nicht!

Finde alle Dokumente, die `MM % % WiSe` (als Phrase) enthalten:

Funktioniert erst recht nicht!

4.5.5 Volltextsuche in TransBase

TransBase unterstützt die Volltexttextsuche auf Attributen vom Typ `CHAR(`n`)` (= `VARCHAR(`n`)`), `CHAR(*)` (= `VARCHAR(4096)`) und `BLOB`.

Für jedes Attribute, das zur Volltextsuche eingesetzt werden soll, muss zunächst ein Volltextindex erzeugt werden.

```
CREATE [POSITIONAL] FULLTEXT INDEX <name> <options>
ON <table> (<attribute>)
```

- Positionelle Volltextindexe, d. h., Volltextindexe, die zu jedem Wort auch noch die Wortposition abspeichern, sind umfangreicher als einfache Volltextindexe, ermöglichen aber Phrasensuche.
- Die Indexerstellung kann durch die Angabe diverser Optionen gesteuert werden:
  - `WORDLIST FROM` <*table w*>
    Nur Wörter aus der Tabelle <*table w*> indexieren (Lexikon).
  - `STOP WORDS FROM` <*table s*>
    Wörter aus Tabelle <*table s*> nicht indexieren (z. B. der, die, das, ein, einer ...). Gerade die Phrasensuche profitiert von der Stop-Word-Tabelle.
  - `CHARMAP FROM` <*table c*>
    Buchstaben ersetzen (A, B, C ... durch a, b, c ...; ä, ö, ü, ß durch ae, oe, ue, ss etc.). Die Tabelle <*table c*> hat zwei Spalten: `char(1)` und `char(*)`.
  - `DELIMITERS NONALPHANUM`, `DELIMITERS FROM` <*table d*>
    Standardmäßig werden Wörter durch Whitespaces voneinander getrennt. Das heißt, Sonderzeichen zählen als Buchstaben. Man kann aber auch alle Sonderzeichen (d. h. Zeichen, die keine Zahlen oder Buchstaben sind) als Worttrenner behandeln (`DELIMITERS NONALPHANUM`). Oder man kann eine eigene Tabelle mit Worttrennsymbolen definieren (`DELIMITERS FROM` <*table d*>).

Beispiel

```
CREATE POSITIONAL FULLTEXT INDEX volltext
ON dokument (autor);
CREATE POSITIONAL FULLTEXT INDEX volltext
ON dokument (titel);
CREATE POSITIONAL FULLTEXT INDEX volltext
ON dokument (inhalt);
```

Für jeden Volltextindex <*f_index*> gibt es Pseudotabellen:

- `FULLTEXT WORDLIST OF <`*f_index*`> (word, wno)`
- `FULLTEXT STOPWORDS OF <`*f_index*`> (word)`
- `FULLTEXT CHARMAP OF <`*f_index*`> (source, target)`
- `FULLTEXT DELIMITERS OF <`*f_index*`> (delimword)`

Diese können wie normale Tabellen in `SELECT`-Statements verwendet werden. Außerdem sind folgende Modifikationen erlaubt: `INSERT` für `WORDLIST` und `DELETE` für `STOPWORDS`. Dies hat allerdings nur einen Einfluss auf zukünftige Volltextindex-Updates (durch Inserts in die Basistabelle), nicht aber auf den aktuellen Index-Inhalt.

Wichtiger ist jedoch die Volltextsuche selbst. In TransBase gibt es dafür das (nicht-standardkonforme) Spezialprädikat `CONTAINS`.

Beispiele

Wortsuche

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  autor
       CONTAINS ('Kowarschick')
```

Phrasensuche

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  autor
       CONTAINS ('Michael' 'Lutz')
```

Wildcardsuche

% und _ haben dieselbe Bedeutung wie im Prädikat `LIKE`.

Die Suche mit Mitten- und/oder Endtrunkierung (`Wasch%`, `Wasch%en`, `Wasch%en%`) funktioniert im Gegensatz zu `LIKE` für jedes Wort im Text, und zwar genauso effizient wie bei `LIKE`. Lediglich die Suche mit Anfangstrunkierung (`%lappen`) ist ebenso wie bei `LIKE` langsam (siehe aber Abschnitt 4.5.7).

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  autor
       CONTAINS ('datenbank%')
```

Wenn man nach einem %, \ oder _ sucht, muss man \%, \\ bzw. _ schreiben.

Wortabstand

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  autor
       CONTAINS ('Vorlesung' [0,3] 'WiSe')
```

Hier wird das Wort `Vorlesung`, gefolgt von bis zu drei beliebigen Wörtern, gefolgt vom Wort `WiSe` gesucht. Die Schreibweise `CONTAINS ('...' [`n`] '...')` ist eine Abkürzung für `CONTAINS ('...' [0,`n`] '...')`.

Volltextsuche mit Boolschen Operatoren

Einzelne Phrasen können mit `AND`, `OR`, `NOT` und Klammerung verknüpft werden.

```
SELECT did, autor, titel, inhalt
FROM   dokument
WHERE  inhalt
       CONTAINS (   'Objekt%' [1] '%systeme'
                 OR 'Verteilte' 'Datenbanksysteme'
                )
```

Multivalue-Atome

Suche alle Inhalte, die irgendein Wort aus der Tabelle `wortliste` enthalten.

```
SELECT did, autor, titel, inhalt
WHERE  inhalt
       CONTAINS (ANY (SELECT * FROM wortliste))
```

Nicht-primitive `CONTAINS`-Atome

Das `CONTAINS`-Prädikat darf auch Attributzugriffe und sogar Unteranfragen enthalten.

Welche Wörter aus der Liste `wortliste` kommen in meinen Abstracts vor?

```
SELECT w.wort FROM wortliste w
WHERE EXISTS
      (SELECT * FROM dokument
       WHERE inhalt CONTAINS(w.wort)
      )
```

Performanz

Die Volltextsuche ist i. Allg. sehr schnell (Ausnahme: Anfangstrunkierung). Das Einfügen neuer Tupel in eine Tabelle mit Volltextindex kann schnell erfolgen. Allerdings wird für die Erzeugung eines ganzen Indexes viel Platz benötigt. Je mehr temporärer Speicher zur Verfügung steht desto besser. Man kann den temporären Speicherplatz bei der Erzeugung des Indexes angeben: `SCRATCH 60 MB`.

Das Löschen ist dagegen *sehr* langsam ($O(n)$, wobei n die Größe des Indexes ist). Es ist besser, viele Tupel auf einmal zu löschen als einzeln hintereinander.

4.5.6 Fehlertolerante Suche

Eine gute Volltextsuche sollte fehlertolerant sein. Insbesondere sollte sie i. Allg. unabhängig von Groß- und Kleinschreibung und spezieller Schreibung der Umlaute funktionieren. Das heißt, man sollte eine ein so genannte Character Map verwenden, die einzelne Buchstaben auf Ersatzzeichen abbilden:

source	target
A	a
B	b
C	c
...	
Ä	ae
Ö	oe
Ü	ue
ä	ae
ö	oe
ü	ue
ß	ss

In TransBase kann son eine Character Map einem Volltext-Index direkt zugeordnet werden, in PostgreSQL muss man einen entsprechenden Parser selbst schreiben (`CREATE TEXT SEARCH PARSER`). Für den handgestrickten Volltextindex sollte man eine entsprechende Map in der Datenbank ablegen, muss aber den zugehörigen Parser außerhalb (z.B. in Java oder PHP) schreiben.

Diese Map muss sowohl bei der Indexerstellung als auch bei der Auswertung von Anfragen verwendet werden. Damit erreicht man, dass beispielsweise die SQL3-Anfrage `CONTAINS('Mädchen') = 1` gleichwertig zur Anfrage `CONTAINS('maedchen') = 1` ist.

Prinzipiell gilt, dass man mit Hilfe von Abbildungen w : Wort $\rightarrow$ Wort, die jedes Wort in ein anderes Wort transformieren (z. B. $w_{\texttt{charmap}}(\texttt{'Mädchen'}) = \texttt{'maedchen'}$), eine große Klasse unterschiedlicher Such-Algortihmen beschreiben kann: Phonetische Suche, Thesaurussuche etc. Diese Abbildung muss sowohl

bei der Erzeugung des zugehörigen Volltextindexes als auch bei der Erzeugung der Anfrage angewendet werden:

1. Bei der Erstellung des Volltextindexes wird jedes Wort mittels w transformiert, bevor es gespeichert wird.
2. Jedes Wort einer Anfrage wird ebenfalls mittels w transformiert.

