Datenbanksysteme 2011

Oliver Vornberger, Nicolas Neubauer
Universität Osnabrück, Institut für Informatik

Die Veranstaltung befasst sich mit der Verwaltung großer Datenbestände. Themen sind: Modellierungskonzepte, B*-Baum, Grid-File, der hierarchische Ansatz, der Netzwerkansatz, der relationale Ansatz, der objektorientierte Ansatz, SQL, XML, Datenbankapplikationen, ODBC, JDBC, PHP, Ruby on Rails, Transaktionsverwaltung, Mehrbenutzersynchronisation, Recovery, Sicherheit.

Literatur:

- Schlageter G., W. Stucky: *Datenbanksysteme: Konzepte und Modelle*, Teubner, 1983
- Ullman, J. D.: *Principles of Data and Knowledge-Base Systems*, Computer Science Press, 1988.
- Date, C.J: *An Introduction to Database Systems*, Addison-Wesley, 1995.
- Hamilton G., R. Cattell, M. Fisher: *JDBC Datenbankzugriff mit Java*, Addison-Wesley, 1999
- Heuer, A. & G. Saake: *Datenbanken - Konzepte und Sprachen*, International Thompson Publishing, 2000
- Elmasri R. & S. Navathe: *Fundamentals of Database Systems* Addison Wesley, 2000
- Connolly, Th. & C. Begg: *Database Systems*, Person Education, 4th Edition, 2004
- Harold, E.: *The XML Bible*, Wiley & Sons, 2004
- Kay, M.: *XSLT Programmer's Reference*, Wrox Press, 2004
- Walmsley, P.: *XQuery - Search Across a Variety of XML Data*, O'Reilly, 2007
- Tidwell, D.: *XSLT - Mastering XML Transformations, O'Reilly*, 2. Auflage, 2008.
- Vossen, G.: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*, Oldenbourg, 2008
- Kemper, A. & A. Eickler: *Datenbanksysteme - Eine Einführung*, Oldenbourg, 2009
- Ruby, S. & Thomas, D. & Hansson, D.: Agile Web Development with Rails - Fourth Edition. The Pragmatic Bookshelf, 2011

Danksagung:

Wir danken ...

- Astrid Heinze für das sorgfältige Erfassen zahlreicher Texte, Grafiken und Tabellen
- Patrick Fox, Friedhelm Hofmeyer, Ralf Kunze und Olaf Müller für die Installation und Erprobung diverser Software-Pakete

Osnabrück, im März 2011

-- Oliver Vornberger, Nicolas Neubauer

Inhaltsverzeichnis

1. Einführung

1.1 Definition

Ein **Datenbanksystem** (auch *Datenbankverwaltungssystem*, abgekürzt *DBMS = data base management system*) ist ein computergestütztes System, bestehend aus einer Datenbasis zur Beschreibung eines Ausschnitts der Realwelt sowie Programmen zum geregelten Zugriff auf die Datenbasis.

1.2 Motivation

Die separate Abspeicherung von teilweise miteinander in Beziehung stehenden Daten durch verschiedene Anwendungen würde zu schwerwiegenden Problemen führen:

- **Redundanz:** Dieselben Informationen werden doppelt gespeichert.
- **Inkonsistenz:** Dieselben Informationen werden in unterschiedlichen Versionen gespeichert.
- **Integritätsverletzung:** Die Einhaltung komplexer Integritätsbedingungen fällt schwer.
- **Verknüpfungseinschränkung:** Logisch verwandte Daten sind schwer zu verknüpfen, wenn sie in isolierten Dateien liegen.
- **Mehrbenutzerprobleme:** Gleichzeitiges Editieren derselben Datei führt zu Anomalien (*lost update*).
- **Verlust von Daten:** Außer einem kompletten Backup ist kein Recoverymechanismus vorhanden.
- **Sicherheitsprobleme:** Abgestufte Zugriffsrechte können nicht implementiert werden.
- **Hohe Entwicklungskosten:** Für jedes Anwendungsprogramm müssen die Fragen zur Dateiverwaltung erneut gelöst werden.

Also bietet sich an, mehreren Anwendungen in jeweils angepasster Weise den Zugriff auf eine gemeinsame Datenbasis mit Hilfe eines Datenbanksystems zu ermöglichen (Abbildung 1).

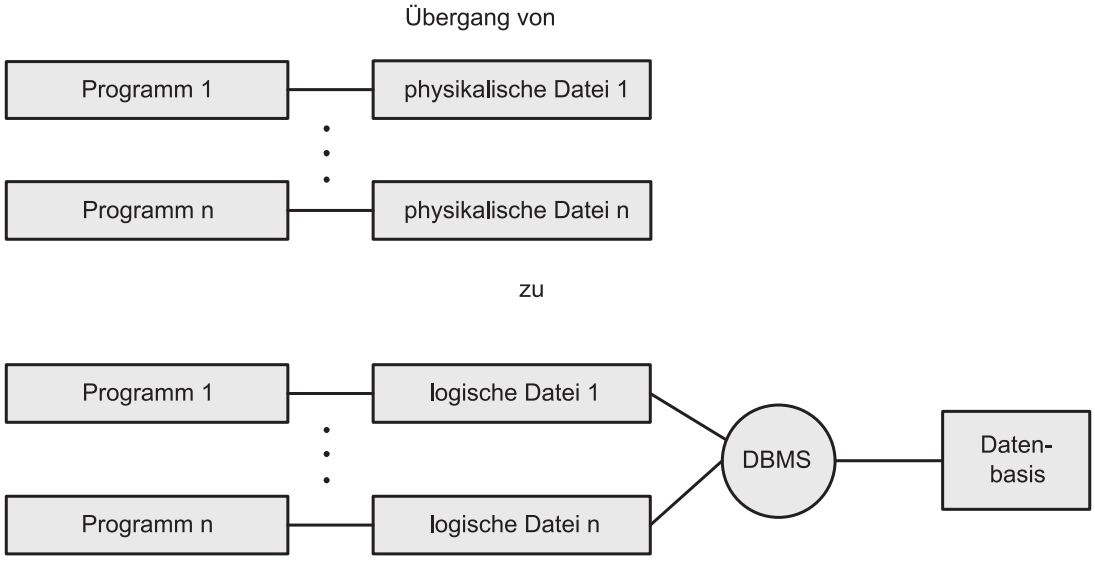


Abb. 1: Isolierte Dateien versus zentrale Datenbasis

1.3 Datenabstraktion

Man unterscheidet drei Abstraktionsebenen im Datenbanksystem (Abbildung 2):

- **Konzeptuelle Ebene:** Hier wird, unabhängig von allen Anwenderprogrammen, die Gesamtheit aller Daten, ihre Strukturierung und ihre Beziehungen untereinander beschrieben. Die Formulierung erfolgt vom *enterprise administrator* mittels einer *DDL (data definition language)*. Das Ergebnis ist das konzeptuelle Schema, auch genannt Datenbankschema.
- **Externe Ebene:** Hier wird für jede Benutzergruppe eine spezielle anwendungsbezogene Sicht der Daten (*view*) spezifiziert. Die Beschreibung erfolgt durch den *application administrator* mittels einer DDL, der Umgang vom Benutzer erfolgt durch eine *DML (data manipulation language)*. Ergebnis ist das externe Schema.
- **Interne Ebene:** Hier wird festgelegt, in welcher Form die logisch beschriebenen Daten im Speicher abgelegt werden sollen. Geregelt werden record-Aufbau, Darstellung der Datenbestandteile, Dateiorganisation, Zugriffspfade. Für einen effizienten Entwurf werden statistische Informationen über die Häufigkeit der Zugriffe benötigt. Die Formulierung erfolgt durch den *database administrator*. Ergebnis ist das interne Schema.

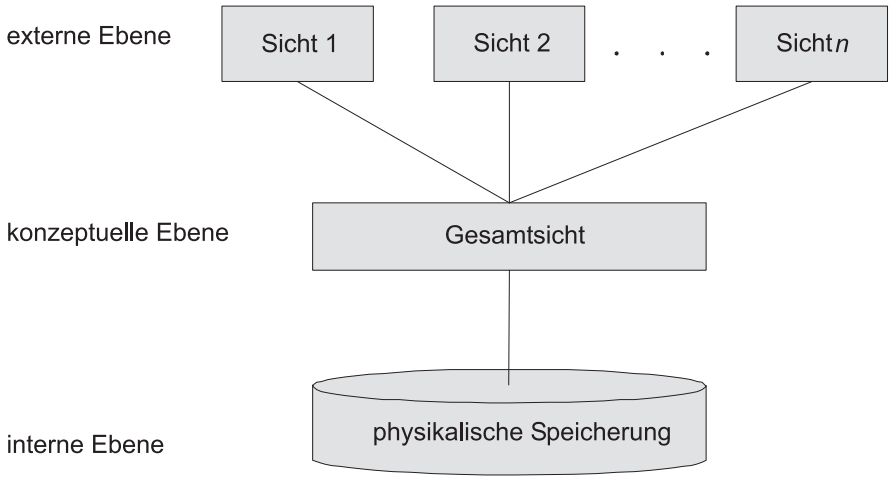


Abb. 2: Drei Abstraktionsebenen eines Datenbanksystems

Das *Datenbankschema* legt also die Struktur der abspeicherbaren Daten fest und sagt noch nichts über die individuellen Daten aus. Unter der *Datenbankausprägung* versteht man den momentan gültigen Zustand der Datenbasis, die natürlich den im Schema festgelegten Strukturbeschreibungen gehorchen muss.

1.4 Transformationsregeln

Die Verbindungen zwischen den drei Ebenen werden durch die *Transformationsregeln* definiert. Sie legen fest, wie die Objekte der verschiedenen Ebenen aufeinander abgebildet werden. Z. B. legt der *Anwendungsadministrator* fest, wie Daten der externen Ebene aus Daten der konzeptuellen Ebene zusammengesetzt werden. Der *Datenbank-Administrator (DBA)* legt fest, wie Daten der konzeptuellen Ebene aus den abgespeicherten Daten der internen Ebene zu rekonstruieren sind.

- **Beispiel Bundesbahn:** Die Gesamtheit der Daten (d. h. Streckennetz mit Zugverbindungen) ist beschrieben im konzeptuellen Schema (Kursbuch). Ein externes Schema ist z. B. beschrieben im Heft *Städteverbindungen Osnabrück*.
- **Beispiel Personaldatei:** Die konzeptuelle Ebene bestehe aus Angestellten mit ihren Namen, Wohnorten und Geburtsdaten. Das externe Schema `Geburtstagsliste` besteht aus den Komponenten `Name`, `Datum`, `Alter`, wobei das `Datum` aus Tag und Monat des Geburtsdatums besteht, und `Alter` sich aus der Differenz vom laufenden Jahr und Geburtsjahr berechnet.

Im internen Schema wird festgelegt, dass es eine Datei PERS gibt mit je einem record für jeden Angestellten, in der für seinen Wohnort nicht der volle Name, sondern eine Kennziffer gespeichert ist. Eine weitere Datei ORT enthält Paare von Kennziffern und Ortsnamen. Diese Speicherorganisation spart Platz, wenn es nur wenige verschiedene Ortsnamen gibt. Sie verlangsamt allerdings den Zugriff auf den Wohnort.

1.5 Datenunabhängigkeit

Die drei Ebenen eines DBMS gewähren einen bestimmten Grad von *Datenunabhängigkeit*:

- **Physische Datenunabhängigkeit:**
  Die Modifikation der physischen Speicherstruktur (z. B. das Anlegen eines Index) verlangt nicht die Änderung der Anwenderprogramme.
- **Logische Datenunabhängigkeit:**
  Die Modifikation der Gesamtsicht (z. B. das Umbenennen von Feldern) verlangt nicht die Änderung der Benutzersichten.

1.6 Modellierungskonzepte

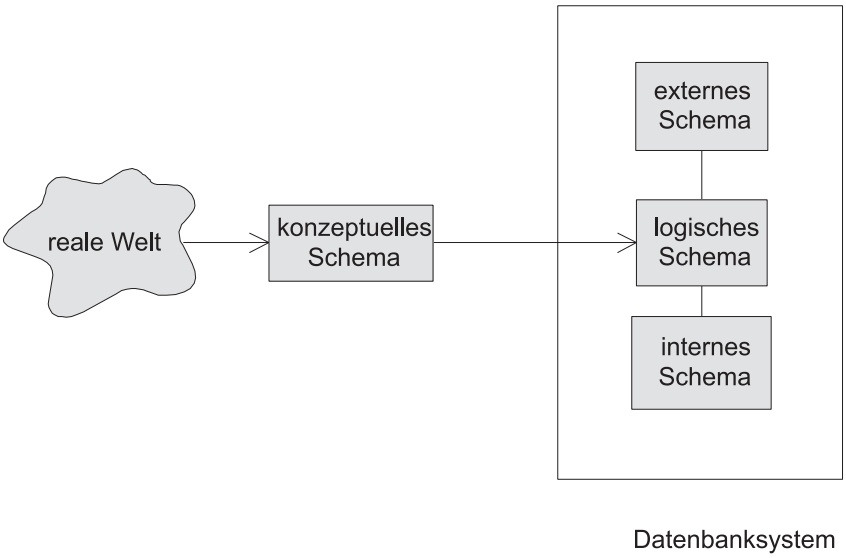


Abb. 3: Zweistufige Modellierung

Das konzeptuelle Schema soll sowohl die reale Welt unabhängig von DV-Gesichtspunkten beschreiben als auch die Grundlage für das interne Schema bilden, welches natürlich stark maschinenabhängig ist. Um diesen Konflikt zu lösen, stellt man dem konzeptuellen Schema ein sogenanntes "logisches" Schema zur Seite, welches die Gesamtheit der Daten zwar hardware-unabhängig, aber doch unter Berücksichtigung von Implementationsgesichtspunkten beschreibt. Das logische Schema heißt darum auch implementiertes konzeptuelles Schema. Es übernimmt die Rolle des konzeptuellen Schemas, das nun nicht mehr Teil des eigentlichen Datenbanksystems ist, sondern etwas daneben steht und z. B. auch aufgestellt werden kann, wenn überhaupt kein Datenbanksystem zum Einsatz kommt (siehe Abbildung 3).

Zur Modellierung der konzeptuellen Ebene verwendet man das **Entity-Relationship-Modell**, welches einen Ausschnitt der Realwelt unter Verwendung von *Entities* und *Relationships* beschreibt :

- **Entity:**
  Gegenstand des Denkens und der Anschauung (z. B. eine konkrete Person, ein bestimmter Ort)
- **Relationship:**
  Beziehung zwischen den entities (z. B. wohnen in)

Entities werden charakterisiert durch eine Menge von Attributen, die gewisse Attributwerte annehmen können. Entities, die durch dieselbe Menge von Attributen charakterisiert sind, können zu einer Klasse, einem Entity-Typ, zusammengefasst werden. Entsprechend entstehen Relationship-Typen.

Beispiel:

Entity-Typ `Studenten` habe die Attribute `Mat-Nr.`, `Name`, `Hauptfach`.
Entity-Typ `Orte` habe die Attribute `PLZ`, `Name`.
Relationship-Typ `wohnen in` setzt `Studenten` und `Orte` in Beziehung zueinander.

Die graphische Darstellung erfolgt durch Entity-Relationship-Diagramme (E-R-Diagramm). Entity-Typen werden durch Rechtecke, Beziehungen durch Rauten und Attribute durch Ovale dargestellt. Abbildung 4 zeigt ein Beispiel.

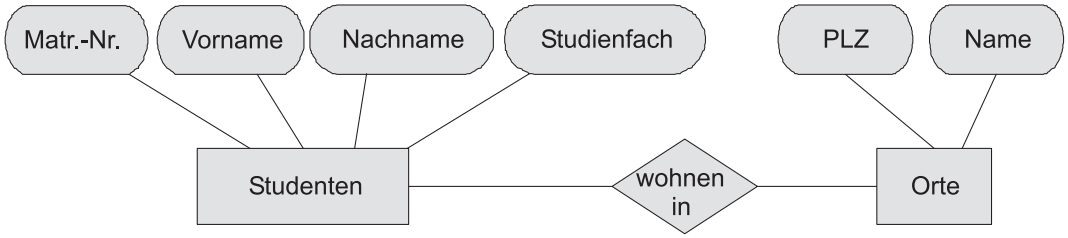


Abb. 4: Beispiel für ein E-R-Diagramm

Zur Formulierung des logischen Schemas stehen je nach zugrundeliegendem Datenbanksystem folgende Möglichkeiten zur Wahl:

- Das hierarchische Modell (z. B. IMS von IBM)
- Das Netzwerkmodell (z. B. UDS von Siemens)
- Das relationale Modell (z. B. Access von Microsoft)
- Das objektorientierte Modell (z. B. O_2 von O_2 Technology)

Das hierarchische Modell (basierend auf dem Traversieren von Bäumen) und das Netzwerkmodell (basierend auf der Navigation in Graphen) haben heute nur noch historische Bedeutung und verlangen vom Anwender ein vertieftes Verständnis der satzorientierten Speicherstruktur. Relationale Datenbanksysteme (basierend auf der Auswertung von Tabellen) sind inzwischen marktbeherrschend und werden teilweise durch Regel- und Deduktionskomponenten erweitert. Objektorientierte Systeme fassen strukturelle und verhaltensmäßige Komponenten in einem Objekttyp zusammen und gelten als die nächste Generation von Datenbanksystemen.

1.7 Architektur

Abbildung 5 zeigt eine vereinfachte Darstellung der Architektur eines Datenbankverwaltungssystems. Im oberen Bereich finden sich vier Benutzerschnittstellen:

- Für häufig zu erledigende und wiederkehrende Aufgaben werden speziell abgestimmte Anwendungsprogramme zur Verfügung gestellt (Beispiel: Flugreservierungssystem).
- Fortgeschrittene Benutzer mit wechselnden Aufgaben formulieren interaktive Anfragen mit einer flexiblen Anfragesprache (wie SQL).
- Anwendungsprogrammierer erstellen komplexe Applikationen durch "Einbettung" von Elementen der Anfragesprache (embedded SQL)
- Der Datenbankadministrator modifiziert das Schema und verwaltet Benutzerkennungen und Zugriffsrechte.

Der DDL-Compiler analysiert die Schemamanipulationen durch den DBA und übersetzt sie in Metadaten.

Der DML-Compiler übersetzt unter Verwendung des externen und konzeptuellen Schemas die Benutzer-Anfrage in eine für den Datenbankmanager verständliche Form. Dieser besorgt die benötigten Teile des internen Schemas und stellt fest, welche physischen Sätze zu lesen sind. Dann fordert er vom Filemanager des Betriebssystems die relevanten Blöcke an und stellt daraus das externe entity zusammen, welches im Anwenderprogramm verarbeitet wird.

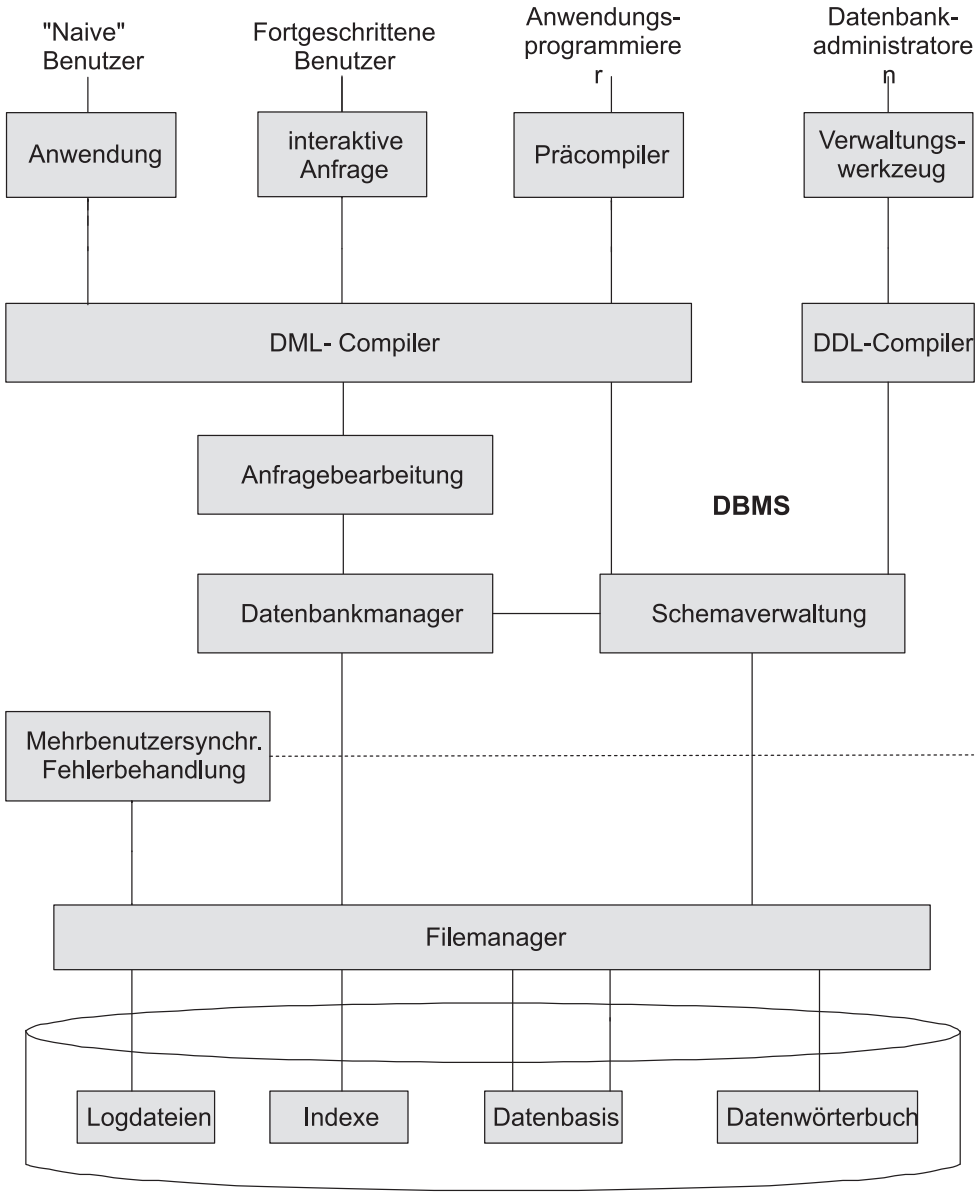


Abb. 5: Architektur eines DBMS

2. Konzeptuelle Modellierung

2.1 Das Entity-Relationship-Modell

Die grundlegenden Modellierungsstrukturen dieses Modells sind die *Entities* (Gegenstände) und die *Relationships* (Beziehungen) zwischen den Entities. Des weiteren gibt es noch *Attribute* und *Rollen*. Die Ausprägungen eines Entity-Typs sind seine Entities, die Ausprägung eines Relationship-Typs sind seine Relationships. Nicht immer ist es erforderlich, diese Unterscheidung aufrecht zu halten.

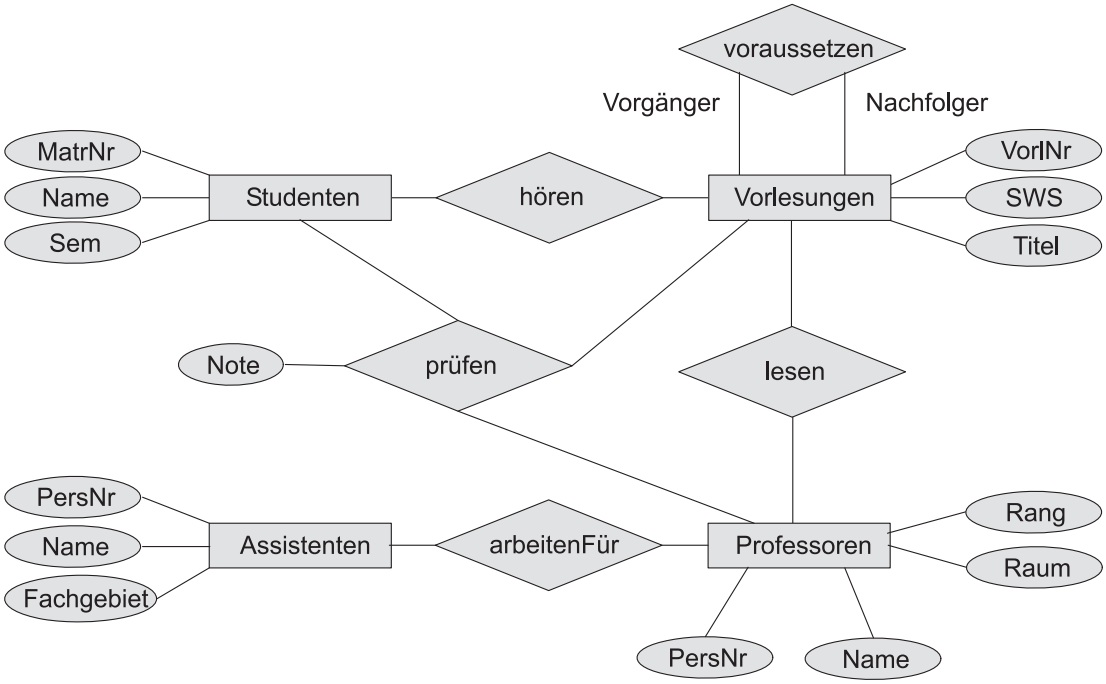


Abb. 6: ER-Diagramm für Universität

Entities sind physisch oder gedanklich existierende Konzepte der zu modellierenden Welt, dargestellt im ER-Diagramm durch Rechtecke. Attribute charakterisieren die Entities und werden durch Ovale beschrieben. Beziehungen zwischen den Entities können binär oder auch mehrwertig sein, sie werden durch Routen symbolisiert.

In Abbildung 6 gibt es einen dreiwertigen Beziehungstyp *prüfen*, der auch über ein Attribut *Note* verfügt. Binäre Beziehungstypen, wie z.B. *voraussetzen*, an denen nur ein Entity-Typ beteiligt ist, werden *rekursive Beziehungstypen* genannt. Durch die Angabe von *Vorgänger* und *Nachfolger* wird die Rolle eines Entity-Typen in einer rekursiven Beziehung gekennzeichnet.

2.2 Schlüssel

Eine minimale Menge von Attributen, welche das zugeordnete Entity eindeutig innerhalb aller Entities seines Typs identifiziert, nennt man *Schlüssel* oder auch *Schlüsselkandidaten*. Gibt es mehrere solcher Schlüsselkandidaten, wird einer als *Primärschlüssel* ausgewählt. Oft gibt es künstlich eingeführte Attribute, wie z.B. Personalnummer *(PersNr)*, die als Primärschlüssel dienen. Schlüsselattribute werden durch Unterstreichung gekennzeichnet. Achtung: Die Schlüsseleigenschaft bezieht sich auf Attribut-Kombinationen, nicht nur auf die momentan vorhandenen Attributwerte!

Beispiel:

Im Entity-Typ `Person` mit den Attributen `Name`, `Vorname`, `PersNr`, `Geburtsdatum`, `Wohnort` ist `PersNr` der Primärschlüssel. Die Kombination `Name`, `Vorname`, `Geburtsdatum` bildet ebenfalls einen (Sekundär-)Schlüssel, sofern garantiert wird, dass es nicht zwei Personen mit demselben Namen und demselben Geburtsdatum gibt.

2.3 Charakterisierung von Beziehungstypen

Ein Beziehungstyp *R* zwischen den Entity-Typen $E_1,\ E_2,\ ...,\ E_n$ kann als Relation im mathematischen Sinn aufgefasst werden. Also gilt:

$$R \subset E_1 \times E_2 \times ... \times E_n$$

In diesem Fall bezeichnet man n als den Grad der Beziehung R. Ein Element $(e_1,\ e_2,\ ...,\ e_n) \in\ R$ nennt man eine Instanz des Beziehungstyps.

Man kann Beziehungstypen hinsichtlich ihrer *Funktionalität* charakterisieren (Abbildung 7). Ein binärer Beziehungstyp R zwischen den Entity-Typen E_1 und E_2 heißt

- **1:1-Beziehung (one-one)**, falls jedem Entity e_1 aus E_1 höchstens ein Entity e_2 aus E_2 zugeordnet ist und umgekehrt jedem Entity e_2 aus E_2 höchstens ein Entity e_1 aus E_1 zugeordnet ist.
  Beispiel: *verheiratet_mit*.
- **1:N-Beziehung (one-many)**, falls jedem Entity e_1 aus E_1 beliebig viele (also keine oder mehrere) Entities aus E_2 zugeordnet sind, aber jedem Entity e_2 aus E_2 höchstens ein Entity e_1 aus E_1 zugeordnet ist.
  Beispiel: *beschäftigen*.
- **N:1-Beziehung (many-one)**, falls analoges zu obigem gilt.
  Beispiel: *beschäftigt_bei*
- **N:M-Beziehung (many-many)**, wenn keinerlei Restriktionen gelten, d.h. jedes Entity aus E_1 kann mit beliebig vielen Entities aus E_2 in Beziehung stehen und umgekehrt kann jedes Entity e_2 aus E_2 mit beliebig vielen Entities aus E_1 in Beziehung stehen.
  Beispiel: *befreundet_mit*.

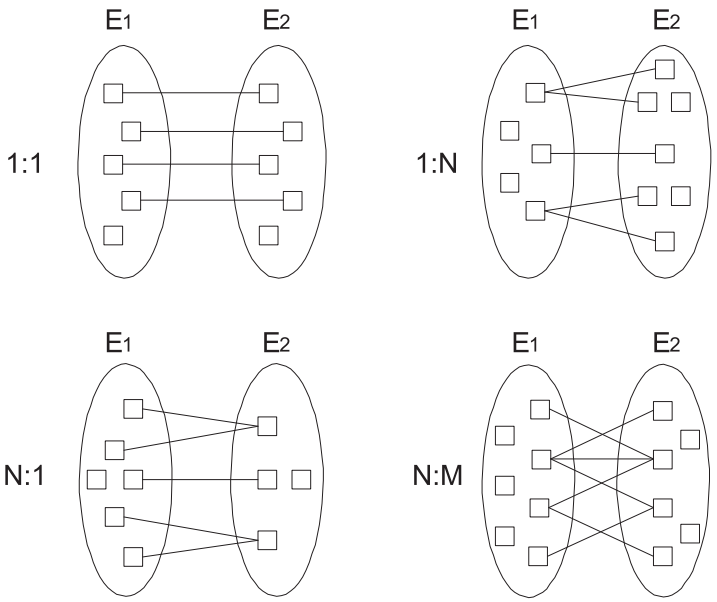


Abb. 7: Mögliche Funktionalitäten von binären Beziehungen

Die binären 1:1-, 1:N- und N:1-Beziehungen kann man auch als *partielle Funktionen* ansehen, welche einige Elemente aus dem Definitionsbereich auf einige Elemente des Wertebereichs abbilden, z. B.

beschäftigt_bei : *Personen→Firmen*

2.4 Die (min, max)-Notation

Bei der *(min, max)*-Notation wird für jedes an einem Beziehungstyp beteiligte Entity ein Paar von Zahlen, nämlich *min* und *max* angegeben. Dieses Zahlenpaar sagt aus, dass jedes Entity dieses Typs mindestens *min*-mal in der Beziehung steht und höchstens *max*-mal. Wenn es Entities geben darf, die gar nicht an der Beziehung teilnehmen, so wird *min* mit 0 angegeben; wenn ein Entity beliebig oft an der Beziehung teilnehmen darf, so wird die *max*-Angabe durch * ersetzt. Somit ist (0,*) die allgemeinste Aussage. Abbildung 8 zeigt die Verwendung der *(min, max)*-Notation anhand der Begrenzungsflächenmodellierung von Polyedern.

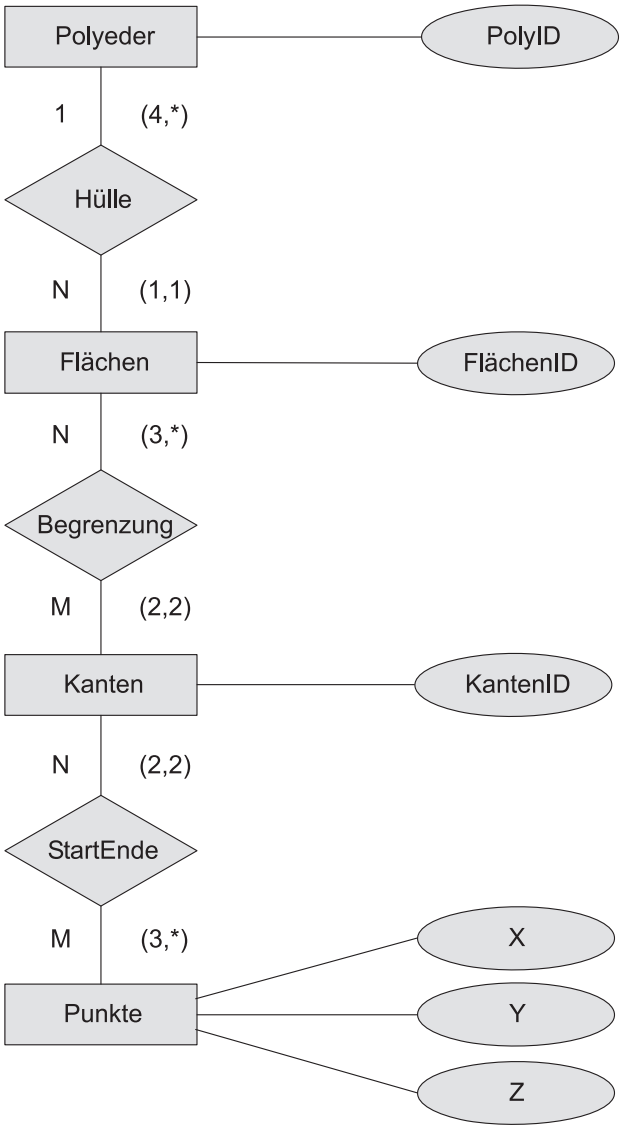


Abb. 8: ER-Diagramm für Begrenzungsflächendarstellung von Polyedern

2.5 Existenzabhängige Entity-Typen

Sogenannte *schwache* Entities können nicht autonom existieren, sondern

- sind in ihrer Existenz von einem anderen, übergeordneten Entity abhängig
- und sind nur in Kombination mit dem Schlüssel des übergeordneten Entity eindeutig identifizierbar.

Abbildung 9 verdeutlicht dieses Konzept anhand von Gebäuden und Räumen. Räume können ohne Gebäude nicht existieren. Die Raumnummern sind nur innerhalb eines Gebäudes eindeutig. Daher wird das entsprechende Attribut gestrichelt markiert. Schwache Entities werden durch doppelt gerahmte Rechtecke repräsentiert und ihre Beziehung zum übergeordneten Entity-Typ durch eine Verdoppelung der Raute und der von dieser Raute zum schwachen Entity-Typ ausgehenden Kante markiert.

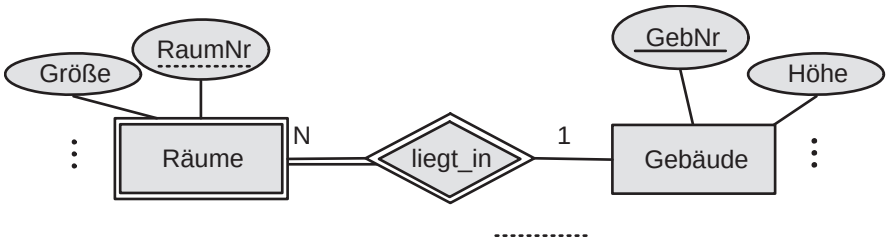


Abb. 9: Ein existenzabhängiger (schwacher) Entity-Typ

2.6 Generalisierung

Zur weiteren Strukturierung der Entity-Typen wird die *Generalisierung* eingesetzt. Hierbei werden Eigenschaften von ähnlichen Entity-Typen einem gemeinsamen *Obertyp* zugeordnet. Bei dem jeweiligen *Untertyp* verbleiben nur die nicht faktorisierbaren Attribute. Somit stellt der Untertyp eine *Spezialisierung* des Obertyps dar. Diese Tatsache wird durch eine Beziehung mit dem Namen **is-a** (ist ein) ausgedrückt, welche durch ein Sechseck, verbunden mit gerichteten Pfeilen symbolisiert wird.

In Abbildung 10 sind *Assistenten* und *Professoren* jeweils Spezialisierungen von *Angestellte* und stehen daher zu diesem Entity-Typ in einer *is-a* Beziehung.

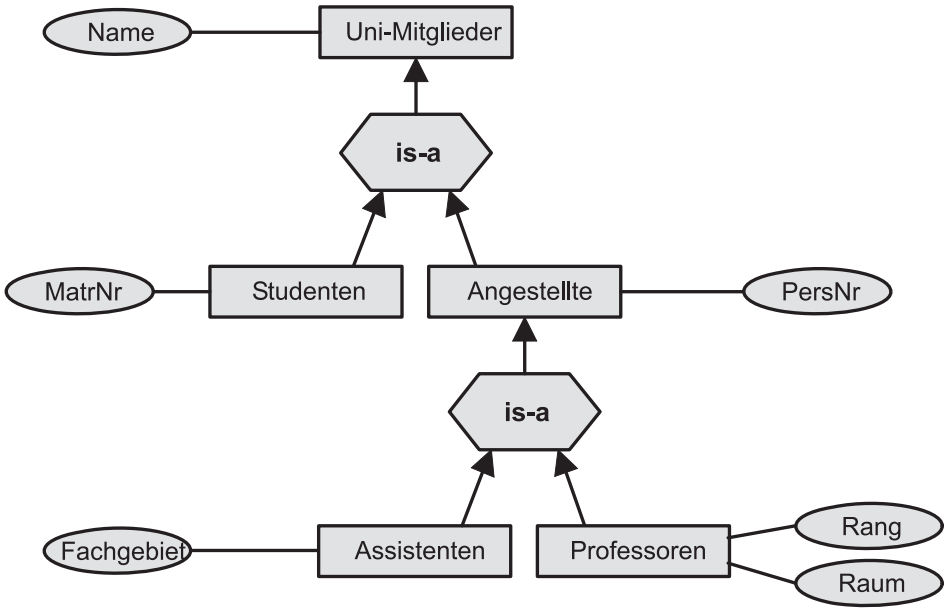


Abb. 10: Spezialisierung der Universitätsmitglieder

Bezüglich der Teilmengensicht ist von Interesse:

- **disjunkte Spezialisierung:**
  die Entitymengen der Untertypen sind paarweise disjunkt
- **vollständige Spezialisierung:**
  die Obermenge enthält keine direkten Elemente, sondern setzt sich komplett aus der Vereinigung der Entitymengen der Untertypen zusammen.