Beispiel

Mehrere phonetisch ähnlich klingende Laute können durch einen Laut ersetzt und stumme Laute können entfernt werden.

source	target
'f'	'ph'
'F'	'ph'
'p'	'b'
'P'	'b'
'c'	'k'
'h'	''
'z'	's'
'ie'	'i'
'ck'	'k'
'tz'	'z'
'dt'	't'
'tt'	't'
⋮	

In diesem Fall werden auch fehlerhafte Schreibwesen in einer Anfrage akzeptiert: `Haubtbanhof`, `Fotograph`, `Viedeo` etc.

Allerdings werden auch mehr Fehltreffer gefunden. Die Frage nach `Haus` liefert auch das Wörtchen `aus` (außer man lässt nur das klein geschriebene `h` entfallen).

Wenn das verwendete DBMS derartige allgemeine Wort-Transformationen nicht unterstützt, so kann man diese Transformationen außerhalb des DBMS mit Hilfe von geeignete C/C++/JAVA/...-Routinen erledigen (siehe z. B. Michael [1999]). Man muss nur darauf achten, dass jeder Volltext und jede Anfrage mit diesen Routinen transformiert wird, *bevor* die Daten an das Datenbanksystem weitergereicht werden:

1. Wende die Routine auf alle Texte an, die volltextindiziert werden sollen und speichere die transformierten Texte (evtl. zusätzlich zu den Originaltexten) in einer Tabelle ab:

```
CREATE TABLE dokument
(id            INTEGER,
 text          CLOB,
 transformtext FULLTEXT
 ⋮
);
```

2. Erzeuge einen Volltextindex auf den *transformierten Texten*.
3. Jede Benutzerangabe wird mit Hilfe der C/C++/JAVA/...-Routine transformiert, bevor sie an die Datenbank wietergeleitet wird.
4. Optional: Jedes Wort eines Treffertextes wird wiederum mit der obigen Routine transformiert (allerdings nur temporär). Jedes Wort, dessen transformierte Form der transformierten Anfrage genügt, wird als Treffer markiert (z. B. in HTML farbig hervorgehoben).

Beispiel

Text 1: Dies kann man lesen

Text 2: Diesen Kahn fährt ein Mann

Trafo 1: dis kan man lesen

Trafo 2: disen kan faert ein man

Die Suche erfolgt nun viel fehlertoleranter als bislang. Es werden aber auch viel mehr falsche Dokumente gefunden. Zum Beispiel liefert die Suche nach „Mann“ und „Kahn“ auch Text 1.

Weitere Möglichkeiten: alle Vokale weglassen, Wörter durch phonetische Äquivalenzen ersetzen (dazu braucht man ein phonetisches Lexikon, z. B. klingen „bot“ und „Boot“ gleich, nicht aber „kann“ und „Kahn“) etc.

4.5.7 Anfangstrunkierung: Suche nach Wortendungen

Ein schon angesprochenes Problem, die effiziente Suche nach Wortendungen wie `%bank` (Anfangstrunkierung), kann man relativ einfach lösen, indem man das Wörterbuch um spezielle Einträge erweiter.

Sehen wir uns noch einmal das Wörterbuch an. Jedem Wort wird ein Identifikator zugeordnet:

woerterbuch

wort	**wid**
datenbank	1
gehen	2
gehoben	5
geht	4
spielbank	3

Die eigentliche Suche erfolgt mit den entsprechenden Identifikatoren, das heißt, jedes Wort einer Anfrage wird durch den zugehörigen Identifikator ersetzt. Die Suche nach Wortanfängen wie `daten%` (Endtrunkierung) kann mit Hilfe von `LIKE` sehr effizient durchgeführt werden, wenn diese Liste alphabetisch geordnet ist, d. h., wenn `wort` als Primärschlüssel gewählt wurde. Man muss nur das erste Wort in der Wortliste suchen, das mit dem gesuchten Teilwort beginnt und dann der Reihe nach alle nachfolgenden Wörter mit demselben Anfang ebenfalls in die Suche einbeziehen (`geh%` → Wörter `2`, `5`, `4`).

Bei der Anfangstrunkierung bleibt einem zunächst nichts weiter übrig, als die ganze Wortliste zu durchsuchen (`%bank` → `1`, `3`).

Der Trick ist nun, die Liste zu erweitern, indem man jedes Wort noch einmal rückwärts geschrieben und geeignet markiert mit demselben Identifikator wie das Ursprungswort in die Liste einfügt.

woerterbuch

wort	**wid**
@knableips	3
@knabnetad	1
@neboheg	5
@neheg	2
@theg	4
datenbank	1
gehen	2
gehoben	5
geht	4
spielbank	3

Wenn man jetzt die Anfrage `%bank` (natürlich automatisch) durch `@knab%` ersetzt, bekommt man die gesuchte Wortindexliste (`3, 1`) ganz schnell. Die einzigen Anfragen, die jetzt noch Schwierigkeiten machen, sind Anfragen mit Anfang- *und* Endtrunkierung: `%ab%` etc. oder gar `%` (ganz übel). Solche Anfragen sollte man herausfiltern und gar nicht erst an das DBMS weiterreichen.

4.5.8 Weitere Volltexttechniken

n-Gramme

Neben den Worttransformationen gibt es auch noch weitere Techniken, um fehlertolerante Volltextsuche zu ermöglichen.

Eine bekannte Technik besteht darin, die Wörter Texte in Trigramme (oder allegemein n-Gramme) zu zerlegen:

> Busbahnhof = ␣bu, bus, usb, sba, bah, ahn, hnh, nhn, hno, nof, of␣

Anfragen werden ebenfalls in n-Gramme zerlegt. Diejenigen Texte, in denen die meisten der gesuchten n-Gramme vorkommen werden als Treffer angezeigt.

Vorteile:

- Bcuhstabenverdreher (sic.) können auch noch zu einem Ergebnis führen (nur ein paar n-Gramme sind falsch).
- Zusammen- und Getrenntschreibung führt zwar zu anderen, aber meist noch ausreichend vielen übereinstimmenden n-Grammen („Bahnhof für Busse“, gemeinsame Trigramme: bah, ahn, hnh, nhn, hno, nof, of␣, ␣bu, bus).

Nachteile:

- Kleine n-Gramme liefern viele Fehltreffer (Jedes Treffer-n-Gramm kann in einem anderen Wort vorkommen.)
- Große n-Gramme sind nicht sehr fehlertolerant.

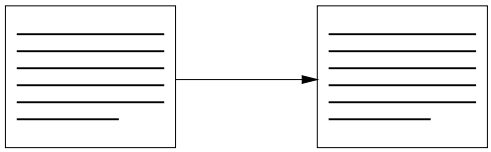
Fazit: Je fehlertoleranter eine Volltextsuche ist, desto mehr Fehltreffer liefert eine Suche.

4.6 Hypermedia (navigierende Suche)

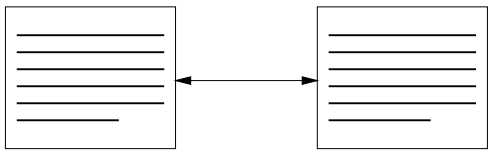
Ein Link hat die Aufgabe, mehrere *Dokumente* (d. h. Texte oder andere Multimedia-Objekte) oder Teildokumente in Beziehung zueinander zu setzen.