In Abbildung 10 ist die Spezialisierung von *Uni-Mitglieder* vollständig und disjunkt, die Spezialisierung von *Angestellte* ist disjunkt, aber nicht vollständig, da es noch andere, nichtwissenschaftliche Angestellte (z.B. Sekretärinnen) gibt.

2.7 Aggregation

Durch die *Aggregation* werden einem übergeordneten Entity-Typ mehrere untergeordnete Entity-Typen zugeordnet. Diese Beziehung wird als *part-of* (Teil von) bezeichnet, um zu betonen, dass die untergeordneten Entities Bestandteile der übergeordneten Entities sind. Um eine Verwechselung mit dem Konzept der Generalisierung zu vermeiden, verwendet man nicht die Begriffe *Obertyp* und *Untertyp*.

Abbildung 11 zeigt die Aggregationshierarchie eines Fahrrads. Zum Beispiel sind *Rohre* und *Lenker* Bestandteile des *Rahmen*; *Felgen* und *Speichen* sind Bestandteile der *Räder*.

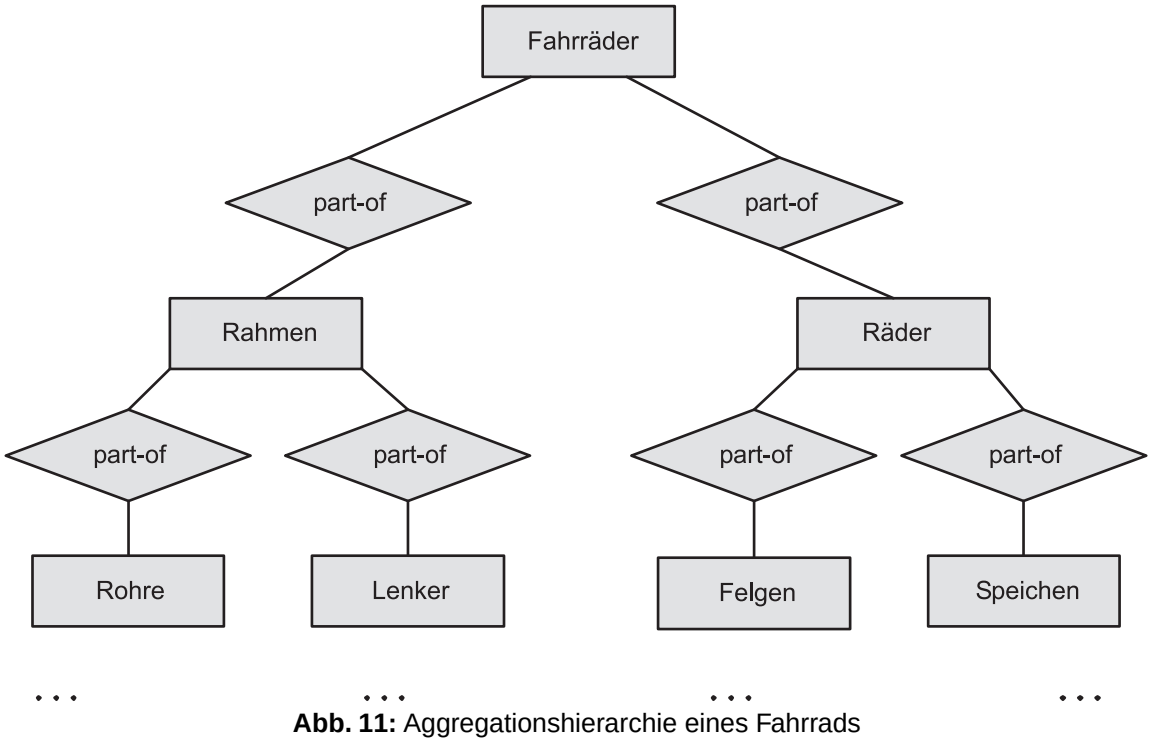


Abb. 11: Aggregationshierarchie eines Fahrrads

2.8 Konsolidierung

Bei der Modellierung eines komplexeren Sachverhaltes bietet es sich an, den konzeptuellen Entwurf zunächst in verschiedene Anwendersichten aufzuteilen. Nachdem die einzelnen Sichten modelliert sind, müssen sie zu einem globalen Schema zusammengefasst werden (Abbildung 12).

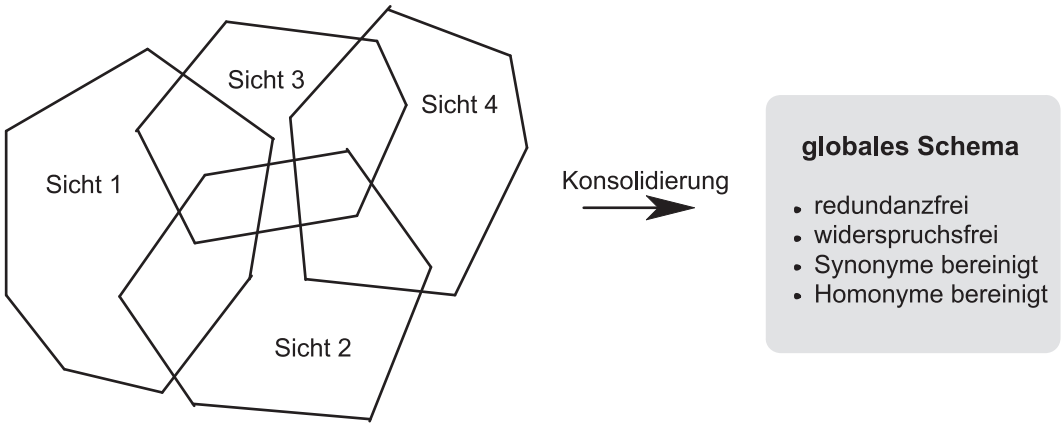
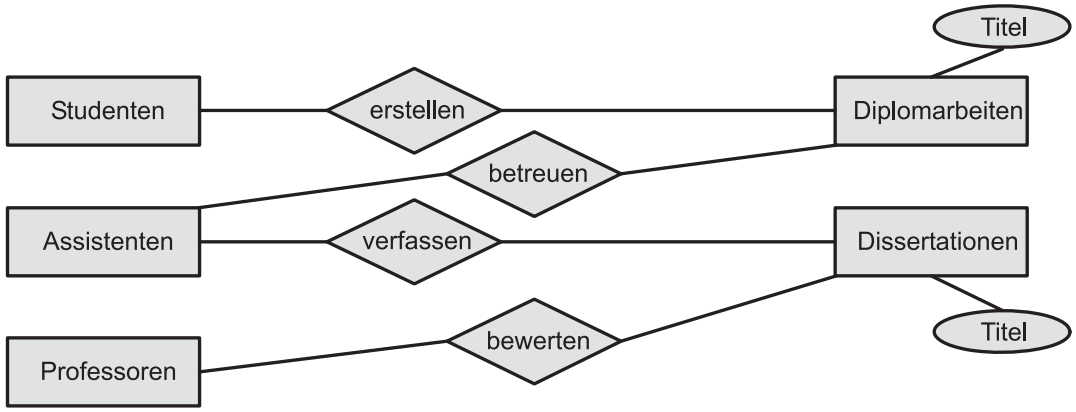


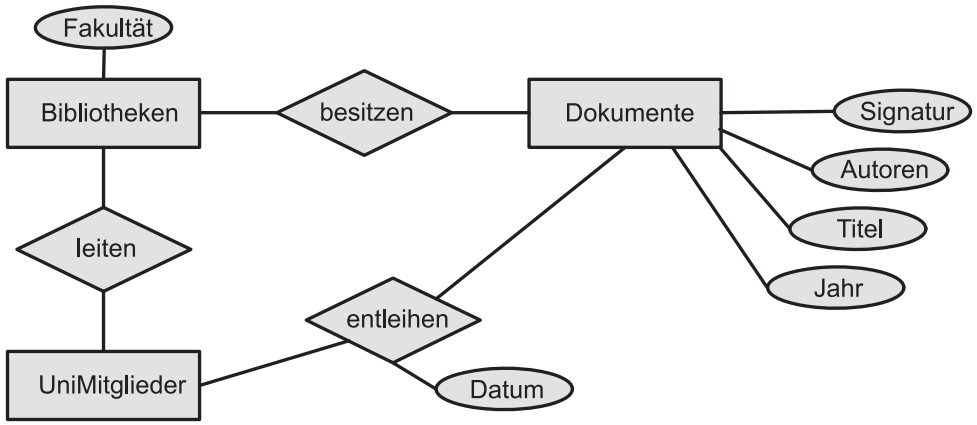
Abb. 12: Konsolidierung überlappender Sichten

Probleme entstehen dadurch, dass sich die Datenbestände der verschiedenen Anwender teilweise überlappen. Daher reicht es nicht, die einzelnen konzeptuellen Schemata zu vereinen, sondern sie müssen *konsolidiert* werden.

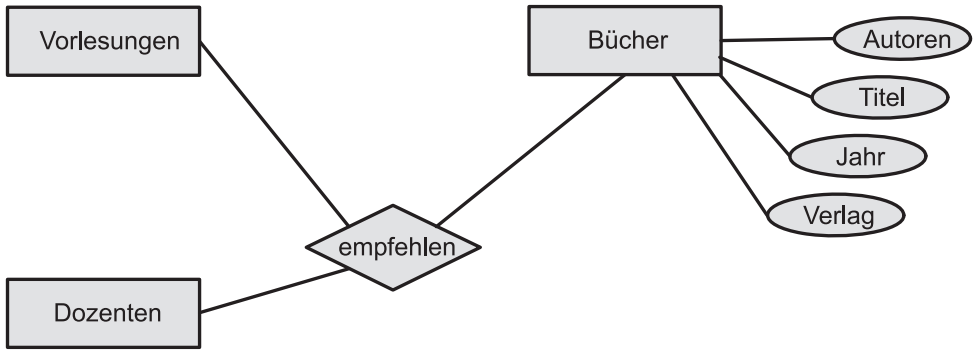
Darunter versteht man das Entfernen von Redundanzen und Widersprüchen. Widersprüche entstehen durch *Synonyme* (gleiche Sachverhalte wurden unterschiedlich benannt) und durch *Homonyme* (unterschiedliche Sachverhalte wurden gleich benannt) sowie durch unterschiedliches Modellieren desselben Sachverhalts zum einen über Beziehungen, zum anderen über Attribute. Bei der Zusammenfassung von ähnlichen Entity-Typen zu einem Obertyp bietet sich die Generalisierung an.



Sicht 1: Erstellung von Dokumenten als Prüfungsleistung



Sicht 2: Bibliotheksverwaltung



Sicht 3: Buchempfehlungen für Vorlesungen

Abb. 13: Drei Sichten einer Universitätsdatenbank

Abbildung 13 zeigt drei Sichten einer Universitätsdatenbank. Für eine Konsolidierung sind folgende Beobachtungen wichtig:

- *Professoren* und *Dozenten* werden synonym verwendet.
- *UniMitglieder* ist eine Generalisierung von *Studenten*, *Professoren* und *Assistenten*.
- *Bibliotheken* werden nicht von beliebigen *UniMitglieder* geleitet, sondern nur von *Angestellte*.

- *Dissertationen*, *Diplomarbeiten* und *Bücher* sind Spezialisierungen von *Dokumente*.
- Die Beziehungen *erstellen* und *verfassen* modellieren denselben Sachverhalt wie das Attribut *Autor*.

Abbildung 14 zeigt das Ergebnis der Konsolidierung. Generalisierungen sind zur Vereinfachung als fettgedruckte Pfeile dargestellt. Das ehemalige Attribut *Autor* ist nun als Beziehung zwischen Dokumenten und Personen modelliert. Zu diesem Zweck ist ein neuer Entity-Typ *Personen* erforderlich, der *UniMitglieder* generalisiert. Damit werden die ehemaligen Beziehungen *erstellen* und *verfassen* redundant. Allerdings geht im konsolidierten Schema verloren, dass Diplomarbeiten von *Studenten* und *Dissertationen* von *Assistenten* geschrieben werden.

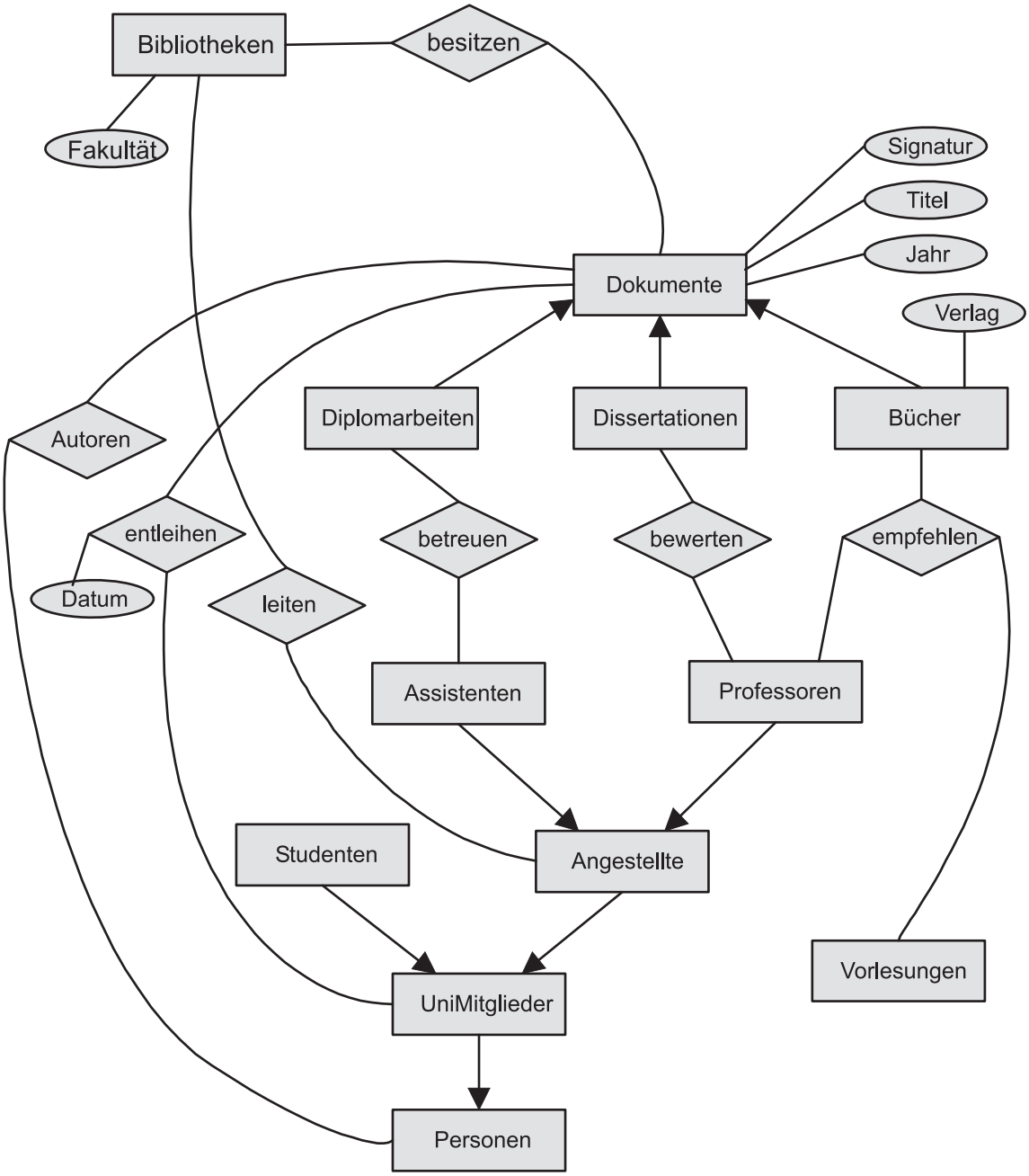


Abb. 14: Konsolidiertes Schema der Universitätsdatenbank

2.9 UML

Im Bereich Software Engineering hat sich die objektorientierte Modellierung mit Hilfe von *UML (Unified Modelling Language)* durchgesetzt. Dieser Formalismus lässt sich auch für den Datenbankentwurf benutzen.

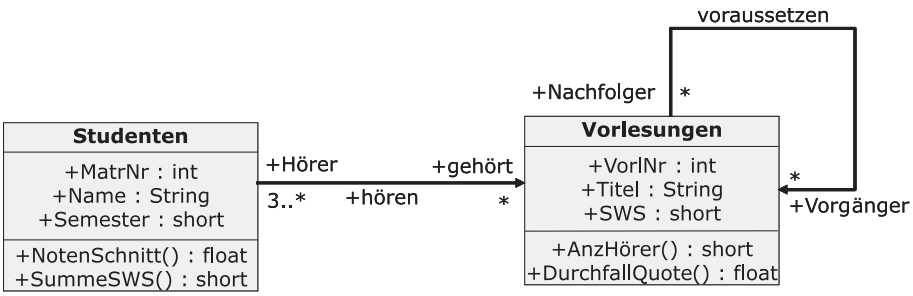


Abb. 15: UML: Klassen und Assoziationen

Abbildung 15 zeigt die beiden Klassen `Studenten` und `Vorlesungen`, die neben den Datenfeldern auch Methoden aufweisen, welche das Verhalten beschreiben. Öffentlich sichtbare Komponenten werden mit + gekennzeichnet; Methoden sind an den beiden Klammern () zu erkennen und Datentypen werden nach dem Doppelpunkt : genannt. Beziehungen zwischen den Klassen werden durch Assoziationen ausgedrückt, welche Aufgrund der Implementierung durch Referenzen über eine Richtung verfügen: So lassen sich effizient zu einem Studenten die Vorlesungen ermitteln, umgekehrt ist das nicht (so leicht) möglich. Am Ende der Pfeile sind jeweils die Rollen vermerkt: `Hörer`, `gehört`, `Nachfolger` und `Vorgänger`. Die sogenannte *Multiplizität* drückt die von der (min,max)-Notation bekannte Komplexität aus, allerdings jeweils am anderen Ende notiert: mindestens drei Hörer sitzen in einer Vorlesung; ein Student kann beliebig viele Vorlesungen besuchen.

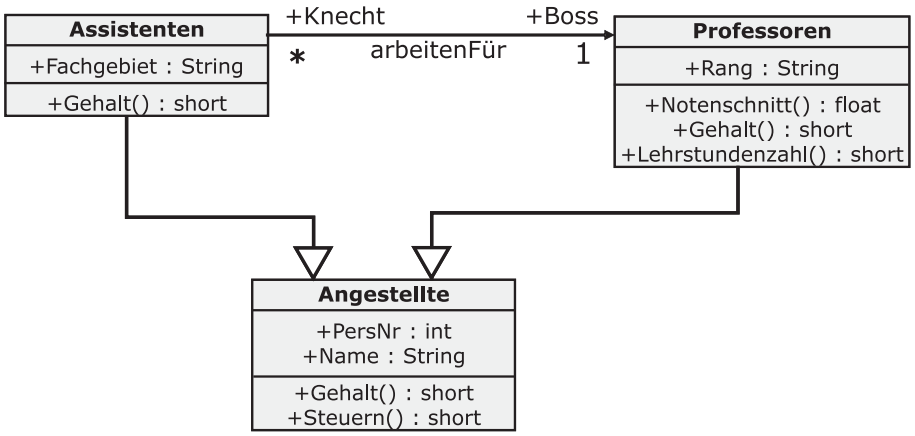


Abb. 16: UML: Generalisierung

In der Abbildung 16 werden `Assistenten` und `Professoren` zu `Angestellten` verallgemeinert. Die gemeinsamen Datenfelder lauten `PersNr` und `Name`. Da sich die Berechnung des Gehaltes bei Assistenten und Professoren unterscheidet, erhalten sie beide ihre eigenen Methoden dafür; sie verfeinern dadurch die in der Oberklasse vorhandene Methode `Gehalt()`. Die Steuern hingegen werden bei beiden nach demselben Algorithmus berechnet, daher reicht eine Methode `Steuern()` in der Oberklasse.

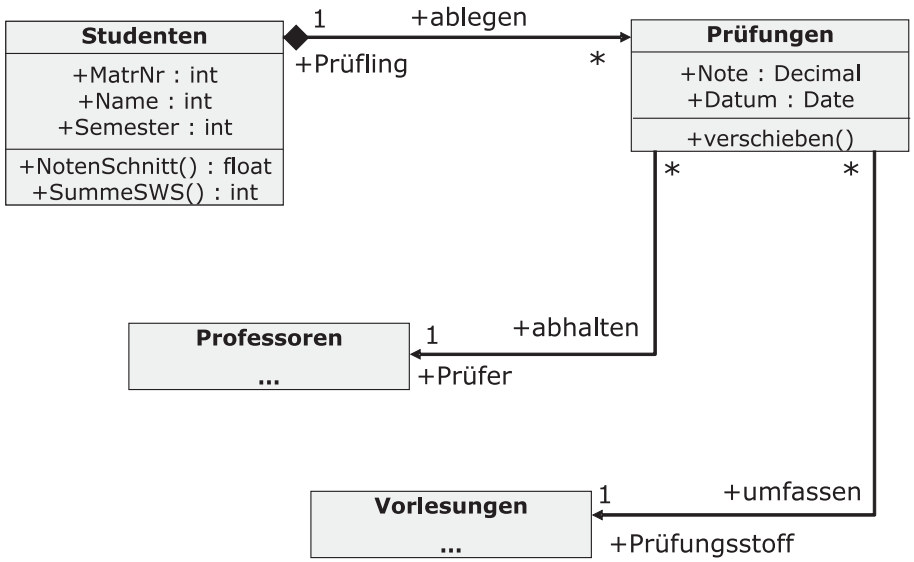


Abb. 17: UML: Aggregation

Durch eine schwarze Raute wird in Abbildung 17 eine Komposition ausgedrückt, welche die Modellierung eines schwachen Entity-Typen übernimmt. Die Beziehung zwischen `Studenten` und `Prüfungen` wird durch eine Assoziation mit dem Namen `ablegen` modelliert; der `Student` übernimmt dabei die Rolle eines `Prüfling`. Die Beziehung zwischen `Professoren` und `Prüfungen` wird durch eine Assoziation mit dem Namen `abhalten` modelliert; der Professor übernimmt dabei die Rolle des `Prüfer`. Die Beziehung zwischen `Vorlesungen` und `Prüfungen` wird durch eine Assoziation mit dem Namen `umfassen` modelliert; die Vorlesung übernimmt dabei die Rolle des `Prüfungsstoff`.

3. Logische Datenmodelle

In Abhängigkeit von dem zu verwendenden Datenbanksystem wählt man zur computergerechten Umsetzung des Entity-Relationship-Modells das hierarchische, das netzwerkorientierte, das relationale oder das objektorientierte Datenmodell.

3.1 Das Hierarchische Datenmodell

Datenbanksysteme, die auf dem hierarchischen Datenmodell basieren, haben (nach heutigen Standards) nur eine eingeschränkte Modellierfähigkeit und verlangen vom Anwender Kenntnisse der interne Ebene. Trotzdem sind sie noch sehr verbreitet (z.B. IMS von IBM), da sie sich aus Dateiverwaltungssystemen für die konventionelle Datenverarbeitung entwickelt haben. Die zugrunde liegende Speicherstruktur waren Magnetbänder, welche nur sequentiellen Zugriff erlaubten.

Im Hierarchischen Datenmodell können nur baumartige Beziehungen modelliert werden. Eine Hierarchie besteht aus einem Wurzel-Entity-Typ, dem beliebig viele Entity-Typen unmittelbar untergeordnet sind; jedem dieser Entity-Typen können wiederum Entity-Typen untergeordnet sein usw. Alle Entity-Typen eines Baumes sind verschieden.

Abbildung 18 zeigt ein hierarchisches Schema sowie eine mögliche Ausprägung anhand der bereits bekannten Universitätswelt. Der konstruierte Baum ordnet jedem Studenten alle Vorlesungen zu, die er besucht, sowie alle Professoren, bei denen er geprüft wird. In dem gezeigten Baum ließen sich weitere Teilbäume unterhalb der *Vorlesung* einhängen, z.B. die Räumlichkeiten, in denen Vorlesungen stattfinden. Obacht: es wird keine Beziehung zwischen den Vorlesungen und Dozenten hergestellt! Die Dozenten sind den Studenten ausschließlich in ihrer Eigenschaft als Prüfer zugeordnet.

Grundsätzlich sind einer Vater-Ausprägung (z.B. `Erika Mustermann`) für jeden ihrer Sohn-Typen jeweils mehrere Sohnausprägungen zugeordnet (z.B. könnte der Sohn-Typ `Vorlesung` 5 konkrete Vorlesungen enthalten). Dadurch entsprechen dem Baum auf Typ-Ebene mehrere Bäume auf Entity-Ebene. Diese Entities sind in Preorder-Reihenfolge zu erreichen, d.h. vom Vater zunächst seine Söhne und Enkel und dann dessen Brüder. Dieser Baumdurchlauf ist die einzige Operation auf einer Hierarchie; jedes Datum kann daher nur über den Einstiegspunkt Wurzel und von dort durch Überlesen nichtrelevanter Datensätze gemäß der Preorder-Reihenfolge erreicht werden.

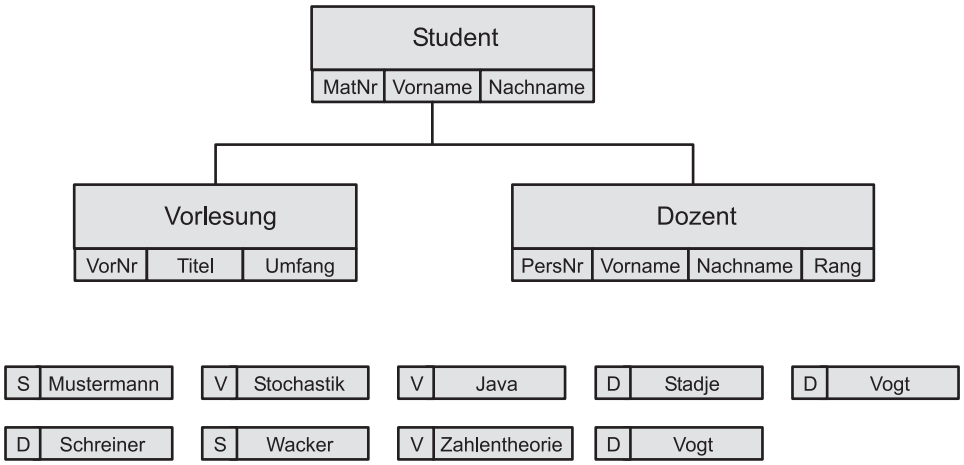


Abb. 18: Hierarchisches Schema und eine Ausprägung. Der Recordtyp sei erkennbar an den Zeichen S (Studenten), V (Vorlesungen) und D (Dozenten)

Die typische Operation besteht aus dem Traversieren der Hierarchie unter Berücksichtigung der jeweiligen Vaterschaft, d. h. der Befehl `GNP VORLESUNG` (gesprochen: GET NEXT VORLESUNG WITHIN PARENT) durchläuft sequentiell ab der aktuellen Position die Preorder-Sequenz solange vorwärts, bis ein dem aktuellen Vater zugeordneter Datensatz vom Typ *Vorlesung* gefunden wird.

Um zu vermeiden, dass alle Angaben zu den Dozenten mehrfach gespeichert werden, kann eine eigene Hierarchie für die Dozenten angelegt und in diese dann aus dem Studentenbaum heraus verwiesen werden.

Es folgt ein umfangreicheres Beispiel, entnommen dem Buch von C.J. Date. Abbildung 19 zeigt das hierarchische Schema, Abbildung 20 eine Ausprägung.

Die Query *Welche Studenten besuchen den Kurs M23 am 13.08.1973?* wird unter Verwendung der DML (Data Manipulation Language) derart formuliert, dass zunächst nach Einstieg über die Wurzel der Zugriff auf den gesuchten Kurs stattfindet und ihn als momentanen Vater fixiert. Von dort werden dann genau solche Records vom Typ *Student* durchlaufen, welche den soeben fixierten Kursus als Vater haben:

```
GU COURSE   (COURSE#='M23')
   OFFERING (DATE='730813');
if gefunden then
begin
  GNP STUDENT;
  while gefunden do
  begin
    write (STUDENT.NAME);
    GNP STUDENT
  end
end;
```

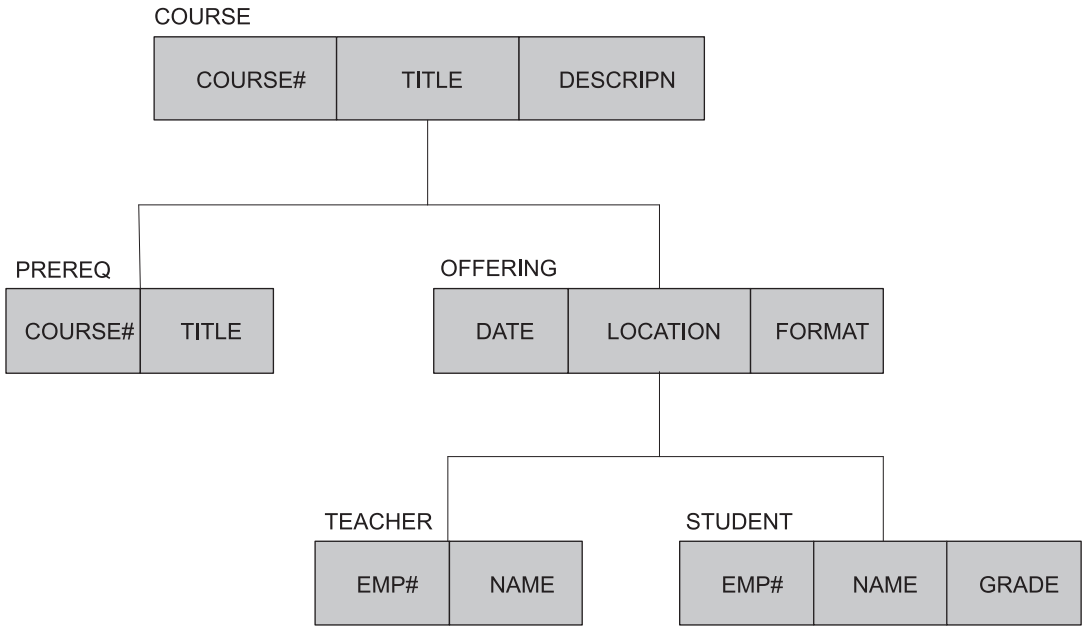


Abb. 19: Beispiel für ein hierarchisches Schema

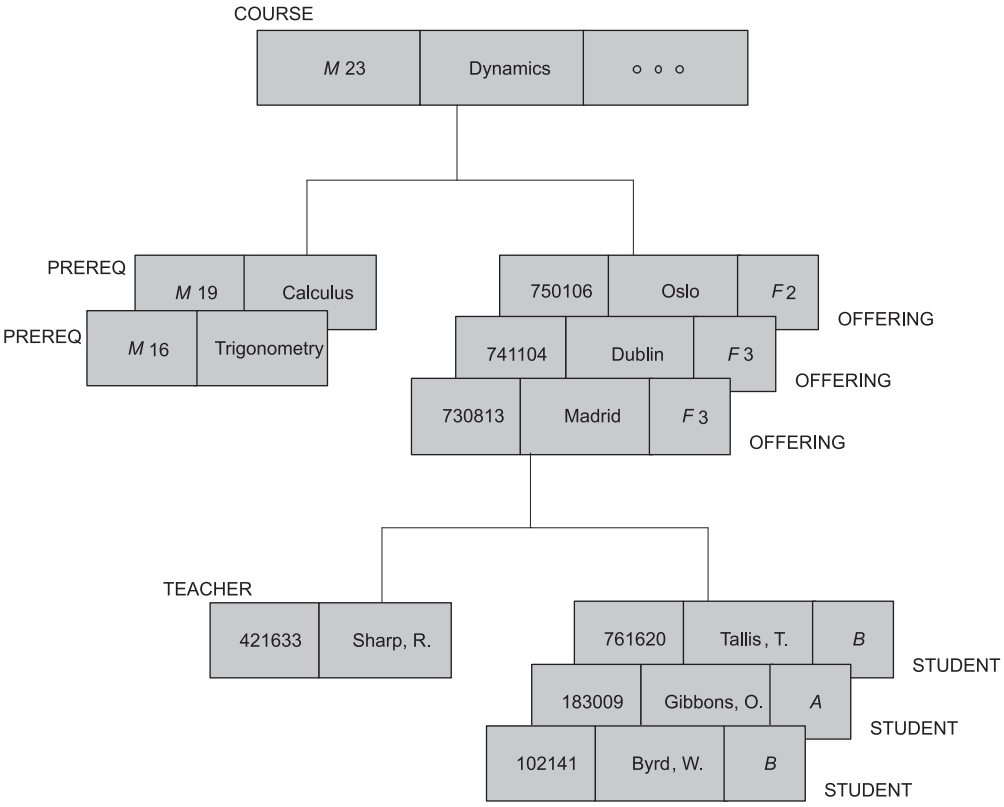


Abb. 20: Beispiel für eine Ausprägung des hierarchischen Schemas

3.2 Das Netzwerk-Datenmodell

Im Netzwerk-Datenmodell können nur binäre many-one- (bzw. one-many)-Beziehungen dargestellt werden. Ein E-R-Diagramm mit dieser Einschränkung heißt *Netzwerk*. Zur Formulierung der many-one-Beziehungen gibt es sogenannte *Set-Typen*, die zwei Entity-Typen in Beziehung setzen. Ein Entity-Typ übernimmt mittels eines Set-Typs die Rolle des *owner* bzgl. eines weiteren Entity-Typs, genannt *member*.

Im Netzwerk werden die Beziehungen als gerichtete Kanten gezeichnet vom Rechteck für *member* zum Rechteck für *owner* (funktionale Abhängigkeit). In einer Ausprägung führt ein gerichteter Ring von einer *owner*-Ausprägung über alle seine *member*-Ausprägungen (Abbildung 21).

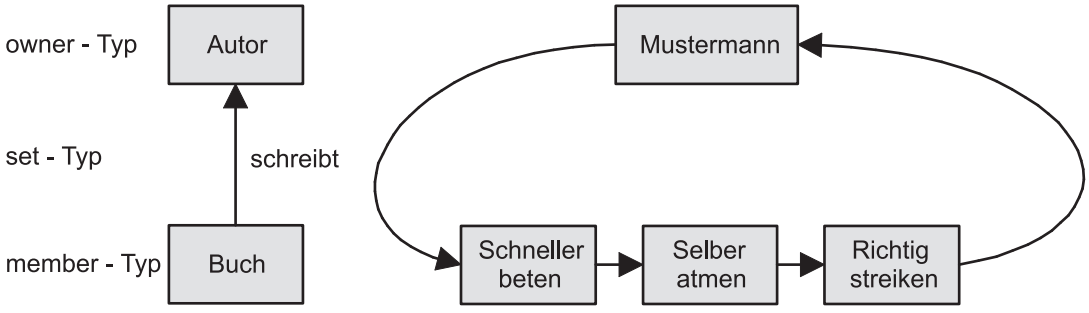


Abb. 21: Netzwerkschema und eine Ausprägung

Bei nicht binären Beziehungen oder nicht many-one-Beziehungen hilft man sich durch Einführung von künstlichen *Kett-Records*. Abbildung 22 zeigt ein entsprechendes Netzwerkschema und eine Ausprägung, bei der zwei Studenten jeweils zwei Vorlesungen hören.

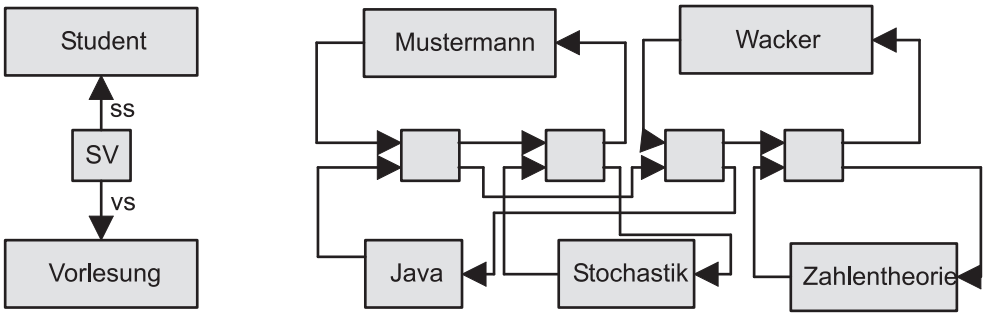


Abb. 22: Netzwerkschema mit Kett-Record und eine Ausprägung

Die typische Operation auf einem Netzwerk besteht in der Navigation durch die verzeigerten Entities. Mit den Befehlen

```
FIND NEXT Student
FIND NEXT sv WITHIN ss
FIND OWNER WITHIN vs
```

lassen sich für einen konkreten Studenten alle seine Kett-Records vom Typ `sv` durchlaufen und dann jeweils der *owner* bzgl. des Sets `vs` ermitteln.