Man unterscheidet zwischen:

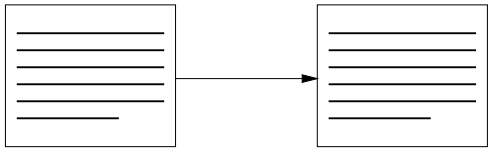
1. Richtung der Links
   - unidirektionale Links, die nur in Richtung von Linkquelle zu Linkziel verfolgt werden können (z. B. WWW)



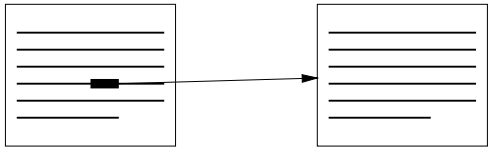
   - bidirektionale Links, die in beide Richtungen verfolgt werden können



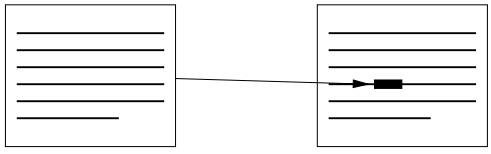
2. Granularität der Links
   - Node-to-Node-Links: die Linkquelle und das Linkziel sind vollständige Dokumente (in HTML vorgesehen, von den meisten Browsern aber nicht unterstützt)



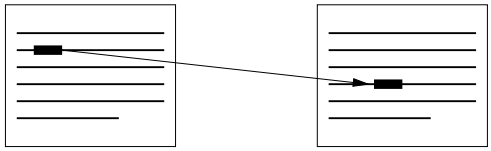
   - Span-to-Node-Links: die Linkquelle ist ein Teildokument, das Linkziel ein vollständiges Dokument (z. B. WWW)



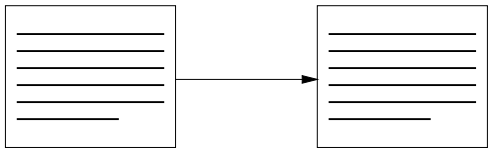
   - Node-to-Span-Links: Linkquelle ist ein vollständiges Dokument, Linkziel ein Teildokument (in der Praxis sehr selten)



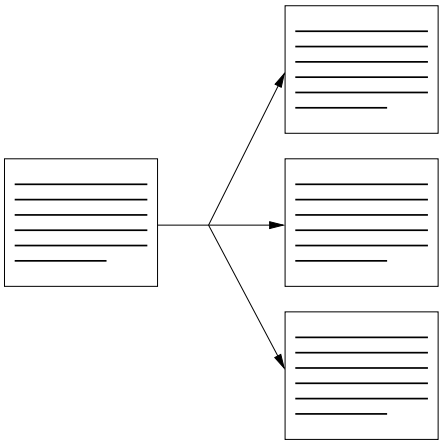
- Span-to-Span-Links: Linkquelle und Linkziel sind Teildokumente (in WWW eingeschränkt vorhanden; HTML-Links können auf Tags innerhalb eines Dokuments zeigen; angezeigt wird allerdings immer das gesamte Dokument; es wird lediglich der Scrollbar der Seite verschoben)



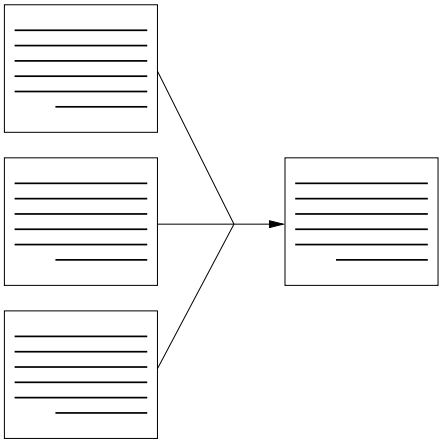
3. Kardinalität der Links
   - 1:1-Links: klassisch, z. B. WWW



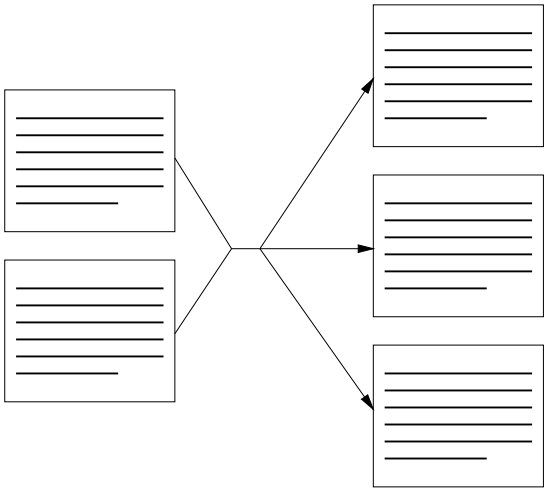
   - 1:n-Links: von einer Quelle aus können mehrere Ziele erreicht werden



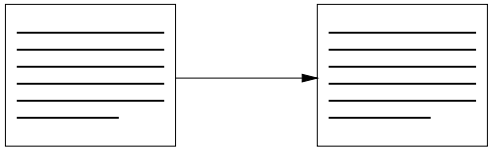
   - n:1-Links: von mehreren Quellen aus kann ein Ziel erreicht werden



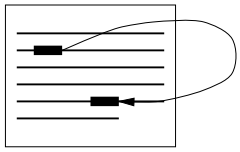
- m:n-Links: von mehreren Quellen aus können mehrere Ziele erreicht werden (auch MSMD-Links genannt: Multi-Source-Multi-Destination-Links)



4. Lokalität der Links
   - Inter-Dokument-Link



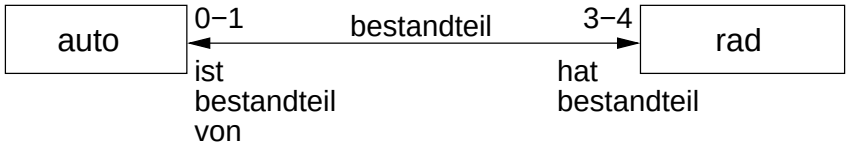
   - Intra-Dokument-Link



   - Inter-Domain-Links (Domain = Rechner, Lan etc.)
   - Intra-Domain-Links
   - absolute Links
   - relative Links

Man beachte, dass alle Kombinationen aller Linkarten möglich sind: bidirektionale Span-To-Node-m:n-Links etc. Bei 1:n- und m:n-Links sind sogar noch zusätzliche Span- und Nodevarianten denkbar.

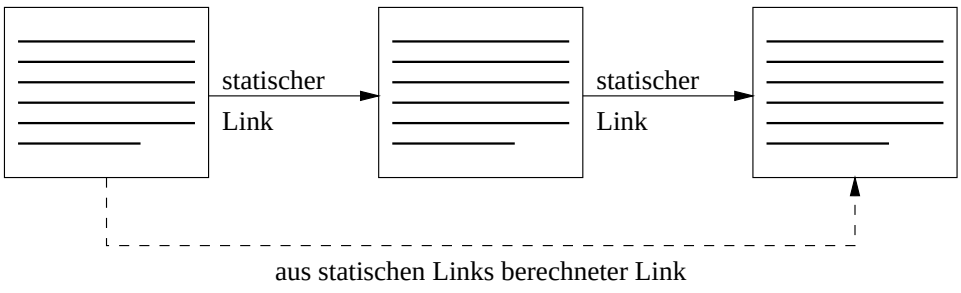
Links können außerdem benannt sein und beliebige andere Attribute enthalten, wie z. B. ein Attribut `relevanz`, das einen Wert zwischen unrelevant (`=0`) und sehr relevant (`1`) annehmen kann. Links können außerdem zusätzliche Attribute (wie z. B. `haeufigkeit`) und/oder Rollennamen an den jeweiligen Linkenden enthalten.



Die möglichen Eigenschaften von Links verallgemeinern die bereits bekannten Eigenschaften von ER- oder objektorientierten Modellen. Und tatsächlich können Links in RDBS ebenso wie Relationships entweder als Attribute oder durch eigene (Link-)Tabellen realisiert werden: m:n-Links mit zusätzlichen Attributen werden in separaten Tabellen gespeichert, unidirektionale 1:1-Links können dagegen als zusätzliche Attribute in der Tabelle der Quellenobjekte abgelegt werden, wenn die Anzahl der möglichen unidirektionalen Links, die von einem Objekt ausgehen können, beschränkt ist (ansonsten benötigt man ebenfalls eine eigene Linktabelle).

Span-Links können entweder – wie in HTML – in die Dokumente selbst eingefügt oder durch spezielle Positionsattribute realisiert werden: z. B. `link_start_pos` und `link_ende_pos`. Die zweite Technik hat den Vorteil, dass man Teildokumente nicht nur von Textdokumenten, sondern von beliebigen Multimedia-Dokumenten als Linkquelle und/oder Linkziel einsetzen kann: Teilvideo- oder Teilaudio-Sequenzen, clickable images etc. Man beachte: Es ist in HTML zwar nicht vorgesehen, dass sich mehrere Span-Links überlappen, aber dennoch ist dies – zumindest in anderen Hypermedia-Systemen – ohne Weiteres vorstellbar.

Links können nicht nur in Tabellen gespeichert, sondern auch berechnet werden (z. B. via JavaScript, Java oder PHP). So wäre es z. B. möglich, neue (implizit vorhandene Links) Links über mehrere Dokumente hinweg zu ermitteln.



4.6.1 Das Dexter-Referenz-Modell

1988 wurde im Dexter Inn, New Hampshire, USA das so genannte *Dexter Hypertext Referenz Modell* entwickelt (Halasz und Schwartz [1994]). Teilnehmer waren Entwickler verschiedener Hypertext-Systeme und verwandter Produkte, wie Intermedia, NoteCards, KMS etc.