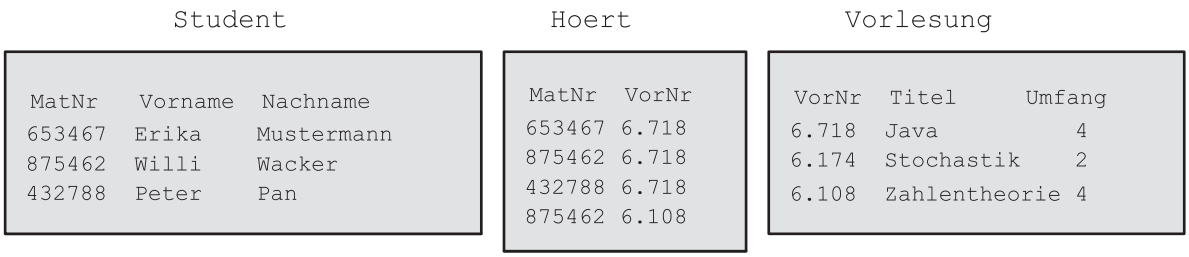
```
STUDENT.NAME := 'Mustermann';
FIND ANY STUDENT USING NAME;
if gefunden then
begin
  FIND FIRST SV WITHIN SS;
  while gefunden do
  begin
    FIND OWNER WITHIN VS;
    GET VORLESUNG;
    WRITE(VORLESUNG.TITEL);
    FIND NEXT VORLESUNG WITHIN VS;
  end
end;
```

3.3 Das Relationale Datenmodell

Seien $D_1, D_2, \ldots, D_k$ Wertebereiche. $R \subseteq D_1 \times D_2 \times \ldots \times D_k$ heißt Relation. Wir stellen uns eine Relation als Tabelle vor, in der jede Zeile einem Tupel entspricht und jede Spalte einem bestimmten Wertebereich. Die Folge der Spaltenidentifizierungen heißt *Relationenschema*. Eine Menge von Relationenschemata heißt *relationales Datenbankschema*, die aktuellen Werte der einzelnen Relationen ergeben eine Ausprägung der relationalen Datenbank.

- pro Entity-Typ gibt es ein Relationenschema mit Spalten benannt nach den Attributen.
- pro Relationshiptyp gibt es ein Relationenschema mit Spalten für die Schlüssel der beteiligten Entity-Typen und ggf. weitere Spalten.

Abbildung 23 zeigt ein Schema zum Vorlesungsbetrieb und eine Ausprägung. Hierbei wurden die über ihre Tabellengrenzen hinausgehenden und zueinander passenden Attribute jeweils gleich genannt, um zu verdeutlichen, dass sie Daten mit derselben Bedeutung speichern.



Student

MatNr	Vorname	Nachname
653467	Erika	Mustermann
875462	Willi	Wacker
432788	Peter	Pan

Hoert

MatNr	VorNr
653467	6.718
875462	6.718
432788	6.718
875462	6.108

Vorlesung

VorNr	Titel	Umfang
6.718	Java	4
6.174	Stochastik	2
6.108	Zahlentheorie	4

Abb. 23: Relationales Schema und eine Ausprägung

Die typischen Operationen auf einer relationaler Datenbank lauten:

- **Selektion:** Suche alle Tupel einer Relation mit gewissen Attributeigenschaften
- **Projektion:** Filtere gewisse Spalten heraus
- **Verbund:** Finde Tupel in mehreren Relationen, die bzgl. gewisser Spalten übereinstimmen.

Beispiel-Query: Welche Studenten hören die Vorlesung Zahlentheorie?

```
SELECT Student.Name
FROM   Student, Hoert, Vorlesung
WHERE  Student.MatrNr   = Hoert.MatNr
  AND  Hoert.VorlNr     = Vorlesung.VorNr
  AND  Vorlesung.Titel  = "Zahlentheorie"
```

3.4 Das Objektorientierte Modell

Eine Klasse repräsentiert einen Entity-Typ zusammen mit einer Struktur- und Verhaltensbeschreibung, welche ggf. an Unterklassen vererbt werden können. Binäre Beziehungen können durch mengenwertige Attribute modelliert werden:

```
class Studenten {
  attribute long   Matrnr;
  attribute String Name;
  relationship set <Vorlesungen> hoert inverse Vorlesungen::Hoerer
}


class Professoren {
  attribute long   PersNr;
  attribute String Name;
  relationship set <Vorlesungen> liest inverse Vorlesungen::gelesenVon
}


class Vorlesungen {
  attribute long   VorlNr;
  attribute String Titel;
  relationship Professoren gelesenVon inverse Professoren::liest
  relationship set <Studenten> Hoerer inverse Studenten::hoert
}
```

Beispiel-Query: Welche Studenten besuchen Vorlesungen von Sokrates ?

```
select s.Name
from   s in AlleStudenten, v in s.hoert
where  v.gelesenVon.Name = "Sokrates"
```

4. Physikalische Datenorganisation

4.1 Grundlagen

Bedingt durch die unterschiedlichen Speichertechnologien weisen Hauptspeicher, Festplatte und Magnetband charakteristische Vor- und Nachteile auf. Folgende Tabelle zeigt die relativen Merkmale bezüglich Größe, Zugriffsgeschwindigkeit, Preis, Granularität und Dauerhaftigkeit. Im Vergleich zum direkt adressierbaren Hauptspeicher ist eine typische Festplatte etwa 1.000 mal größer, verursacht einen etwa 100.000 mal langsameren Zugriff und kostet nur ein Hundertstel bezogen auf ein gespeichertes Byte.

	Primärspeicher	Sekundärspeicher	Tertiärspeicher
Größe	klein	groß [10^3]	sehr groß
Tempo	schnell	langsam [10^{-5}]	sehr langsam
Preis	teuer	billig [10^{-2}]	billig
Granularität	fein	grob	grob
Dauerhaftigkeit	flüchtig	stabil	stabil

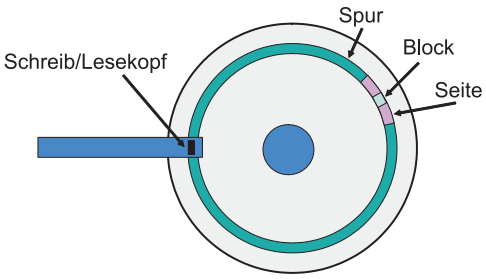


Abb. 24: Schematischer Festplattenaufbau: Sicht von oben

Abbildung 24 zeigt den schematischen Aufbau einer Festplatte. Zum Lesen eines Blockes muss zunächst der Schreib-/Lesekopf oberhalb der zuständigen Spur positioniert werden (Seek Time), dann wird gewartet, bis der betreffende Block vorbeisaust (Latency Time), und schließlich kann der Block übertragen werden (Transfer Time). Oft werden mehrere Blöcke nur zusammengefasst auf einer Seite übertragen.

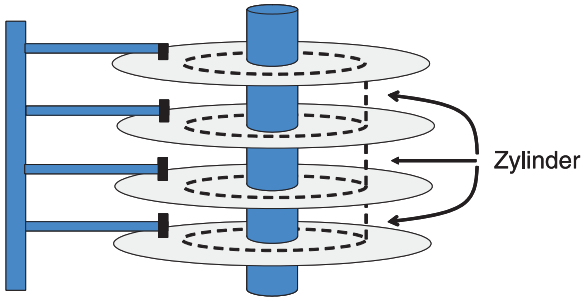


Abb. 25: Schematischer Festplattenaufbau: Sicht von der Seite

Abildung 25 verdeutlicht, dass der Lesearm mehrere starr miteinander verbundene Schreib-/Leseköpfe gemeinsam bewegen und somit auf die jeweils übereinanderliegenden Spuren aller Magnetscheiben (genannt: Zylinder) gleichzeitig zugreifen kann. Der Block als kleinste direkt adressierbare Speichereinheit spielt daher für alle Datenstrukturen und Suchalgorithmen die zentrale Rolle.

Die grundsätzliche Aufgabe bei der Realisierung eines internen Modells besteht aus dem Abspeichern von Datentupeln, genannt *Records*, in einem *File*. Jedes Record hat ein festes Record-Format und besteht aus mehreren Feldern meistens fester, manchmal auch variabler Länge mit zugeordnetem Datentyp. Folgende Operationen sind erforderlich:

- **INSERT:** Einfügen eines Records
- **DELETE:** Löschen eines Records
- **MODIFY:** Modifizieren eines Records
- **LOOKUP:** Suchen eines Records mit bestimmtem Wert in bestimmten Feldern.

Files werden abgelegt im Hintergrundspeicher (Magnetplatte), der aus *Blöcken* fester Größe (etwa 2^9 - 2^{12} Bytes) besteht, die direkt adressierbar sind. Ein File ist verteilt über mehrere Blöcke, ein Block enthält mehrere Records. Records werden nicht über Blockgrenzen verteilt. Einige Bytes des Blockes sind unbenutzt, einige werden für den *header* gebraucht, der Blockinformationen (Verzeigerung, Record-Interpretation) enthält.

Die *Adresse* eines Records besteht entweder aus der Blockadresse und einem *Offset* (Anzahl der Bytes vom Blockanfang bis zum Record) oder wird durch den sogenannten *Tupel-Identifikator* (TID) gegeben. Der Tupel-Identifikator besteht aus der Blockadresse und einer Nummer eines Eintrags in der internen Datensatztabelle, der auf das entsprechende Record verweist. Sofern genug Information bekannt ist, um ein Record im Block zu identifizieren, reicht auch die Blockadresse. Blockzeiger und Tupel-Identifikatoren erlauben das Verschieben der Records im Block (*unpinned records*), Record-Zeiger setzen fixierte Records voraus (*pinned records*), da durch Verschieben eines Records Verweise von außerhalb missinterpretiert würden (*dangling pointers*).

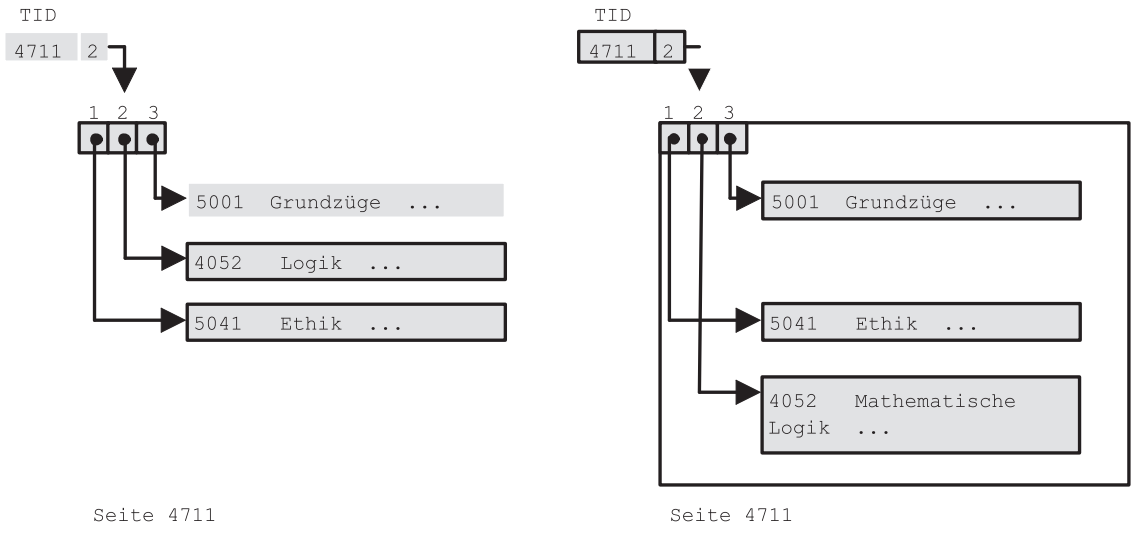


Abb. 26: Verschieben eines Tupels innerhalb einer Seite

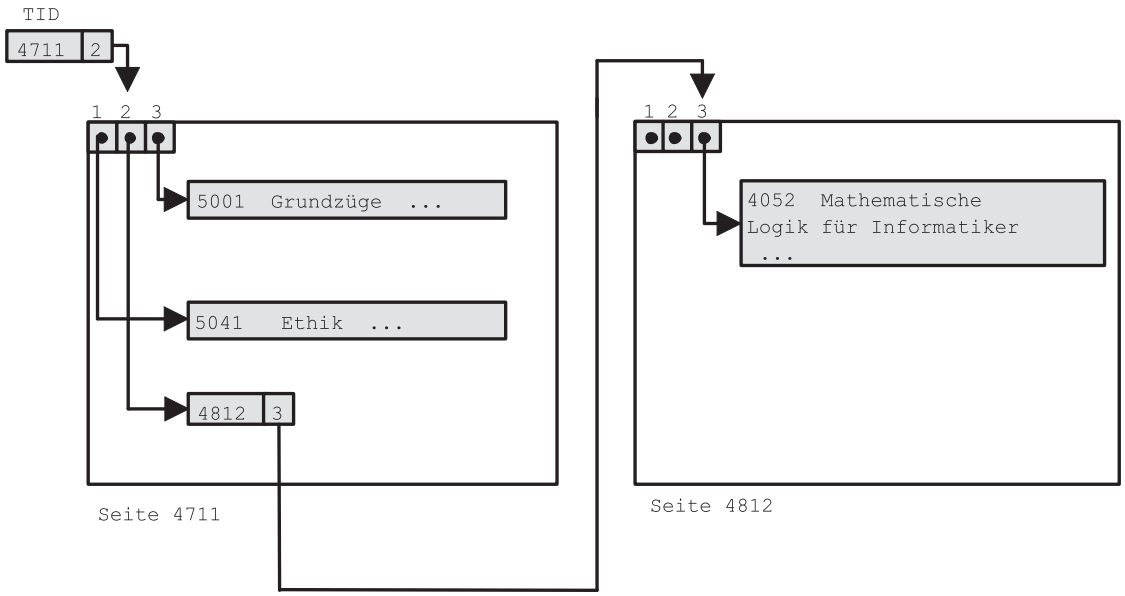


Abb. 27: Verdrängen eines Tupels von einer Seite

Abbildung 26 zeigt das Verschieben eines Datentupels innerhalb seiner ursprünglichen Seite; in Abbildung 27 wird das Record schließlich auf eine weitere Seite verdrängt.

Das Lesen und Schreiben von Records kann nur im Hauptspeicher geschehen. Die Blockladezeit ist deutlich größer als die Zeit, die zum Durchsuchen des Blockes nach bestimmten Records gebraucht wird. Daher ist für Komplexitätsabschätzungen nur die Anzahl der Blockzugriffe relevant. Zur Umsetzung des Entity-Relationship-Modells verwenden wir

- Records für Entities
- Records für Relationships (pro konkrete Beziehung ein Record mit TID-Tupel)

4.2 Heap-File

Die einfachste Methode zur Abspeicherung eines Files besteht darin, alle Records hintereinander zu schreiben. Die Operationen arbeiten wie folgt:

- **INSERT:** Record am Ende einfügen (ggf. überschriebene Records nutzen)
- **DELETE:** Lösch-Bit setzen
- **MODIFY:** Record überschreiben
- **LOOKUP:** Gesamtes File durchsuchen

Bei großen Files ist der lineare Aufwand für LOOKUP nicht mehr vertretbar. Gesucht ist daher eine Organisationsform, die

- ein effizientes LOOKUP erlaubt,
- die restlichen Operationen nicht ineffizient macht,
- wenig zusätzlichen Platz braucht.

4.3 Hashing

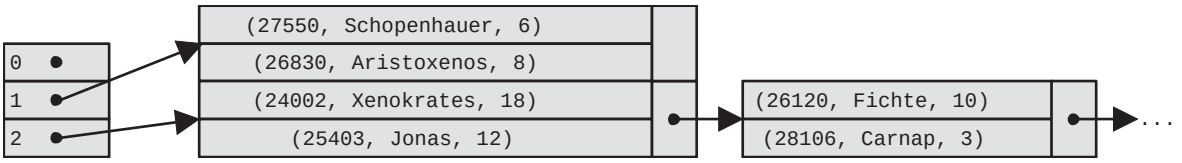


Abb. 28: Hash-Tabelle mit Einstieg in Behälter

Die grundlegende Idee beim *offenen Hashing* ist es, die Records des Files auf mehrere Behälter (englisch: *Bucket*) aufzuteilen, die jeweils aus einer Folge von verzeigerten Blöcken bestehen. Es gibt eine *Hash-Funktion* h, die einen Schlüssel als Argument erhält und ihn auf die Bucket-Nummer abbildet, unter der der Block gespeichert ist, welcher das Record mit diesem Schlüssel enthält. Sei B die Menge der Buckets, sei V die Menge der möglichen Record-Schlüssel, dann gilt gewöhnlich $|V| \gg |B|$.

Beispiel für eine Hash-Funktion: Fasse den Schlüssel v als k Gruppen von jeweils n Bits auf. Sei d_i die i-te Gruppe als natürliche Zahl interpretiert. Setze

$$h(v) = \left(\sum_{i=1}^{k} d_i \right) \mathrm{mod} B$$

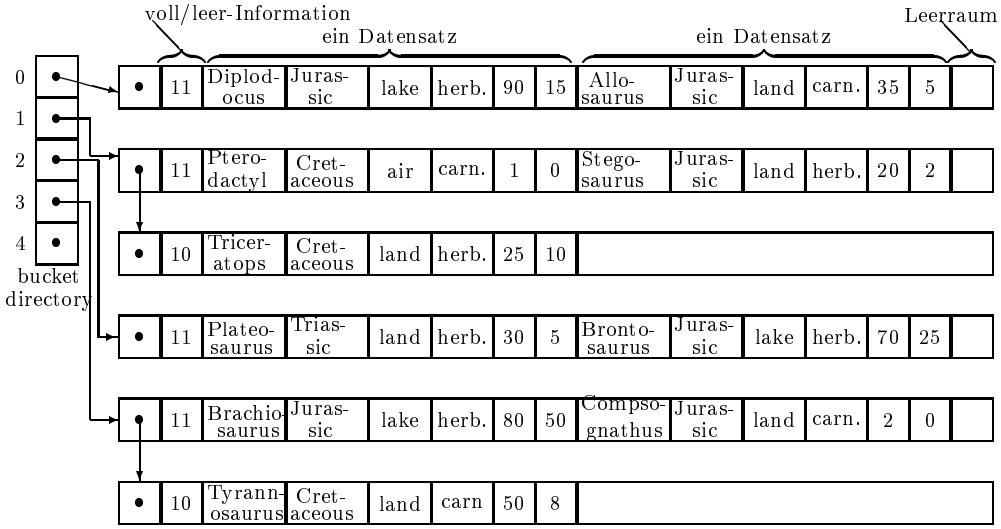


Abb. 29: Hash-Organisation vor Einfügen von Elasmosaurus

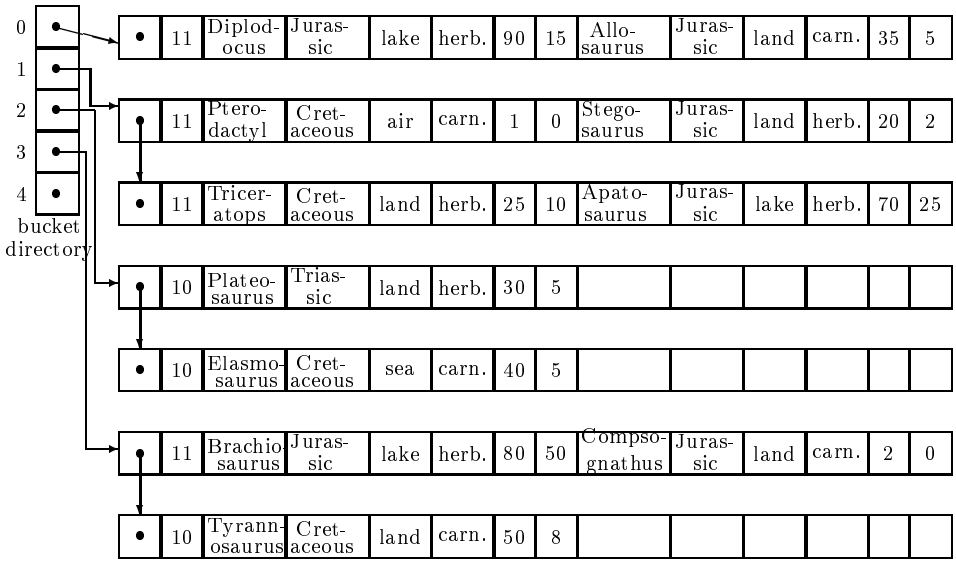


Abb. 30: Hash-Organisation nach Einfügen von Elasmosaurus und Umbenennen

Im Bucket-Directory findet sich als $h(v)$-ter Eintrag der Verweis auf den Anfang einer Liste von Blöcken, unter denen das Record mit Schlüssel v zu finden ist. Abbildung 28 zeigt eine Hash-Tabelle, deren Hash-Funktion h die Personalnummer x durch $h(x) = x \bmod 3$ auf das Intervall $[0..2]$ abbildet.

Falls B klein ist, kann sich das Bucket-Directory im Hauptspeicher befinden; andernfalls ist es über mehrere Blöcke im Hintergrundspeicher verteilt, von denen zunächst der zuständige Block geladen werden muss.

Jeder Block enthält neben dem Zeiger auf den Folgeblock noch jeweils 1 Bit pro Subblock (Platz für ein Record), welches angibt, ob dieser Subblock leer (also beschreibbar) oder gefüllt (also lesbar) ist. Soll die Möglichkeit von *dangling pointers* grundsätzlich ausgeschlossen werden, müssten gelöschte Records mit einem weiteren, dritten Zustand versehen werden, der dafür sorgt, dass dieser Speicherplatz bis zum generellen Aufräumen nicht wieder verwendet wird.

Zu einem Record mit Schlüssel v laufen die Operationen wie folgt ab:

- **LOOKUP:** Berechne $h(v) = i$. Lies den für i zuständigen Directory-Block ein, und beginne bei der für i vermerkten Startadresse mit dem Durchsuchen aller Blöcke.
- **MODIFY:** Falls Schlüssel von Änderung betroffen: DELETE und INSERT durchführen. Falls Schlüssel von Änderung nicht betroffen: LOOKUP durchführen und dann Überschreiben.
- **INSERT:** Zunächst LOOKUP durchführen. Falls Satz mit v vorhanden: Fehler. Sonst: Freien Platz im Block überschreiben und ggf. neuen Block anfordern.
- **DELETE:** Zunächst LOOKUP durchführen. Bei Record Löschbit setzen.

Der Aufwand aller Operationen hängt davon ab, wie gleichmäßig die Hash-Funktion ihre Funktionswerte auf die Buckets verteilt und wie viele Blöcke im Mittel ein Bucket enthält. Im günstigsten Fall ist nur ein Directory-Zugriff und ein Datenblock-Zugriff erforderlich und ggf. ein Blockzugriff beim Zurückschreiben. Im ungünstigsten Fall sind alle Records in dasselbe Bucket gehasht worden und daher müssen ggf. alle Blöcke durchlaufen werden.

- **Beispiel** für offenes Hashing (übernommen aus *Ullman, Kapitel 2*):

  Abbildungen 29 und 30 zeigen die Verwaltung von Dinosaurier-Records. Verwendet wird eine Hash-Funktion h, die einen Schlüssel v abbildet auf die Länge von $v \ mod \ 5$. Pro Block können zwei Records mit Angaben zum Dinosaurier gespeichert werden sowie im Header des Blocks zwei Bits zum Frei/Belegt-Status der Subblocks.
  Abbildung 29 zeigt die Ausgangssituation. Nun werde `Elasmosaurus` (Hashwert = 2) eingefügt. Hierzu muss ein neuer Block für Bucket 2 angehängt werden. Dann werde `Brontosaurus` umgetauft in

`Apatosaurus`. Da diese Änderung den Schlüssel berührt, muss das Record gelöscht und modifiziert neu eingetragen werden. Abbildung 30 zeigt das Ergebnis.

Bei geschickt gewählter Hash-Funktion werden sehr kurze Zugriffszeiten erreicht, sofern das Bucket-Directory der Zahl der benötigten Blöcke angepasst ist. Bei statischem Datenbestand lässt sich dies leicht erreichen. Problematisch wird es bei dynamisch wachsendem Datenbestand. Um immer größer werdende Buckets zu vermeiden, muss von Zeit zu Zeit eine völlige Neuorganisation der Hash-Tabelle durchgeführt werden.

4.4 ISAM

Offenes und auch erweiterbares Hashing sind nicht in der Lage, Datensätze in sortierter Reihenfolge auszugeben oder Bereichsabfragen zu bearbeiten. Für Anwendungen, bei denen dies erforderlich ist, kommen Index-Strukturen zum Einsatz (englisch: *index sequential access method* = *ISAM*) Wir setzen daher voraus, dass sich die Schlüssel der zu verwaltenden Records als Zeichenketten interpretieren lassen und damit eine lexikographische Ordnung auf der Menge der Schlüssel impliziert wird. Sind mehrere Felder am Schlüssel beteiligt, so wird zum Vergleich deren Konkatenation herangezogen.

Neben der Haupt-Datei (englisch: *main file*), die alle Datensätze in lexikographischer Reihenfolge enthält, gibt es nun eine Index-Datei (english: *index file*) mit Verweisen in die Hauptdatei. Die Einträge der Index-Datei sind Tupel, bestehend aus Schlüsseln und Blockadressen, sortiert nach Schlüsseln. Liegt $< v, a >$ in der Index-Datei, so sind alle Record-Schlüssel im Block, auf den a zeigt, größer oder gleich v. Zur Anschauung: Fassen wir ein Telefonbuch als Hauptdatei auf (eine Seite $\equiv$ ein Block), so bilden alle die Namen, die jeweils links oben auf den Seiten vermerkt sind, einen Index. Da im Index nur ein Teil der Schlüssel aus der Hauptdatei zu finden sind, spricht man von einer dünn besetzten Index-Datei (englisch: *sparse index*).

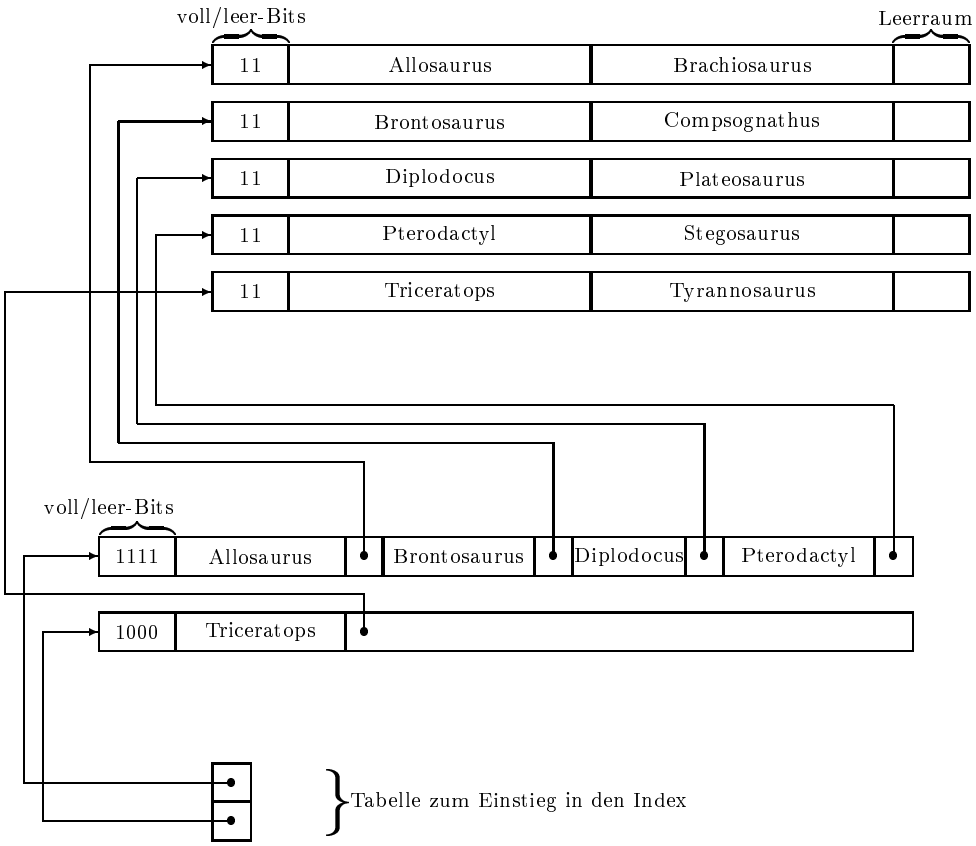


Abb. 31: ISAM: Ausgangslage

Wir nehmen an, die Records seien verschiebbar und pro Block sei im Header vermerkt, welche Subblocks belegt sind. Dann ergeben sich die folgenden Operationen:

- **LOOKUP:** Gesucht wird ein Record mit Schlüssel v_1. Suche (mit binary search) in der Index-Datei den letzten Block mit erstem Eintrag $v_2 \leq v_1$. Suche in diesem Block das letzte Paar (v_3, a) mit $v_3 \leq v_1$. Lies Block mit Adresse a und durchsuche ihn nach Schlüssel v_1.
- **MODIFY:** Führe zunächst LOOKUP durch. Ist der Schlüssel an der Änderung beteiligt, so wird MODIFY wie ein DELETE mit anschließendem INSERT behandelt. Andernfalls kann das Record überschrieben und dann der Block zurückgeschrieben werden.
- **INSERT:** Eingefügt wird ein Record mit Schlüssel v. Suche zunächst mit LOOKUP den Block B_i, auf dem v zu finden sein müsste (falls v kleinster Schlüssel, setze i =1). Falls B_i nicht vollständig gefüllt ist: Füge Record in B_i an passender Stelle ein, und verschiebe ggf. Records um eine Position nach rechts (Full/Empty-Bits korrigieren). Wenn v kleiner als alle bisherigen Schlüssel ist, so korrigiere Index-Datei. Wenn B_i gefüllt ist: Überprüfe, ob B_{i+1} Platz hat. Wenn ja: Schiebe überlaufendes Record nach B_{i+1} und korrigiere Index. Wenn nein: Fordere neuen Block B_i' an, speichere das Record dort, und füge im Index einen Verweis ein.
- **DELETE:** analog zu INSERT

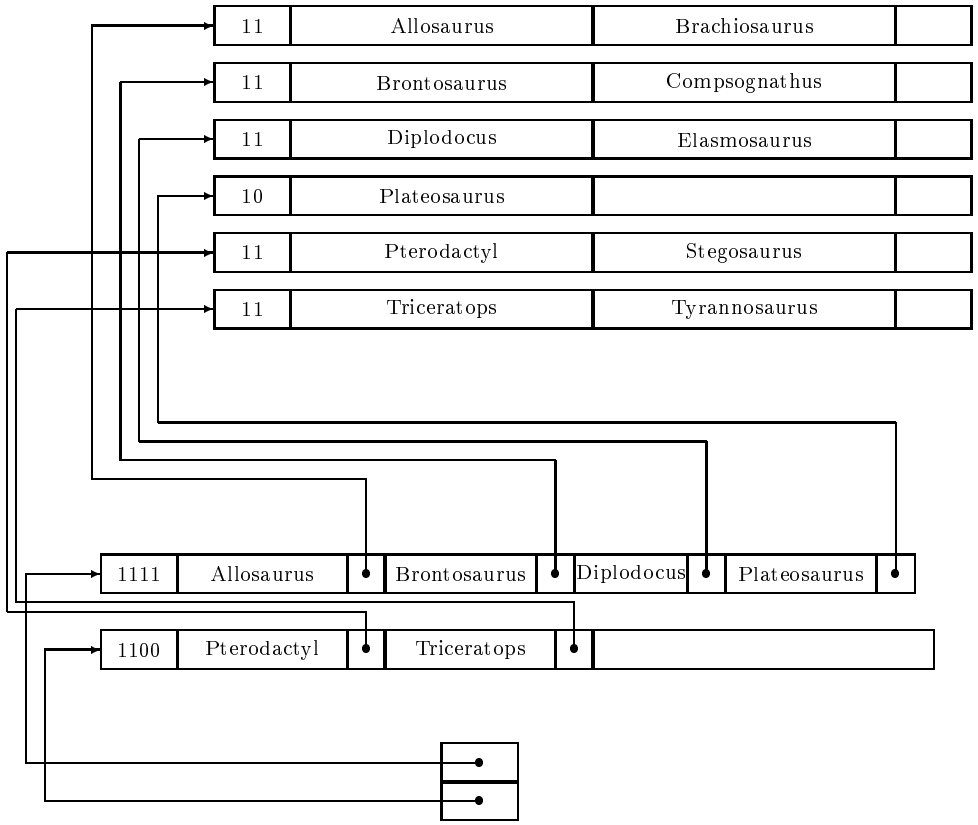


Abb. 32: ISAM: nach Einfügen von Elasmosaurus

Bemerkung: Ist die Verteilung der Schlüssel bekannt, so sinkt für n Index-Blöcke die Suchzeit durch *Interpolation Search* auf $log\ log\ n$ Schritte!

Abbildung 31 zeigt die Ausgangslage für eine Hauptdatei mit Blöcken, die jeweils 2 Records speichern können. Die Blöcke der Index-Datei enthalten jeweils vier Schlüssel-Adress-Paare. Weiterhin gibt es im Hauptspeicher eine Tabelle mit Verweisen zu den Index-Datei-Blöcken.

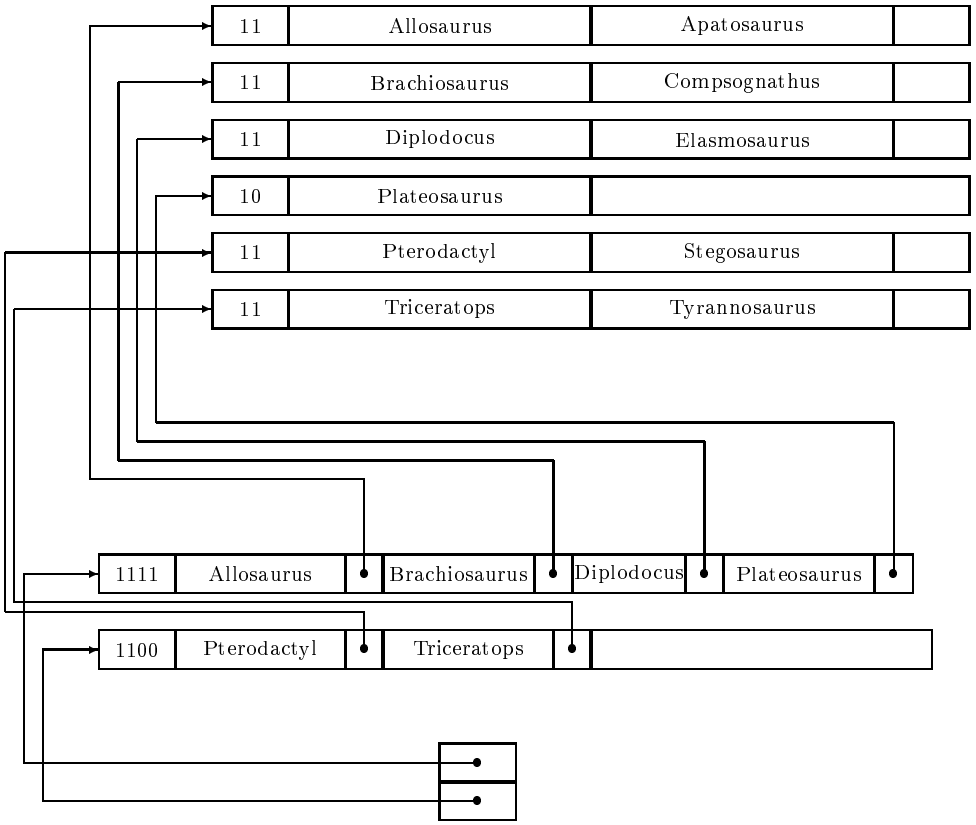


Abb. 33: ISAM: nach Umbenennen von Brontosaurus

Abbildung 32 zeigt die Situation nach dem Einfügen von `Elasmosaurus`. Hierzu findet man zunächst als Einstieg `Diplodocus`. Der zugehörige Dateiblock ist voll, so dass nach Einfügen von `Elasmosaurus` für das überschüssige Record `Plateosaurus` ein neuer Block angelegt und sein erster Schlüssel in die Index-Datei eingetragen wird.

Nun wird `Brontosaurus` umbenannt in `Apatosaurus`. Hierzu wird zunächst `Brontosaurus` gelöscht, sein Dateinachfolger `Compsognathus` um einen Platz vorgezogen und der Schlüssel in der Index-Datei, der zu diesem Blockzeiger gehört, modifiziert. Das Einfügen von `Apatosaurus` bewirkt einen Überlauf von `Brachiosaurus` in den Nachfolgeblock, in dem `Compsognathus` nun wieder an seinen alten Platz rutscht. Im zugehörigen Index-Block verschwindet daher sein Schlüssel wieder und wird überschrieben mit `Brachiosaurus` (Abbildung 33).

4.5 B*-Baum

Betrachten wir das Index-File als Daten-File, so können wir dazu ebenfalls einen weiteren Index konstruieren und für dieses File wiederum einen Index usw. Diese Idee führt zum B*-Baum.

Ein B*-Baum mit Parameter k wird charakterisiert durch folgende Eigenschaften:

- Jeder Weg von der Wurzel zu einem Blatt hat dieselbe Länge.
- Jeder Knoten außer der Wurzel und den Blättern hat mindestens k Nachfolger.
- Jeder Knoten hat höchstens $2 \cdot k$ Nachfolger.
- Die Wurzel hat keinen oder mindestens 2 Nachfolger.

Der Baum T befindet sich im Hintergrundspeicher, und zwar nimmt jeder Knoten einen Block ein. Ein Knoten mit j Nachfolgern speichert j Paare von Schlüsseln und Adressen $(s_1,\ a_1), \ldots, (s_j,\ a_j)$. Es gilt $s_1 \ \leq s_2 \ \leq \ \ldots \ \leq s_j$. Eine Adresse in einem Blattknoten bezeichnet den Datenblock mit den restlichen Informationen zum zugehörigen Schlüssel, sonst bezeichnet sie den Block zu einem Baumknoten: Enthalte der Block für Knoten p die Einträge $(s_1, a_1), \ldots, (s_j, a_j)$. Dann ist der erste Schlüssel im i-ten Sohn von p gleich s_i, alle weiteren Schlüssel in diesem Sohn (sofern vorhanden) sind größer als s_i und kleiner als s_{i+1}.

Wir betrachten nur die Operationen auf den Knoten des Baumes und nicht auf den eigentlichen Datenblöcken. Gegeben sei der Schlüssel s.