Ziele

- Standardterminologie
- Referenzmodell zur formalisierten Beschreibung bestehender Hypertext-Systeme
- Basis für die Entwicklung von Austauschformaten

Die Ergebnisse waren nicht nur auf Hypertext-, sondern sogar auf beliebig Hypermedia-Systeme anwendbar.

Im Dextermodell werden drei Schichten unterschieden:

1. Runtime Layer
   Präsentationstools, Benutzer-Interaktion
2. Storage Layer
   enthält das Netzwerk aus Dokumenten und Links
3. Within-Component Layer
   enthält die Datenstrukturen und Inhalte der Dokumente

Der Within-Component Layer wird im Dexter-Modell nicht genauer spezifiziert, das Hauptaugenmerk liegt auf dem Storage Layer, d. h. dem eigentlichen Netzwerk.

Im Dexter-Modell sind Links eigenständige Objekte, die eine Liste von *Specifiern* enthalten: Jeder Specifier beschreibt einen Linkpfeil:

`UID` Dokumentidentifikator

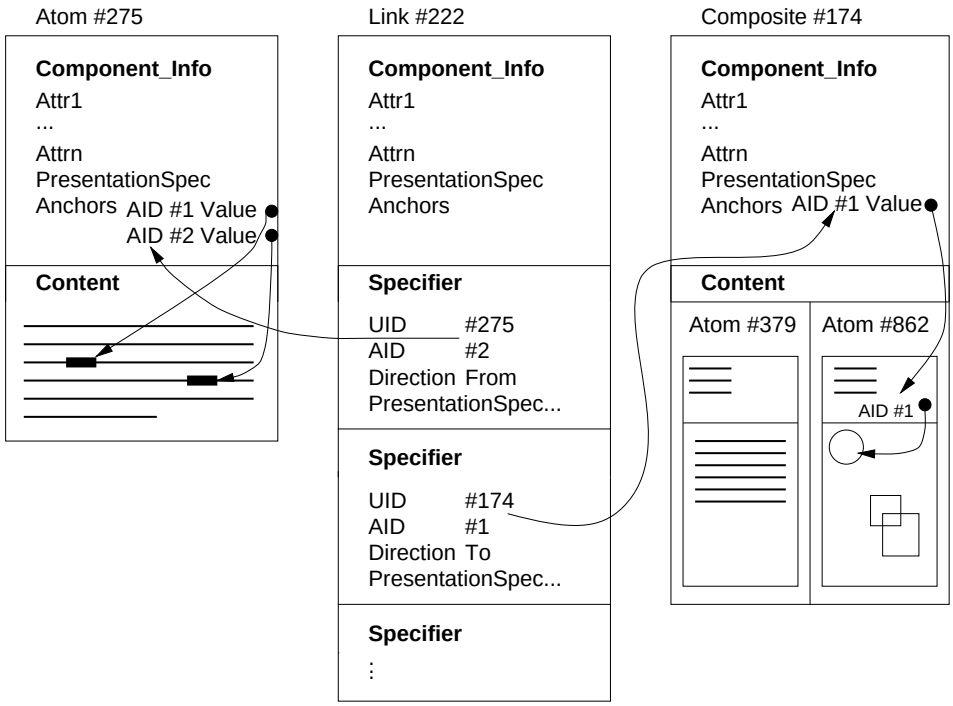
`AID` Anchoridentifikator im Dokument
(jeder Anchor beschreibt ein Teildokument)

`Direction` Flag, mit 4 möglichen Werten: `FROM`, `TO`, `BIDIRECT`, `NONE`

`presentation specification`
(optional) Information über Präsentationsart (für Runtime Layer)

Mit der Presentation Specification kann z. B. angegeben werden, ob eine Animation abgespielt werden (z. B. `student link`) oder mit einem passenden Programm bearbeitet, d. h. editiert werden (z. B. `teacher link`) soll.

Dokumente sind entweder primitiv (so genannte *atoms*) oder zusammengesetzte aus anderen Dokumenten (*composite objects*).



Vorteile des Dexter-Modells:

- 1:n-, m:n-Links
- bidirektionale Links
- zusammengesetzte Objekte
- sehr mächtige Resolvefunktionen zur Linkverfolgung (boolesche Operationen etc.; Ergebnis ist Anchormenge)
- keine *dangling references* (ein Link der weg ist, ist weg – in allen Dokumenten!)
- Links auf Links sind erlaubt (Wirklich sinnvoll?)

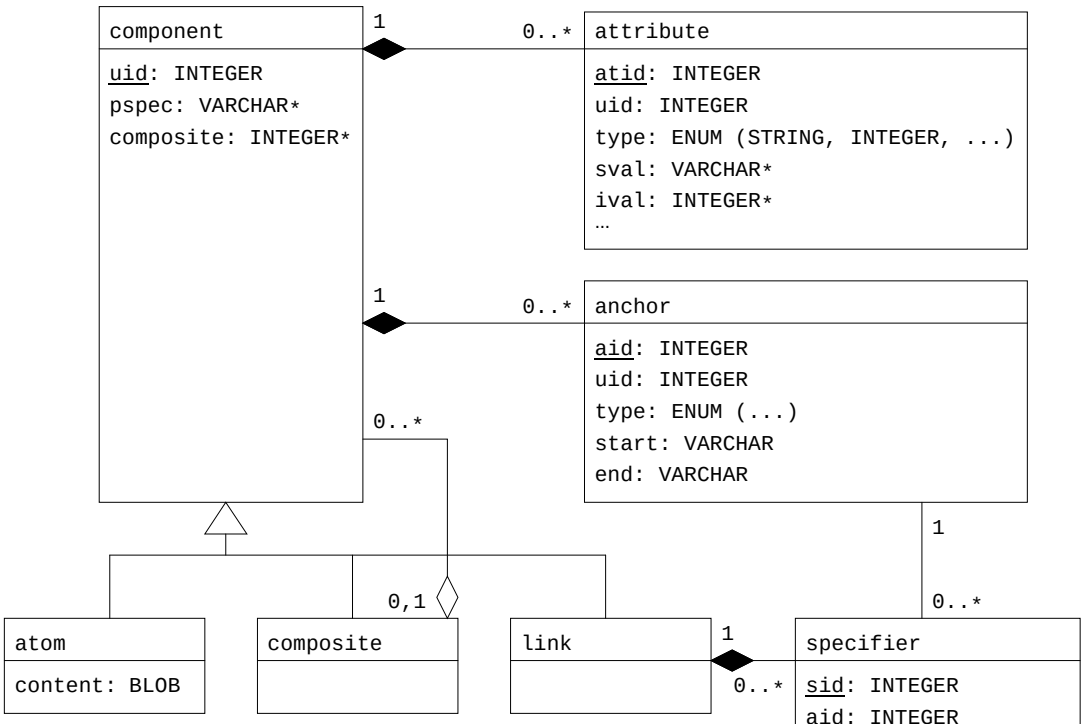
Nachteile des Dexter-Modells:

- keine Synchronisation von zeitkritischen Medien (Videos, Audio)
- der Within Component Layer wurde nicht spezifiziert (medienabhängige Datenmodelle fehlen)

Realisierung

Links, die dem Dexter-Modell genügen, lassen sich mit Hilfe einer geeigneten Datenbank relativ problemlos realisieren. Das folgende einfache Datenmodell bildet

die wesentlichen Eigenschaften des Dexter-Modells direkt ab:



Weitere Atom-Tabellen für
Texte (CLOB), XML etc.
sind denkbar.

Achtung: Für die Behandlung vom Composites werden rekursive Anfragen benötigt.

Natürlich kann dieses Modell in vielerlei Hinsicht erweitert werden. So könnten z.B. Dokumenttypen definiert werden, die festlegen, welche Attribute ein Dokument mindestens haben muss. Oder man definiert Tabellen für die Verwaltung von Benutzerrechten. Et cetera pp.

Wozu braucht man eigentlich derart komplexe Links?

Aufgrund des World Wide Web haben wir uns so an unidirektionale Primitiv-Links gewöhnt, dass einem die Vorteile der Dexter-Links nicht sofort einleuchten.