- **LOOKUP:**
  Beginnend bei der Wurzel steigt man den Baum hinab in Richtung des Blattes, welches den Schlüssel s enthalten müsste. Hierzu wird bei einem Knoten mit Schlüsseln $s_1, s_2, \ldots, s_j$ als nächstes der i-te Sohn besucht, wenn gilt $s_i \leq s < s_{i+1}$.
- **MODIFY:**
  Wenn das Schlüsselfeld verändert wird, muss ein DELETE mit nachfolgendem INSERT erfolgen. Wenn das Schlüsselfeld nicht verändert wird, kann der Datensatz nach einem LOOKUP überschrieben werden.
- **INSERT:**
  Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthalten soll. Wenn B weniger als $2k$ Einträge hat, so wird s eingefügt, und es werden die Vorgängerknoten berichtigt, sofern s kleinster Schlüssel im Baum ist. Wenn $B\ 2 \cdot k$ Einträge hat, wird ein neues Blatt B' generiert, mit den größeren k Einträgen von B gefüllt und dann der Schlüssel s eingetragen. Der Vorgänger von B und B' wird um einen weiteren Schlüssel s' (kleinster Eintrag in B') erweitert. Falls dabei Überlauf eintritt, pflanzt sich dieser nach oben fort.
- **DELETE:**
  Nach LOOKUP sei Blatt B gefunden, welches den Schlüssel s enthält. Das Paar (s, a) wird entfernt und ggf. der Schlüsseleintrag der Vorgänger korrigiert. Falls B jetzt $k-1$ Einträge hat, wird der unmittelbare Bruder B' mit den meisten Einträgen bestimmt. Haben beide Brüder gleich viel Einträge, so wird der linke genommen. Hat B' mehr als k Einträge, so werden die Einträge von B und B' auf diese beiden Knoten gleichmäßig verteilt. Haben B und B' zusammen eine ungerade Anzahl, so erhält der linke einen Eintrag mehr. Hat B' genau k Einträge, so werden B und B' verschmolzen. Die Vorgängerknoten müssen korrigiert werden.

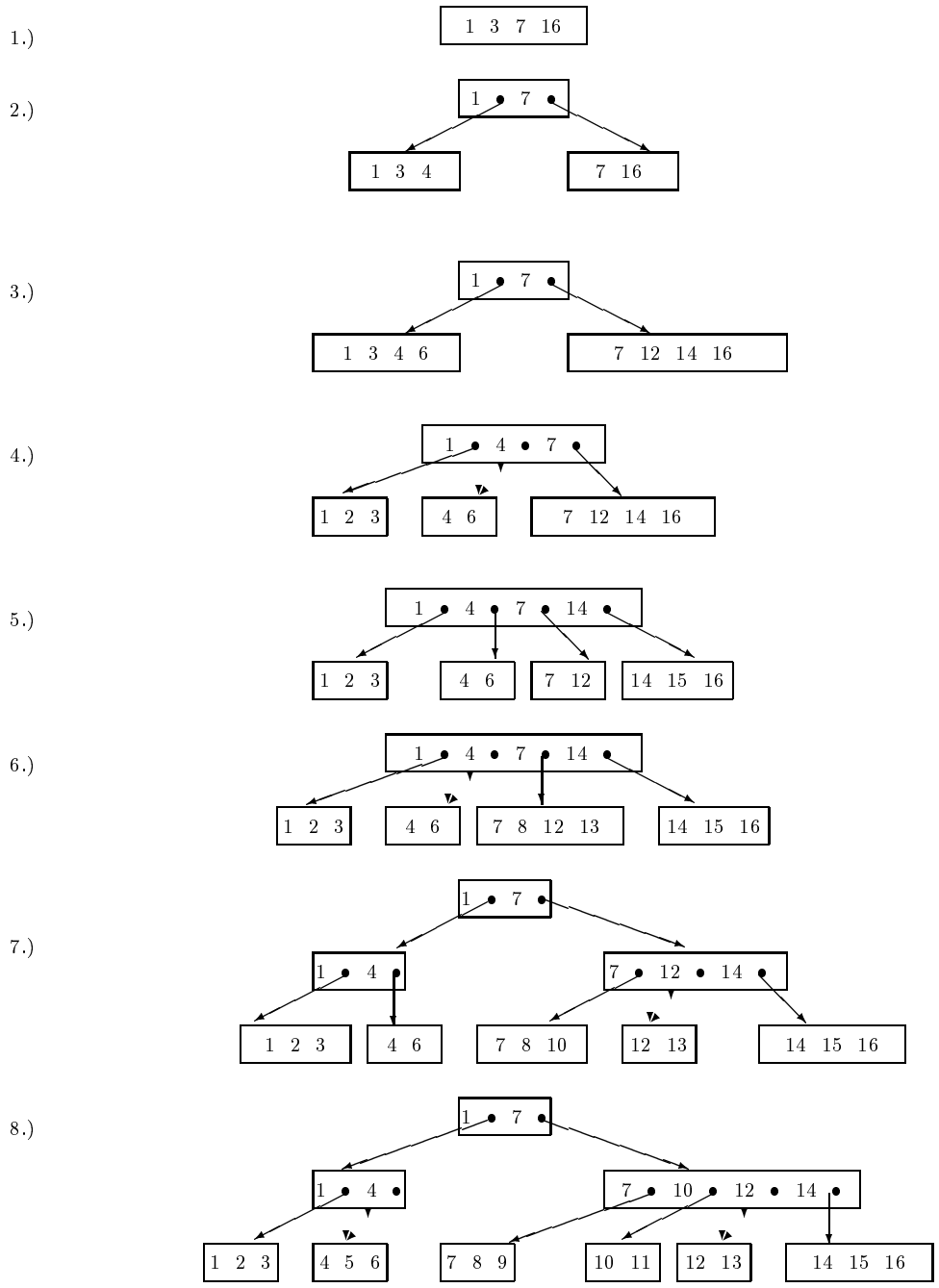


Abb. 34: dynamisches Verhalten eines B*-Baums

Abbildung 34 zeigt das dynamische Verhalten eines B*-Baums mit dem Parameter $k = 2$. Es werden 16 Schlüssel eingefügt und 8 Schnappschüsse gezeichnet:

Schlüssel	3	7	1	16	4	14	12	6	2	15	13	8	10	5	11	9
				↑	↑			↑	↑	↑		↑	↑			↑
Zeitpunkt				1	2			3	4	5		6	7			8

Der Parameter k ergibt sich aus dem Platzbedarf für die Schlüssel-Adress-Paare und der Blockgröße. Die Höhe des Baumes ergibt sich aus der benötigten Anzahl von Verzweigungen, um in den Blättern genügend Zeiger auf die Datenblöcke zu haben.

- **Beispiel** für die Berechnung des Platzbedarfs eines B*-Baums:

Gegeben seien 300.000 Datenrecords à 100 Bytes. Jeder Block umfasse 1.024 Bytes. Ein Schlüssel sei 15 Bytes lang, eine Adresse bestehe aus 4 Bytes. Daraus errechnet sich der Parameter k wie folgt

$$\lfloor \frac{1024}{15+4} \rfloor = 53 \Rightarrow k = 26$$

Die Wurzel sei im Mittel zu 50 % gefüllt (hat also 26 Söhne), ein innerer Knoten sei im Mittel zu 75 % gefüllt (hat also 39 Söhne), ein Datenblock sei im Mittel zu 75 % gefüllt (enthält also 7 bis 8 Datenrecords). 300.000 Records sind also auf $\lfloor \frac{300.000}{7,5} \rfloor = 40.000$ Datenblöcke verteilt.

Die Zahl der Zeiger entwickelt sich daher auf den oberen Ebenen des Baums wie folgt:

Ebene	Anzahl Knoten	Anzahl Zeiger
0	1	26
1	26	26 $*$ 39 = 1.014
2	26 $*$ 39	26 $*$ 39 $*$ 39 = 39.546

Damit reichen drei Ebenen aus, um genügend Zeiger auf die Datenblöcke bereitzustellen. Der Platzbedarf beträgt daher $1 + 26 + 26 \cdot 39 + 39546 \approx 40.000$ Blöcke.

Das LOOKUP auf ein Datenrecord verursacht also vier Blockzugriffe: es werden drei Indexblöcke auf Ebene 0, 1 und 2 sowie ein Datenblock referiert. Zum Vergleich: Das Heapfile benötigt 30.000 Blöcke.

Soll für offenes Hashing eine mittlere Zugriffszeit von 4 Blockzugriffen gelten, so müssen in jedem Bucket etwa 5 Blöcke sein (1 Zugriff für Hash-Directory, 3 Zugriffe im Mittel für eine Liste von 5 Blöcken). Von diesen 5 Blöcken sind 4 voll, der letzte halbvoll. Da 10 Records in einen Datenblock passen, befinden sich in einem Bucket etwa $4\frac{1}{2} \cdot 10 = 45$ Records. Also sind $\frac{300.000}{45} = 6.666$ Buckets erforderlich. Da 256 Adressen in einen Block passen, werden $\lfloor \frac{6666}{256} \rfloor = 26$ Directory-Blöcke benötigt. Der Platzbedarf beträgt daher $26 + 5 \cdot 6666 = 33356$.

Zur Bewertung von B*-Bäumen lässt sich sagen:

- **Vorteile:** dynamisch, schnell, Sortierung generierbar (ggf. Blätter verzeigern).
- **Nachteile:** komplizierte Operationen, Speicheroverhead.

4.6 Sekundär-Index

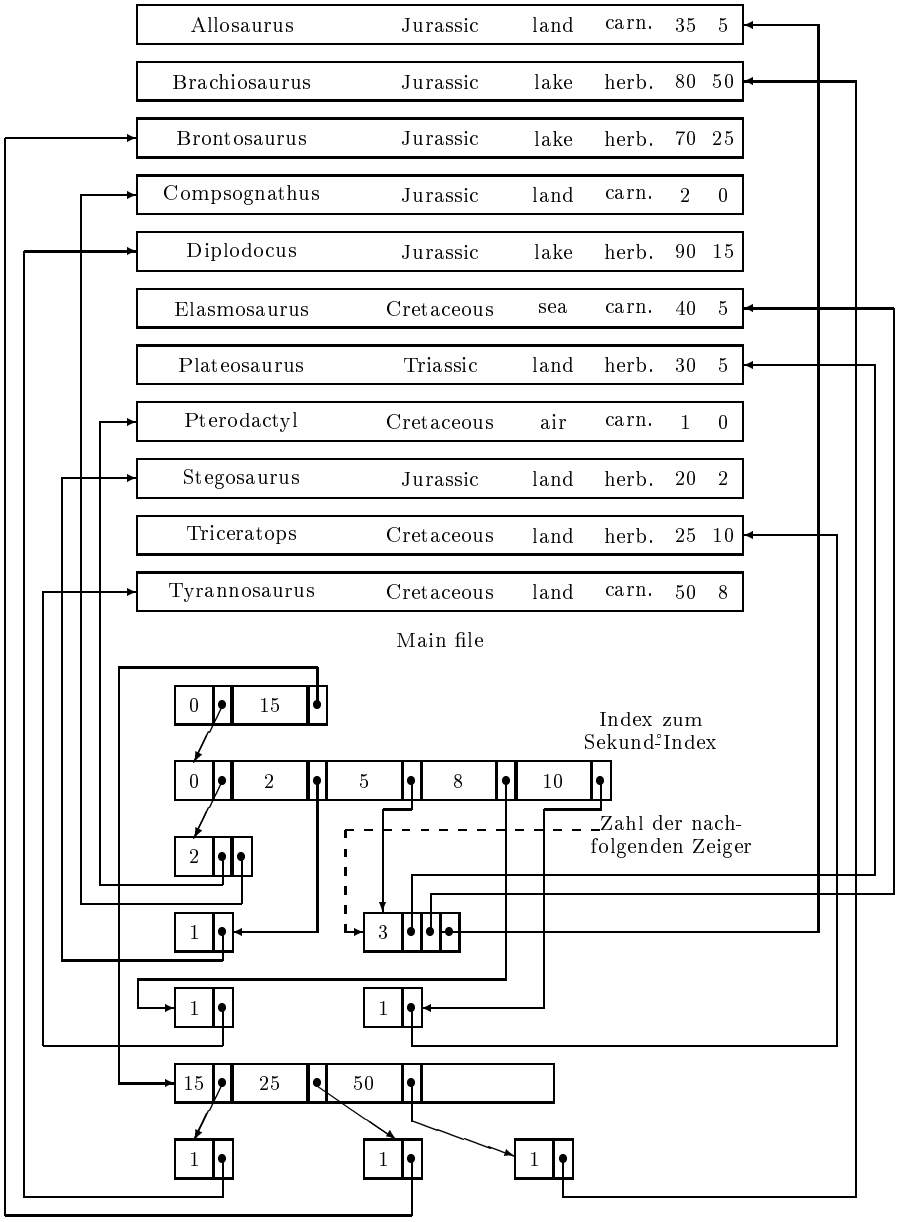


Abb. 35: Sekundär-Index für GEWICHT

Die bisher behandelten Organisationsformen sind geeignet zur Suche nach einem Record, dessen Schlüssel gegeben ist. Um auch effizient Nicht-Schlüssel-Felder zu behandeln, wird für jedes Attribut, das unterstützt werden soll, ein sogenannter Sekundär-Index (englisch: *secondary index*) angelegt. Er besteht aus einem Index-File mit Einträgen der Form <Attributwert, Adresse>.

Abbildung 35 zeigt für das Dinosaurier-File einen *secondary index* für das Attribut GEWICHT, welches, gespeichert in der letzten Record-Komponente, von 5 bis 50 variiert. Der Sekundär-Index (er wird erreicht über einen Index mit den Einträgen 0 und 15) besteht aus den Blöcken <0, 2, 5, 8, 10> und <15, 25, 50>. Die beim Gewicht g gespeicherte Adresse führt zunächst zu einem Vermerk zur Anzahl der Einträge mit dem Gewicht g und dann zu den Adressen der Records mit Gewicht g.

4.7 Google

Als Beispiel für einen heftig genutzten Index soll die grundsätzliche Vorgehensweise bei der Firma *Google* besprochen werden.

Google wurde 1998 von Sergey Brin und Larry Page gegründet und ist inzwischen auf 20.00 Mitarbeiter angewachsen. Die Oberfläche der Suchmaschine wird in mehr als 100 Sprachen angeboten. Etwa 1.000 Milliarden Webseiten liegen im Cache, der mehr als 1000 TeraByte an Plattenplatz belegt. Im Lexikon befinden sich etwa 14 Millionen Schlüsselwörter, nach denen täglich mit mehr als 250 Millionen Queries gefragt wird.

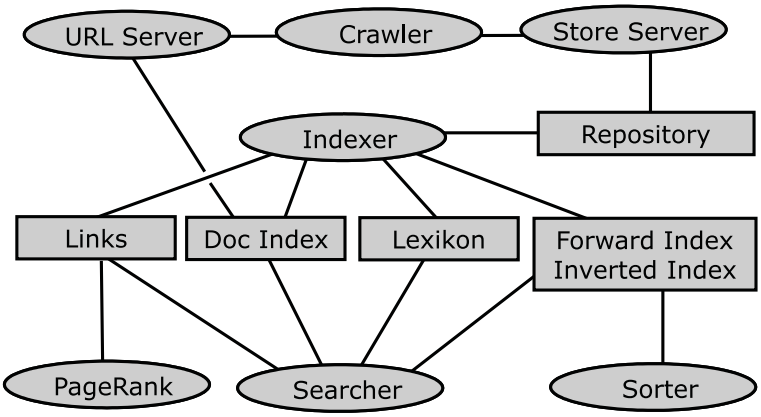


Abb. 36: Google Architektur

Abbildung 36 zeigt die prinzipielle Systemarchitektur: Der *URL-Server* generiert eine Liste von URLs, welche vom *Crawler* systematisch besucht werden. Dabei werden neu entdeckte URLs im *DocIndex* abgelegt. Der *Store Server* legt den kompletten, abgerufenen HTML-Text eines besuchten Dokuments in komprimierter Form im *Repository* ab. Der *Indexer* verarbeitet den Repository-Inhalt: Jedem Dokument wird eine eindeutige `docID` zugeteilt. Aus jedem HREF-Konstrukt, welches von Seite A zur Seite B zeigt, wird ein Paar (docID(A), docID(B)) erzeugt und in der Liste aller Verweis-Paare eingefügt (*Links*). Jedes beobachtete Schlüsselwort wird ins *Lexikon* eingetragen zusammen mit einer eindeutigen *wordID*, der Zahl der relevanten Dokumente und einem Verweis auf die erste Seite des *Inverted Index*, auf der solche `docID`s liegen, in dessen Dokumenten dieses Wort beobachtet wurde oder auf die es innerhalb eines Anchor-Textes verweist.

Die Zeilen im Lexikon haben daher die Struktur

```
wordID #docs pointer
wordID #docs pointer
wordID #docs pointer
wordID #docs pointer
```

Durch Inspektion der Webseiten im Repository erstellt der *Indexer* zunächst den *Forward Index*, welcher zu jedem Dokument und jedem Wort die Zahl der Hits und ihre Positionen notiert.

Die Struktur im Forward Index lautet

```
docID wordID #hits hit hit hit hit hit hit
      wordID #hits hit hit hit
docID wordID #hits hit hit hit hit
      wordID #hits hit hit hit hit hit
      wordID #hits hit hit hit hit
```

Daraus erzeugt der *Sorter* den *Inverted Index*, welcher folgende Struktur aufweist:

```
wordID docID #hits hit hit hit hit hit hit
```

```
       docID #hits hit hit hit
       docID #hits hit hit hit hit hit
wordID docID #hits hit hit hit hit
       docID #hits hit hit hit hit hit
```

Aus technischen Gründen ist der Forward Index bereits auf 64 *Barrels* verteilt, die jeweils für einen bestimmten WordID-Bereich zuständig sind. Hierdurch entsteht beim Einsortieren zwar ein Speicheroverhead, da die Treffer zu einer docID sich über mehrere Barrels verteilen, aber die anschliesende Sortierphase bezieht sich jeweils auf ein Barrel und kann daher parallelisiert werden.

Jeder *hit* umfasst zwei Bytes: 1 Capitalization Bit, 3 Bits für die Gröse des verwendeten Fonts und 12 Bit für die Adresse im Dokument. Dabei werden alle Positionen gröser als 4095 auf diesen Maximalwert gesetzt. Es wird unterschieden zwischen *plain hits*, welche innerhalb von normalem HTML auftauchen, und *fancy hits*, welche innerhalb einer URL, eines HTML-Title, im Anchor-Text oder in einem Meta-Tag auftauchen.

Der PageRank-Algorithmus berechnet auf Grundlage der Seitenpaare in der Datei *Links* die sogenannte *Link Popularität*, welche ein Mas für die Wichtigkeit einer Seite darstellt: Eine Seite ist wichtig, wenn andere wichtige Seiten auf sie verweisen.