Daher zähle ich hier einfach ein paar Vorteile auf:

- Es gibt keine Dangling References (wenn man die Foreign Keys des Datenmodells korrekt umsetzt).
- Ein Link kann automatisch auf mehrere uni-direktionale Links abgeblidet werden. Zum Beispiel wird, sobald einer Lehrperson eine Lehrveranstaltung zugeordnet wird, der Lehrveranstaltung automatisch auch die Lehrperson zugeordnet.
- Links können auf beliebige Medien bzw. Anker (wie z.B. eine Videoszene) verweisen.
- Links können von beliebigen Medien oder Ankern ausgehen.
- Man kann Multilinks realisieren: Bei einem Klick auf einen Link öffnen sich entweder mehrere Medien auf einmal oder eine Auswahlbox.
- Da man typisierte Links zur Verfügung hat (Presentation Specification), kann man Links abhängig vom Ausgabemedium und den Benutzerrechten präsentieren.

Das heißt: Dieses Modell (oder eine verbesserte Variante davon) ist ideal für die Realisierung von Content-Management-Systemen geeignet.

4.7 DVD-Datenbanken

Multimediadaten benötigen vor allem eines: Platz. So ist es nicht verwunderlich, dass billiger Speicherplatz ein begehrtes Datenbankmedium ist.

Den billigsten Speicherplatz war bis vor ein paar Jahren das Magnetband. Allerdings sind die Zugriffszeiten – wegen der linearen Suche – meist indiskutabel langsam. So eignen sich Bänder nur für wenige Anwendungsgebiete wie z. B. Video on Demand (ein Film = ein Blob pro Band). Das jeweils gesuchte Band kann dabei von Robotern eingelegt werden $\Rightarrow$ Datenbanken im Petabytebereich.

Ein anderes sehr billiges Speichermedium ist die DVD. Auch hier können viele DVDs (oder auch Blue Rays) von einer so genannten Jukebox verwaltet werden (Platz für mehrere 100 DVDs sowie ein oder mehrere DVD-Lesegeräte). Jede DVD ist dann in einem von drei Zuständen:

online: DVD ist im Lesegerät

nearline: DVD ist in der Jukebox

offline: DVD ist außerhalb de Jukebox und wird bei Bedarf vom Operator eingelegt

Der Zugriff auf Online-DVDs dauert einige Millisekunden, der Zugriff auf Nearline-DVDs einige Sekunden und der Zugriffauf Offline-DVDs mehrere Minuten (wenn's gut geht).

Um teure Nearline-Zugriffe zu vermeiden, sollten Daten, die häufig gemeinsam benötigt werden, nach Möglichkeit auf einer DVD untergebrachte werden.

TransBase bietet beispielsweise die Möglichkeit DVD-Datenbanken zu erstellen (TransBase CD [2012]):

1. Entwicklung einer Datenbank wie gewohnt.
2. Erzeugung der späteren DVD-Datenbank zunächst auf Festplatte (Testphase).
3. Brennen der DVD.
4. Einrichten der DVD-Datenbank
5. Benutzen der Datenbank mit zweistufigem Cache: Hauptspeicher und Festplatte. Es ist auch möglich, Daten auf der DVD zu „überschreiben". Dazu werden die neueren Tupel auf der Festplatte abgelegt und die alten Tupel verschattet (auf der Festplatte als ungültig markiert).

Kapitel 5

Normalformtheorie

(siehe Ullman [1988] oder ein beliebiges anderes Standardwerk über DBMS)

5.1 Funktionale Abhängigkeit

Es sei $R(a_1, \ldots, a_n)$ ein Relationenschema R mit den Attributen $A = \{a_1, \ldots, a_n\}$.

Eine Attributmenge $C \subseteq A$ heißt *funktional abhängig* (*functional dependent*) von einer Attributmenge $B \subseteq A$, in Zeichen $B \rightarrow C$, wenn gleiche Attributwerte für B auch gleiche Attributwerte für C erzwingen, d. h., wenn es zu **keinem Zeitpunkt** zwei Tupel geben kann, die in allen B-Attributen übereinstimmen, sich aber in irgendwelchen C-Attributen unterscheiden.

Beispiel

```
haendler
hid name    adresse       artikel   preis
1   'Mayer' 'Königsbrunn' 'PIII500' 500
1   ?       ?             'PIII600' 1200
```

Wenn der Name und die Adresse funktional vom Idenitfikator abhängen (`hid` → `name`, `adresse`), dann müssen die Wert von ? gleich `'Mayer'` bzw. `'Königsbrunn'` sein.

Bestimmte funktionale Abhängigkeiten können zu so genannten *Update*-Anomalien führen.

- Update-Anomalie
  Die Änderung der Adresse von `'Mayer'` (mit `hid=1`) muss in jedem Tupel geschehen.
- Insert-Anomalie
  Der Name und die Adresse von `'Mayer'` (mit `hid=1`) kann nicht erfasst werden, wenn `'Mayer'` (noch) nichts liefert.

– Delete-Anomalie
  Löscht man das gesamte Liefersortiment von `'Mayer'` (mit `hid=1`) – z. B. bei einer Produktumstellung –, verliert man seine Adresse.

Ziel der Normalformtheorie ist es, Anomalien, die durch *redundante* Speicherung von Fakten entstehen, zu vermeiden.

Im Folgenden werden wir uns auf Redundanzen, die sich durch funktionale Abhängigkeiten ergeben, beschränken.

5.1.1 Reduzierte funktionale Abhängigkeiten

Eine funktionale Abhängigkeit $B \rightarrow a$ heißt *reduziert*, wenn

1. a ein einzelnes Attribut ist
2. $a \notin B$ ($\ldots a \ldots \rightarrow a$ gilt nämlich immer)
3. Für keine echte Teilmenge $B' \subset B$ gilt $B' \rightarrow a$
   (a heißt in diesem Fall *voll funktional abhängig von* B)

Jede Menge von funktionalen Abhängigkeiten (kurz *FD-Menge*) kann in eine äquivalente (gleichwertige) Menge von reduzierten funktionalen Abhängigkeiten (kurz *rFD-Menge*) umgewandelt werden.

5.1.2 Minimale Überdeckung

Wenn in einer rFD-Menge keine funktionale Abhängigkeit aus anderen berechnet werden kann (wie in $A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$), spricht man von einer *minimalen Überdeckung*.

5.1.3 Schlüsselkandidaten

Eine Menge $S \subseteq A$ von Attributen heißt *Schlüsselkandidat* von R, wenn alle übrigen Attribute von R funktional von S abhängig sind und es keine echte Teilmenge $S' \subseteq S$ gibt, für die dies ebenfalls gilt.

Ein Attribut a, das zu keinem Schlüsselkandidaten gehört, heißt *Nichtschlüssel-Attribut*, alle anderen Attribute heißen *Schlüssel-Attribute*.

5.2 Normalformen

5.2.1 Erste Normalform (1NF)

Im Weiteren nehmen wir an, dass alle Attribute primitive Domänen (wie `BOOLEAN`, `VARCHAR`, `INTEGER` etc.) anstelle von komplexen Domänen (wie `SET`, `LIST`, `TUPLE` etc.) haben.

Wie bereits bekannt ist, spricht man in diesem Fall von erster Normalform (1NF). In RDBS ist diese Bedingung in der Regel erfüllt, in ODBS dagegen nicht.

5.2.2 Zweite Normalform (2NF)

Eine rFD-Menge $\mathcal{F}$ erfüllt die zweite Normalform (2NF), wenn für alle $B \rightarrow a \in \mathcal{F}$ mit *Nichtschlüssel-Attribut* a gilt, dass B ein Schlüsselkandidat ist oder B ein Nichtschlüssel-Attribut enthält.

Das heißt, es gibt kein Nichtschlüssel-Attribut, das voll funktional abhängig von einem echten *Teil*schlüssel ist.

Beispiel

Das Relationenschema

`name artikel adresse preis`

mit `name` $\rightarrow$ `adresse` ist nicht in 2NF, da `name` kein Schlüsselkandidat ist (sondern `name`, `artikel`), wohl aber ein Schlüsselattribut und da `adresse` ein Nichtschlüssel-Attribut ist.

Die Relationenschemata

`name adresse` `name` $\rightarrow$ `adresse`
`name artikel, preis`

sind dagegen in 2NF.

5.2.3 Dritte Normalform (3NF)

Eine rFD-Menge $\mathcal{F}$ erfüllt die dritte Normalform (ist in 3NF), wenn für alle $B \rightarrow a \in \mathcal{F}$ mit *Nichtschlüssel-Attribut* a gilt, dass B ein Schlüsselkandidat ist.

Diese Normalform ist strenger als 2NF, da Nichtschlüssel-Attribute nur noch von Schlüssel-Kandidaten abhängen dürfen, nicht mehr aber von Nichtschlüssel-Kandidaten: 3NF $\Rightarrow$ 2NF ($\Rightarrow$ 1NF)

Beispiel

Das Relationenschema

artikel_nr	dm_preis	euro_preis
1	1,96	1,00
2	1,96	1,00

dm_preis → euro_preis
euro_preis → dm_preis

ist nicht in 3NF (wohl aber in 2NF).

Die Relationenschemata

artikel_nr euro_preis	
euro_preis dm_preis	euro_preis → dm_preis dm_preis → euro_preis

oder

artikel_nr dm_preis	
dm_preis euro_preis	dm_preis → euro_preis euro_preis → dm_preis

sind dagegen alle in 3NF.

5.2.4 Boyce-Codd-Normalform (BCNF)

Eine rFD-Menge $\mathcal{F}$ erfüllt die Boyce-Codd-Normalform (ist in BCNF), wenn für alle $B \rightarrow a \in \mathcal{F}$ gilt, dass B ein Schlüsselkandidat ist.

Diese Normalform berücksichtigt – im Gegensatz zu 2NF und 3NF – auch Abhängigkeiten zwischen Schlüsselattributen. Es gilt also:

BCNF $\Rightarrow$ 3NF $\Rightarrow$ 2NF ($\Rightarrow$ 1NF)

Beispiel

Das Relationenschema

stadt	**strasse**	**postleitzahl**
Königsbrunn	Blumenallee	86343
? (Redundanz)	Lilienstraße	86343

mit der rFD-Menge

stadt,strasse → postleitzahl, postleitzahl → stadt

ist in 3NF (da es gar keine Nichtschlüssel-Attribute gibt), aber nicht in BCNF. Allerdings sind diese funktionalen Abhängigkeiten realtitätsfern (vgl. Abschnitt 2.2, Fußnote zu Beispiel 2).

Grund: Es gibt zwei Schlüsselkandidaten:

stadt, strasse und strasse, postleitzahl

Das Attribut postleitzahl alleine ist allerdings kein Schlüsselkandidat.

Das Relationenschemata

stadt, strasse
strasse, postleitzahl

sind in BCNF. Nun können allerdings die funktionalen Abhängigkeiten des Ausgangsschemas *nicht* mehr gewährleistet werden (außer, wenn beide Tabellen immer *gleichzeitig* und dann konsistent modifiziert werden).

5.3 Zerlegung eines Relationenschemas

Ein gegebenes Relationenschema R kann bzgl. einer Menge $\mathcal{F}$ immer so in Relationenschemata $R_1, \ldots, R_n$ zerlegt werden (= *Dekomposition*) , dass Folgendes gilt:

1. $R_1, \ldots, R_n$ sind – je nach Wunsch – in 2NF, 3NF oder BCNF
2. Für jede Relation r, die $\mathcal{F}$ erfüllt, gilt $r = \pi_{R_1}(r) \bowtie \ldots \bowtie \pi_{R_n}(r)$

Die zweite Bedingung heißt *Lossless-Join-* oder *Nonadditive-Join-Eigenschaft*. Sie besagt, das man mit Hilfe von (Natural-)Joins die ursprüngliche Relation aus den zerlegten Relationen berechnen kann.

Man beachte, dass diese Bedingung nicht für jede beliebige Zerlegung gilt:

R

mitarbeiter	alter	tel
Anton	38	427
Berta	38	305

R_1

mitarbeiter	alter
Anton	38
Berta	38

R_2

alter	tel
38	427
38	305

$R_1 \bowtie R_2$

mitarbeiter	alter	tel
Anton	38	427
Anton	38	305
Berta	38	427
Berta	38	305

Die angegebene Zerlegung ist *nicht* möglich, da eine funktionale Abhängigkeit `alter` $\rightarrow$ `tel` nicht existiert.

5.3.1 Algorithmen

Algorithmus 3NF-Dekomposition

Gegeben seien ein Relationenschema R zusammen mit einer Menge $\mathcal{F}$ von funktionalen Abhängigkeiten. Alle Attribute sollten in irgendeiner (wenn auch nicht notwendigerweise funktionalen) Beziehungen zu anderen Attributen stehen. Sollte es Attribute ohne irgendeine Beziehung zueinander geben (wie z. B. `geburtstag`, `kbyte`, `anzahl_reserve_raeder`), so sollten R und $\mathcal{F}$ vorher in mehrere Teilschemata aufgespalten werden.

Algorithmus

1. Erzeuge für jede Abhängigkeit $B \rightarrow a \in \mathcal{F}$ das Relationenschema Ba.
2. Wenn keines der erzeugten Relationenschemata einen vollständigen Schlüsselkandidaten enthält, wähle einen Schlüsselkandidaten aus und füge ihn als eigenes Schema hinzu.
3. Entferne Schemata, die Teil eines anderen Schemas sind.
4. Fasse Relationenschemata mit gleichen Schlüsselkandidaten zu einem Schema zusammen, sofern zwischen den Nichtschlüssel-Attributen keine funktionalen Abhängigkeiten bestehen.

   $\underline{\mathsf{S}}\ \mathsf{A}, \underline{\mathsf{S}}\ \mathsf{B} \mathrel{\hat{=}} \underline{\mathsf{S}}\ \mathsf{A}\ \mathsf{B}$

Algorithmus BCNF-Dekomposition

Wenn $B \rightarrow a$ die BCNF-Eigenschaft von $R = a_1, \ldots, a_n$ verletzt, zerlege R in

$R_1 = R - \{a\}$ und

$R_2 = Ba$

und zerlege die resultierenden Schemata weiter:

BCNF-Dekomposition $(R_1, \mathcal{F})$

BCNF-Dekomposition $(R_2, \mathcal{F})$

Beispiel

R

V Vorlesung
P Professor
H Stunde (+ Tag)
R Raum
S Student
N Note

$\mathcal{F}$

V → P Ein Professor pro Vorlesung.
HR → V höchstens eine Vorlesung pro Stunde und Raum.
HP → R Ein Professor kann zur selben Zeit nicht an zwei Orten sein.
HS → R Auch Studenten sind unteilbar.
VS → N Ein Student hat eine Note pro Fach.

3NF-Dekomposition

$\underline{\mathsf{VP}}\ \underline{\mathsf{HR}}\mathsf{V}\ \underline{\mathsf{HP}}\mathsf{R}\ \underline{\mathsf{HS}}\mathsf{R}\ \underline{\mathsf{VS}}\mathsf{N}$

Schlüsselkandidaten

V → VP
HR → HRVP
HP → HPRV
VS → VSNP
HS → HSRVPN

HS ist in HSR bereits enthalten, damit ist das obige Schema das gesuchte 3NF-Schema. Es ist nicht ideal, da die Lossless-join-Dekomposition von R auch mit einigen echten Teilmengen dieses Schemas erreicht wird (⇒ Redundanz). Man kann aber auf keine Tabelle verzichten, wenn man alle Abhängigkeiten berücksichtigen will.

BCNF-Dekomposition

VPHRSN
V → P: V ist kein Schlüsselkandidat
VHRSN VP
HR → V: HR ist kein Schlüsselkandidat
HRSN HRV VP
HP → R: entfällt
VS → N: entfällt
HS → R: HS ist ein Schlüsselkandidat
HRSN HRV VP

Dies ist ein BCNF-Schema, aber ein schlechtes. Zum Beispiel ist die Note nicht bei der zugehörigen Vorlesung, zwei Abhängigkeiten werden gar nicht mehr beachtet.

Beispiel 2

R: stadt strasse plz
$\mathcal{F}$: stadt strasse → plz, plz → stadt

3NF-Dekomposition

stadt strasse plz
plz stadt

Das zweite Schema entfällt, da es Teil des ersten ist.

Das erste Schema ist kein Schlüsselkandidat, da es folgende Schlüsselkandidaten gibt:

stadt strasse → plz stadt strasse

BCNF-Dekomposition

`plz` $\rightarrow$ `stadt` d. h. , `plz` ist kein Schlüsselkandidat

```
stadt strasse
plz   stadt
```

Fazit

Die Zerlegung (Dekomposition) eines Relationenschemas in 3NF-Schemata (und damit auch in 2NF-Schemata) bewahrt alle funktionalen Abhängigkeiten, ist also *abhängigkeitserhaltend*. BCNF ist dagegen nicht abhängigkeitserhaltend (Integritätsbedingungen fallen weg) vermeidet aber – in seltenen Spezialfällen – mehr Redundanz und damit Updateanomalien. In manchen Fällen verzichtet man aus Effizienzgründen auf eine vollständige 3NF- oder BCNF-Normalisierung.

5.4 Weitere Normalformen

Funktionale Abhängigkeiten sind nicht die einzigen Abhängigkeiten zwischen Attributen, die zu redundanter Speicherung führen. Es gibt weitere Abhängigkeiten, wie

- Mehrwertige Abhängigkeiten ($\Rightarrow$ 4NF)
- Join-Abhängigkeiten ($\Rightarrow$ 5NF)
- Abhängigkeiten zwischen Relationen (bislang keine Normalformen)
- etc.

Jede weitere Normalform sollte die bekannten Normalformen verallgemeinern. Es gilt z. B.

$$5\text{NF} \Rightarrow 4\text{NF} \Rightarrow \text{BCNF} \Rightarrow 2\text{NF} \Rightarrow 1\text{NF}$$

Das heißt, jede weitere Normalform zerschlägt ein Relationenschema in immer mehr, kleinere Teilrelationen.

Vorteil: Redundanzvermeidung, Anomalienvermeidung

Nachteil: Effizienzprobleme durch mehr Joins

In der Praxis ist 3NF (und manchmal sogar 2NF) fast immer ausreichend. Schemata, die zwar 3NF, aber nicht BCNF sind, kommen in der Praxis so gut wie nicht vor (z. B. ist das obige Stadt/Straße/Postleitzahl-Beispiel *nicht* praxisgerecht, da beide Abhängigkeiten in der Praxis nicht gelten).

5.5 Normalformen und ER-Modellierung

Entityschemata mit *einem* künstlichen Schlüsselattribut (wie z. B. `personalnr`) sind automatisch in 2NF, wenn es keine weiteren Schlüsselkandidaten gibt: Jedes Attribut ist voll funktional abhängig vom Primärschlüssel.

In der Praxis gibt es manchmal gar keine weiteren Schlüsselkandidaten. So wird eine Person unter Umständen noch nicht einmal durch

```
name, vorname, geburtsdatum, adresse
```

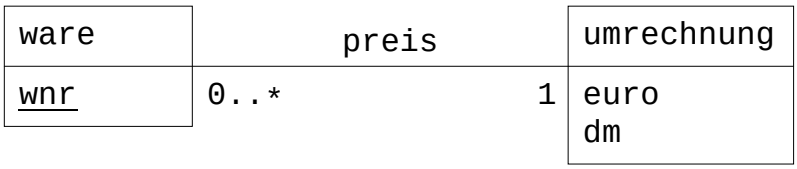
eindeutig identifiziert (z. B. zwei Sepp Meier, geb. 1.1.1960, wohnhaft in demselben Hochhaus). Allerdings ist es fast immer sinnvoll, neben einem künstlichen Schlüssel auch eine Schlüsselkandidaten zu haben, der von menschlichen Datenbankbenutzern gelesen und verstanden wird.

Sollte es einen weiteren Schlüsselkandidaten geben, so sollte man diesen – im Sinne von BCNF – zusammen mit dem (künstlichen) Primärschlüssel in einer eigenen Tabelle speichern.

Abhängigkeiten zwischen Nichtschlüssel-Attributen sollten – gemäß 3NF – vermieden werden. Das heißt, folgendes Schema ist schlecht:

ware
wnr dm_preis euro_preis

Besser ist folgendes Schema:



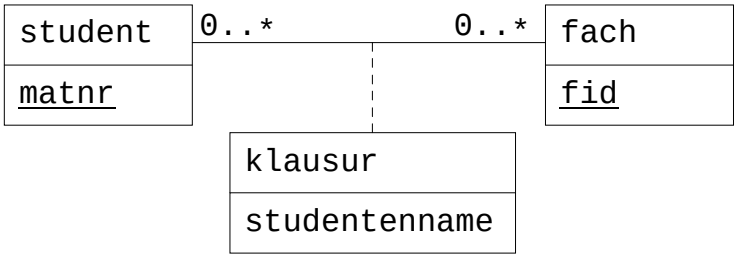
oder gar:

ware
wnr euro_preis dm_preis()

Das heißt, `dm_preis` wird als berechnetes Attribut realisiert, z. B. mit einer View und einem Trigger. In diesem Fall ist nämlich sogar eine mathematische *Funktion* zur Berechnung der *funktionalen* Abhängigkeit bekannt.

1 DM = 1 Euro : 1,95583 gerundet auf sechs Stellen

m.n-Beziehungen haben mindestens zwei Schlüsselattribute. Funktionale Abhängigkeiten, die sich nicht auf alle Foreign-Key-Attribute beziehen, sind i. Allg. auf Designfehler zurückzuführen.



Designfehler: Der Studentenname gehört zum Studenten und nicht zur Klausur:

```
klausur: matnr, fid, studentenname
matnr → studentenname ⇒ nicht 2NF
```

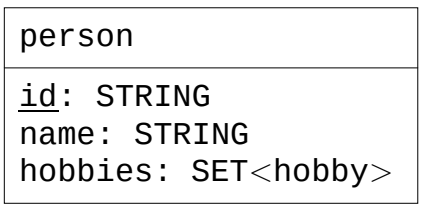
Mehrwertige Abhängigkeiten entstehen i. Allg. durch mengenwertige Attribute:

```
name     hobbies          geburtstagsjahr
Wolfgang {Tischtennis,    1961
          Akkordeon,
          Fahrrad
         }
```

⇒ 1NF-Form

```
name     hobbies      geburtstagsjahr
Wolfgang Tischtennis  1961
Wolfgang Akkordeon    1961
Wolfgang Fahrrad      1961
```

Die redundante, nicht gewollte Speicherung von `Wolfgang` und `1961` ist auf eine schlechte 1NF-Repräsentation zurückzuführen. Wie bereits bekannt ist, realisieren wir



durch

d. h. durch mehrere Tabellen. Deshalb treten mehrwertige Abhängigkeiten (i. Allg.) nicht auf und müssen nicht weiter berücksichtigt werden ($\Rightarrow$ keine 4NF-Probleme).

5NF-Probleme sind bei umfangreichen DB-Schemata nur schwer zu erkennen und werden daher i. Allg. nicht berücksichtigt.

Kapitel 6

Fehlende Kapitel

SQL: Window-Funktionen

Indexe: Primär- und Sekundärindexe, `CREATE INDEX` etc.

Trigger: SQL-Standard, PostgreSQL, SQLite

XML: SQL-Standard, PostgreSQL, XQuery, eXist

Literatur

L. Andersen [2006]: *JDBC™ 4.0 Specification*, SUM Microsystems, Inc., November 2006

R. Bayer [1982]: *Datenstrukturen – Kurseinheit 6: Datenstrukturen für Peripheriespeicher*, Band 6, Fernuniversität – Gesamthochschule – in Hagen, Hagen

R. Bayer [1996]: Datenbanksysteme, Vorlesungsskript, Technische Universität München

R. Bayer, E. M. McCreight [1970]: Organization and maintenance of large ordered indexes, In: *SIGFIDET Workshop*, Seiten 107–141, ACM

Chen [1976]: The entity-relationship model: Towards a unified view of data, *ACM Transactions on Database Systems*, Jahrgang 1, Nr. 1, Seiten 9–36, März 1976

E. Codd [1970]: A relational model of data for large shared data banks, *Communications of the ACM*, Jahrgang 13, Nr. 6, Seiten 377–387

C. Date, H. Darwen [1993]: *A Guide to the SQL Standard*, Addison-Wesley Publishing Company, 3. Ausgabe

J. Ellis, L. Ho, M. Fisher [2001]: *JDBC™ 3.0 Specification*, SUM Microsystems, Inc., Oktober 2001

R. Elmasri, S. B. Navathe [1994]: *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, 2. Ausgabe

R. Elmasri, S. B. Navathe [2002]: *Grundlagen von Datenbanksystemen*, Pearson Studium

D. Flanagan [1996]: *Java in a Nutshell*, O'Reilly International Thomson

D. Flanagan [1998]: *Java in a Nutshell – Deutsche Ausgabe*, O'Reilly International Thomson

P. Forbrig [2001]: *Objektorientierte Softwareentwicklung mit UML*, Fachbuchverlag Leipzig im Carl Hanser Verlag

H. Garcia-Molina, J. D. Ullman, J. Widom [2002]: *Database Systems – The Complete Book*, Prentice-Hall International Inc.

P. Gulutzan, T. Pelzer [1999]: *SQL-99 Complete*, R&D Books

A. Guttman [1984]: R-trees: A dynamic index structure for spatial searching, In: *SIGMOD Conference* (Hg. B. Yormark), Seiten 47–57, ACM Press

F. Halasz, M. Schwartz [1994]: The Dexter hypertext reference model, *Communications of the ACM*, Jahrgang 37, Nr. 2, Seiten 30–39

A. Heuer [1997]: *Objektorientierte Datenbanken*, Band 2, Addison-Wesley Publishing Company

C. Horstmann, G. Cornell [1997]: *Core Java*, Band I – Fundamentals, SUN Microsystems Press, Prentice Hall

C. Horstmann, G. Cornell [1998]: *Core Java*, Band II – Advanced Features, SUN Microsystems Press, Prentice Hall

ISO:2003-1 [2006]: *ISO/IEC 9075-1:2003: Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

ISO:2003-10 [2007]: *ISO/IEC 9075-10:2003: Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

ISO:2003-11 [2003]: *ISO/IEC 9075-11:2003: Information technology—Database languages—SQL—Part 11: Information and Definition Schemas (SQL/Schemata)*, International Organization for Standardization (ISO), Genf, 1. Ausgabe

ISO:2003-13 [2003]: *ISO/IEC 9075-13:2003: Information technology—Database languages—SQL—Part 13: SQL Routines and Types Using the Java TM Programming Language (SQL/JRT)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

ISO:2003-2 [2007]: *ISO/IEC 9075-2:2003: Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

ISO:2003-3 [2007]: *ISO/IEC 9075-3:2003: Information technology—Database languages—SQL—Part 3: Call-Level Interface (SQL/CLI)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

ISO:2003-4 [2003]: *ISO/IEC 9075-4:2003: Information technology—Database languages—SQL—Part 4: Persistent Stored Modules (SQL/PSM)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

ISO:2003-9 [2003]: *ISO/IEC 9075-9:2003: Information technology—Database languages—SQL—Part 9: Management of External Data (SQL/MED)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

ISO:2006-14 [2006]: *ISO/IEC 9075-14:2006: Information technology—Database languages—SQL—Part 14: XML-Related Specifications (SQL/XML)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

M. Jeckle, C. Rupp, J. Hahn, B. Zengler, S. Queins [2003]: *UML 2 glasklar*, Fachbuchverlag Leipzig im Carl Hanser Verlag

J. Kempe, W. Kowarschick, W. Kießling, R. Hitzelberger, F. Dutkowski [1995]: The OCAD benchmark for object-oriented database systems, FORWISS-Report FR-1995-004, Bayerisches Forschungszentrum für Wissensbasierte Systeme, Erlangen, München, Passau

W. Kießling, G. Köstler [1998]: *Multimedia-Kurs Datenbanksysteme*, Springer-Verlag

C. Kowarschick [1999]: Universalvehikel – Anwendungsbezogen: SAP Standard Application Benchmark, *iX Magazin für professionelle Informationstechnik*, Jahrgang 12,

Seiten 136ff, Dezember 1999

W. Kowarschick [1994]: Praxis wissensbasierter Systeme, Vorlesungsskript, Universität Augsburg

G. Köstler, W. Kowarschick, W. Kießling [1997]: Client-server optimization for multimedia document exchange, In: *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA'97)*, Seiten 135–144, Melbourne, Australia, April 1997

G. Lausen, G. Vossen [1996]: *Objekt-orientierte Datenbanken: Modelle und Sprachen*, R. Oldenbourg Verlag

J. Michael [1999]: Doppelgänger gesucht – Ein Programm für kontextsensitive phonetische Textumwandlung , *c't Magazin für Computertechnik*, Jahrgang 1999, Nr. 25, Seiten 252–261, Dezember 1999

H. Neumann [1998]: *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*, Carl Hanser Verlag

K. Neumann [1996]: *Datenbanktechnik für Anwender*, Carl Hanser Verlag

Oracle [2002]: www.oracle.com

O. Preußler [1962]: *Der Räuber Hotzenplotz*, Thienemann Verlag, 55. Ausgabe

O. Preußler [2006]: *Neues vom Räuber Hotzenplotz – Noch eine Kasperlgeschichte*, Thienemann Verlag, 39. Ausgabe

R. Ramakrishnan, J. Gehrke [2002]: *Database Management Systems*, McGraw-Hill Higher Education, 3. Ausgabe

F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, R. Bayer [2000]: Integrating the UB-Tree into a database system kernel, In: *VLDB* (Hg. A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, K.-Y. Whang), Seiten 263–272, Morgan Kaufmann

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen [1991]: *Object-Oriented Modeling and Design*, Prentice-Hall

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen [1993]: *Objektorientiertes Modellieren und Entwerfen*, Hanser Verlag, Prentice-Hall

SQL:2008-1 [2008]: *ISO/IEC 9075-1:2008: Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL:2008-10 [2008]: *ISO/IEC 9075-10:2008: Information technology—Database languages—SQL—Part 10: Object Language Bindings (SQL/OLB)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL:2008-11 [2008]: *ISO/IEC 9075-11:2008: Information technology—Database languages—SQL—Part 11: Information and Definition Schemas (SQL/Schemata)*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

SQL:2008-13 [2008]: *ISO/IEC 9075-13:2008: Information technology—Database languages—SQL—Part 13: SQL Routines and Types Using the Java TM Programming Language (SQL/JRT)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL:2008-14 [2008]: *ISO/IEC 9075-14:2008: Information technology—Database languages—SQL—Part 14: XML-Related Specifications (SQL/XML)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL:2008-2 [2008]: *ISO/IEC 9075-2:2008: Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL:2008-3 [2008]: *ISO/IEC 9075-3:2008: Information technology—Database languages—SQL—Part 3: Call-Level Interface (SQL/CLI)*, International Organization for Standardization (ISO), Genf, 4. Ausgabe

SQL:2008-4 [2008]: *ISO/IEC 9075-4:2008: Information technology—Database languages—SQL—Part 4: Persistent Stored Modules (SQL/PSM)*, International Organization for Standardization (ISO), Genf, 4. Ausgabe

SQL:2008-9 [2008]: *ISO/IEC 9075-9:2008: Information technology—Database languages—SQL—Part 9: Management of External Data (SQL/MED)*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL/MM:2003-2 [2003]: *ISO/IEC 13249-3:2003: Information technology—Database languages—SQL Multimedia an Application Packages—Part 2: Full-Text*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

SQL/MM:2003-5 [2003]: *ISO/IEC 13249-5:2003: Information technology—Database languages—SQL Multimedia an Application Packages—Part 5: Still image*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

SQL/MM:2006-3 [2006]: *ISO/IEC 13249-3:2006: Information technology—Database languages—SQL Multimedia an Application Packages—Part 3: Spatial*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL/MM:2006-6 [2006]: *ISO/IEC 13249-6:2006: Information technology—Database languages—SQL Multimedia an Application Packages—Part 6: Date mining*, International Organization for Standardization (ISO), Genf, 2. Ausgabe

SQL/MM:2007-1 [2007]: *ISO/IEC 13249-1:2007: Information technology—Database languages—SQL Multimedia an Application Packages—Part 1: Framework*, International Organization for Standardization (ISO), Genf, 3. Ausgabe

SQL/MM:2007-7 [2007]: *ISO/IEC 13249-7: Information technology—Database languages—SQL Multimedia an Application Packages—Part 7: History (*in Entwicklung*)*, International Organization for Standardization (ISO), Genf, 1. Ausgabe

A. Teynor, W. Müller, W. L. J. Kowarschick [2005]: Compressed domain image retrieval using JPEG2000 and gaussian mixture models, In: *8th International Conference on Visual Information Systems* (Hg. S. Bres, R. Laurini), Band 3736 von *Lecture Notes*

in Computer Science, Seiten 132–142, Springer

TransAction [2002]: `www.transaction.de`

TransBase CD [2012]: `http://www.transaction.de/produkte/transbase-cd/`

C. Türker [2003]: *SQL:1999 & SQL:2003*, Band 1, dpunkt.verlag

J. Ullman [1988]: *Principles of Database and Knowledge-Base Systems: Classical Database Systems*, Band 1, Computer Science Press

J. Ullman [1989]: *Principles of Database and Knowledge-Base Systems: The New Technologies*, Band 2, Computer Science Press

K. Wilhelm, J. Pfister, C. Kowarschick, S. Gradek [2001]: Monitoring und Benchmarking für das R/3-Performance-Engineering, In: *Kursbuch Kapazitätsmanagement – Kompendium für Planung, Analyse und Tuning von IT-Systemen*, Norderstedt, Books on Demand