Zur formalen Berechnung machen wir folgende Annahmen: Seite T habe $C(T)$ ausgehende Links. Auf Seite A mögen die Seiten $T_1, T_2, ..., T_n$ verweisen. Gegeben sei ein Dämpfungsfaktor $0 \leq d \leq 1$. (1-d) modelliert die Wahrscheinlichkeit, dass ein Surfer im Web eine Seite nicht über einen Link findet, sondern eine Verweiskette schlieslich abbricht und die Seite spontan anwählt.

Dann ergibt sich der *PageRank* von Seite A wie folgt:

$$PR(A) := (1-d) \ + \ d \cdot \sum_{i=1}^{n} \frac{PR(T_i)}{C(T_i)}$$

Je wichtiger Seite T_i ist (groses $PR(T_i)$) und je weniger ausgehende Links sie hat (kleines $C(T_i)$), desto mehr strahlt von ihrem Glanz etwas auf Seite A.

Hierdurch entsteht ein Gleichungssystem, welches sich durch ein Iterationsverfahren lösen lässt; Google braucht dafür ein paar Stunden.

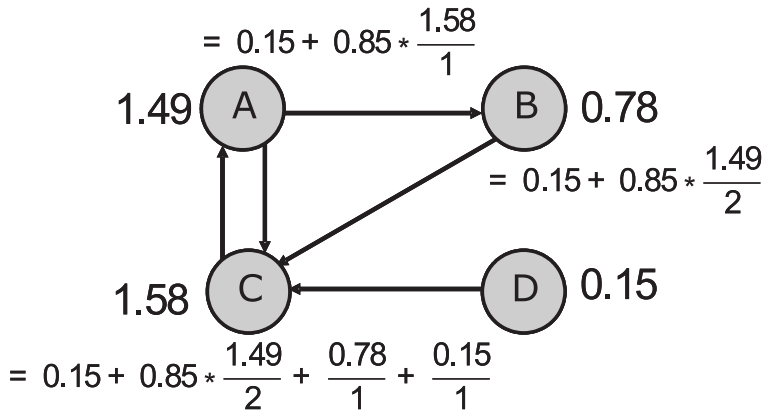


Abb. 37: PageRank errechnet mit Dämpungsfaktor d=0.85

Abbildung 37 zeigt ein Mini-Web mit vier untereinander verzeigerten Seiten A, B, C, D zusammen mit den ermittelten PageRanks, basierend auf einem Dämpfungsfaktor von $d = 0.85$.

Der vorberechnete PageRank und der vorberechnete Inverted Index erlauben nun eine effiziente Suche nach einem oder mehreren Schlüsselwörtern.

Bei einer Single-Word-Query w werden zunächst mit Hilfe des Inverted Index alle Seiten ermittelt, in denen w vorkommt. Für jede ermittelte Seite d werden die Hit-Listen ausgewertet bzgl. des Treffer-Typs (abnehmende

Wichtigkeit für Treffer in title, anchor, URL, plain text large font, plain text small font, ...). Der gewichtete Treffervektor wird skalar multipliziert mit dem Trefferhäufigkeitsvektor und ergibt den *Weight-Score(d,w)*. Dieser wird nach einem geheimen Google-Rezept mit dem *PageRank(d)* kombiniert und es entsteht der *Final-Score(d,w)*, welcher durch Sortieren die Trefferliste bestimmt.

Bei einer Multi-Word-Query $w_1, w_2, ..., w_n$ werden zunächst 10 Entfernungsklassen gebildet (von 'unmittelbar benachbart' bis 'sehr weit entfernt') und als Grundlage für einen sogenannten *Proximity-Score*$(d, w_1, w_2, ..., w_n)$ ausgewertet: nah in einem Dokument beeinanderliegende Suchwörter werden belohnt. Dann wird für jede gemeinsame Trefferseite d und jedes Suchwort w_i der konventionelle *Weight-Score*(d, w_i) kombiniert mit dem *Proximity-Rank*$(d, w_1, w_2, ..., w_n)$ und dem *PageRank(d)*. Der dadurch errechnete *Final-Score*$(d, w_1, w_2, ..., w_n)$ bestimmt nach dem Sortieren die Trefferliste.

5. Mehrdimensionale Suchstrukturen

5.1 Problemstellung

Ein Sekundär-Index ist in der Lage, alle Records mit $x_1 \leq a \leq x_2$ zu finden. Nun heißt die Aufgabe: Finde alle Records mit $x_1 \leq a_1 \leq x_2$ und $y_1 \leq a_2 \leq y_2, \ldots$

Beispiel für mehrdimensionale Bereichsabfrage: Gesucht sind alle Personen mit der Eigenschaft

Alter	zwischen 20 und 30 Jahre alt
Einkommen	zwischen 2000 und 3000 Euro
PLZ	zwischen 40000 und 50000

Im folgenden betrachten wir (wegen der einfacheren Veranschaulichung) nur den 2-dimensionalen Fall. Diese Technik ist auf beliebige Dimensionen verallgemeinerbar.

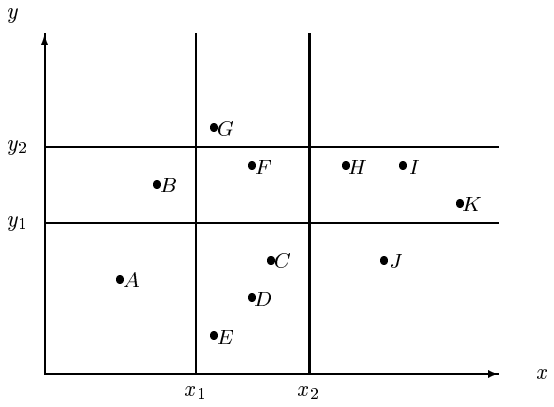


Abb. 38: Fläche mit Datenpunkten

Abbildung 38 zeigt eine zweidimensionale Fläche mit Datenpunkten sowie ein Query-Rechteck, gegeben durch vier Geraden.

Die Aufgabe besteht darin, alle Punkte zu ermitteln, die im Rechteck liegen. Hierzu bieten sich zwei naheliegende Möglichkeiten an:

- Projektion durchführen auf x oder y mit binärer Suche über vorhandenen Index, danach sequentiell durchsuchen, d.h. zunächst werden G, F, C, D, E ermittelt, danach bleibt F übrig
- Projektion durchführen auf x und Projektion durchführen auf y, anschließend Durchschnitt bilden.

Es ist offensichtlich, dass trotz kleiner Trefferzahl ggf. lange Laufzeiten auftreten können. Dagegen ist für die 1-dimensionale Suche bekannt: Der Aufwand beträgt O($k + \log n$) bei k Treffern in einem Suchbaum mit n Knoten.

5.2 k-d-Baum

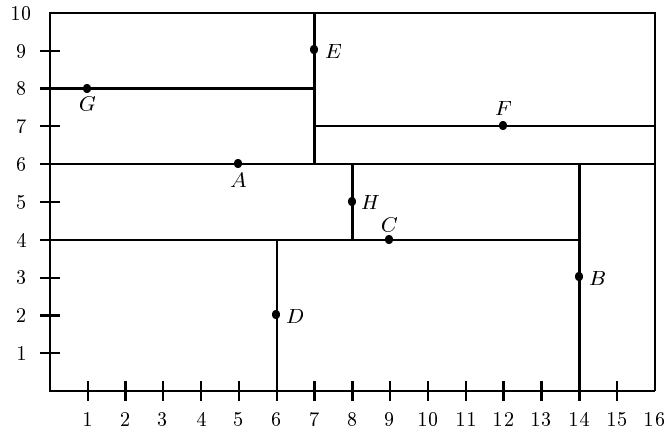


Abb. 39: Durch die Datenpunkte A,B,C,D,E,F,G,H partitionierte Fläche

Eine Verallgemeinerung eines binären Suchbaums mit einem Sortierschlüssel bildet der *k-d-Baum* mit k-dimensionalem Sortierschlüssel. Er verwaltet eine Menge von mehrdimensionalen Datenpunkten, wie z.B. Abbildung 39 für den 2-dimensionalen Fall zeigt. In der homogenen Variante enthält jeder Baumknoten ein komplettes Datenrecord und zwei Zeiger auf den linken und rechten Sohn (Abbildung 40). In der inhomogenen Variante enhält jeder Baumknoten nur einen Schlüssel und die Blätter verweisen auf die Datenrecords (Abbildung 41). In beiden Fällen werden die Werte der einzelnen Attribute abwechselnd auf jeder Ebene des Baumes zur Diskriminierung verwendet. Es handelt sich um eine statische Struktur; die Operationen Löschen und die Durchführung einer Balancierung sind sehr aufwendig.

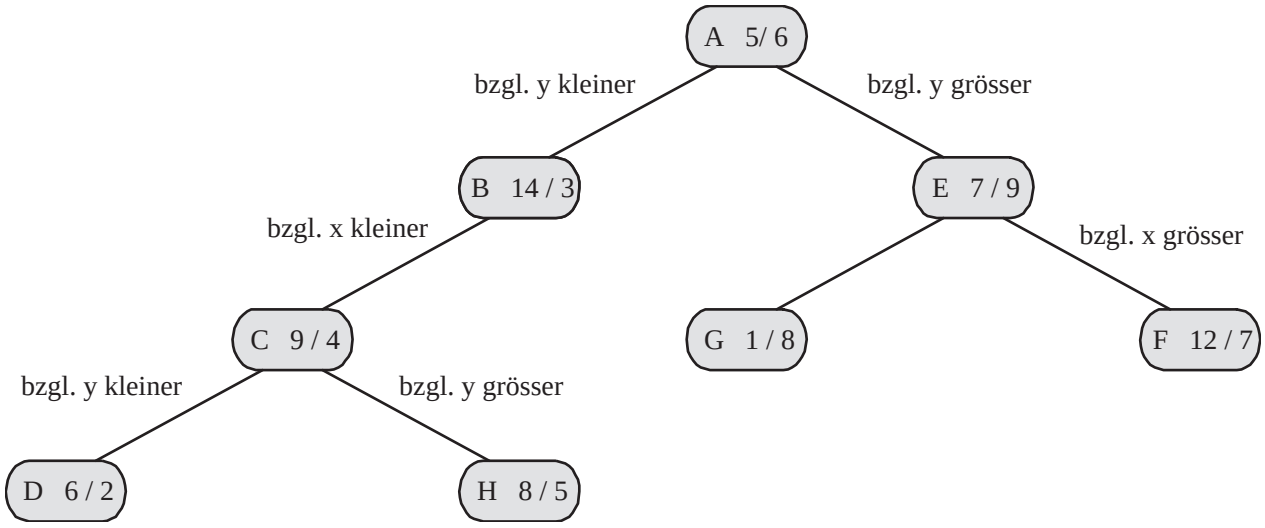


Abb. 40: 2-d-Baum (homogen) zu den Datenpunkten A,B,C,D,E,F,G,H

Im 2-dimensionalen Fall gilt für jeden Knoten mit Schlüssel $[x/y]$:

	im linken Sohn	im rechten Sohn
auf ungerader Ebene	alle Schlüssel $\leq x$	alle Schlüssel $> x$
auf gerader Ebene	alle Schlüssel $\leq y$	alle Schlüssel $> y$

Die Operationen auf einem $2-d$-Baum laufen analog zum binärem Baum ab:

- **Insert** (z.B. Record $[10/2]$):
  Suche mit Schlüssel $[x/y]$ unter Abwechslung der Dimension die Stelle, wo der $[x/y]$-Knoten sein müsste und hänge ihn dort ein.
- **Exakt Match** (z.B. finde Record $[15/5]$):
  Suche mit Schlüssel $[x/y]$ unter Abwechslung der Dimension bis zu der Stelle, wo der $[x/y]$-Knoten sein müsste.
- **Partial Match** (z.B. finde alle Records mit $x = 7$):
  An den Knoten, an denen nicht bzgl. x diskriminiert wird, steige in beide Söhne ab; an den Knoten, an denen bzgl. x diskriminiert wird, steige in den zutreffenden Teilbaum ab.
- **Range-Query** (z.B. finde alle Records $[x, y]$ mit $7 \leq x \leq 13, 5 \leq y \leq 8$):
  An den Knoten, an denen die Diskriminatorlinie das Suchrechteck schneidet, steige in beide Söhne ab, sonst steige in den zutreffenden Sohn ab. Beobachtung: Laufzeit $k + \log n$ Schritte bei k Treffern!
- **Best-Match** (z.B. finde nächstgelegenes Record zu $x = 7, y = 3$):
  Dies entspricht einer Range-Query, wobei statt eines Suchrechtecks jetzt ein Suchkreis mit Radius gemäß Distanzfunktion vorliegt. Während der Baumtraversierung schrumpft der Suchradius. Diese Strategie ist erweiterbar auf k-best-Matches.

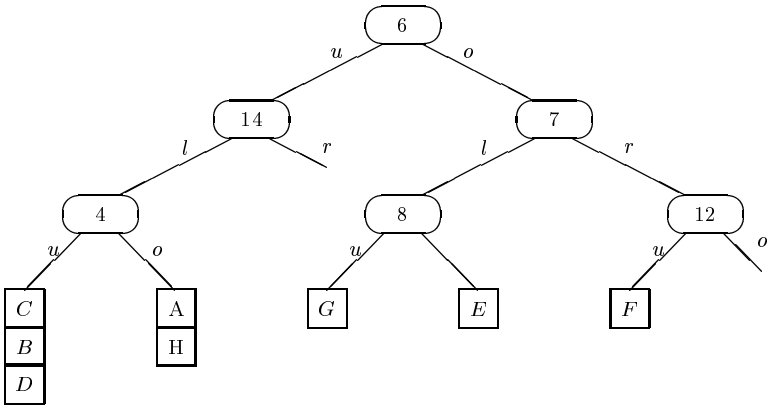


Abb. 41: 2-d-Baum (inhomogen) zu den Datenpunkten A,B,C,D,E,F,G,H

Bei der inhomogenen Variante enthalten die inneren Knoten je nach Ebene die Schlüsselinformation der zuständigen Dimension sowie Sohnzeiger auf weitere innere Knoten. Nur die Blätter verweisen auf Datenblöcke der Hauptdatei, die jeweils mehrere Datenrecords aufnehmen können. Auch die inneren Knoten werden zu Blöcken zusammengefasst, wie auf Abbildung 42 zu sehen ist. In Abbildung 41 befinden sich z.B. die Datenrecords *C, B* und *D* in einem Block.

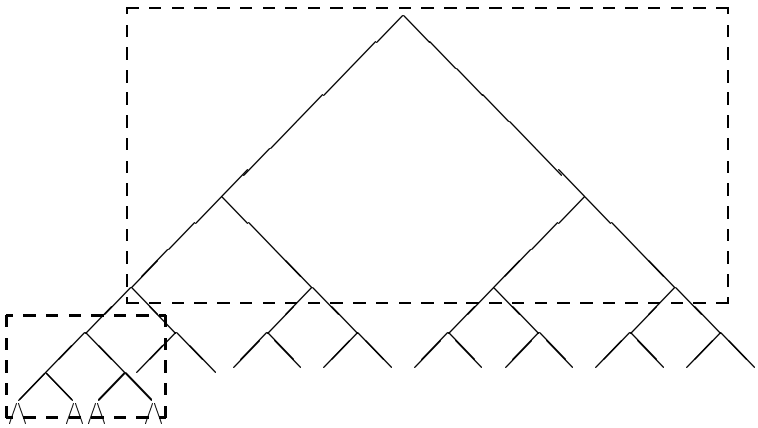


Abb. 42: Zusammenfassung von je 7 inneren Knoten auf einem Index-Block

Abbildung 43 zeigt, dass neben der oben beschriebenen 2-d-Baum-Strategie eine weitere Möglichkeit existiert, den Datenraum zu partitionieren. Dies führt zu den sogenannten *Gitterverfahren*.

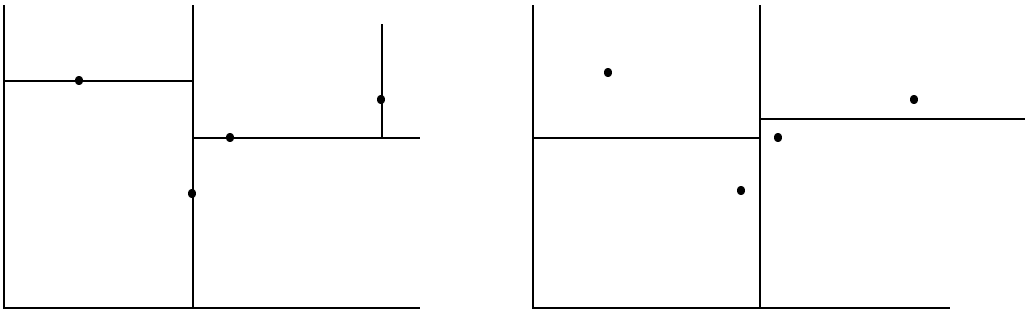


Abb. 43: Partitionierungsmöglichkeiten des Raumes

5.3 Gitterverfahren mit konstanter Gittergröse

Gitterverfahren, die mit konstanter Gittergröße arbeiten, teilen den Datenraum in Quadrate fester Größe auf. Abbildung 44 zeigt eine Anordnung von 24 Datenblöcken, die jeweils eine feste Anzahl von Datenrecords aufnehmen können. Über einen Index werden die Blöcke erreicht. Diese statische Partitionierung lastet die Datenblöcke natürlich nur bei einer Gleichverteilung wirtschaftlich aus und erlaubt bei Ballungsgebieten keinen effizienten Zugriff.

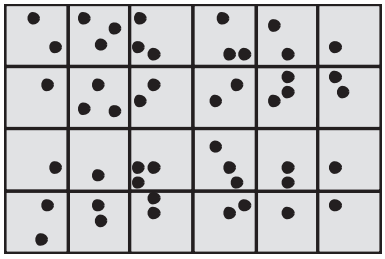


Abb. 44: Partitionierung mit fester Gittergröße

5.4 Grid File

Als Alternative zu den Verfahren mit fester Gittergröße stellten Hinrichs und Nievergelt im Jahre 1981 das *Grid File* vor, welches auch bei dynamisch sich änderndem Datenbestand eine *2-Platten-Zugriffsgarantie* gibt.

Erreicht wird dies (bei k-dimensionalen Tupeln) durch

- k Skalen zum Einstieg ins Grid-Directory (im Hauptspeicher)
- Grid-Directory zum Finden der Bucket-Nr. (im Hintergrundspeicher)
- Buckets für Datensätze (im Hintergrundspeicher)

Zur einfacheren Veranschaulichung beschreiben wir die Technik für Dimension $k = 2$. Verwendet werden dabei

- **zwei eindimensionale Skalen**, welche die momentane Unterteilung der X- bzw. Y-Achse enthalten:
```
var X: array [0..max_x] of attribut_wert_x;
var Y: array [0..max_y] of attribut_wert_y;
```
- **ein 2-dimensionales Grid-Directory**, welches Verweise auf die Datenblöcke enthält:
```
var G: array [0..max_x - 1, 0..max_y - 1] of pointer;
```

D.h. `G[i, j]` enthält eine Bucketadresse, in der ein rechteckiger Teilbereich der Datenpunkte abgespeichert ist. Zum Beispiel sind alle Punkte mit $30 < x \leq 40, 2050 < y \leq 2500$ im Bucket mit Adresse `G[1,2]` zu finden (in Abbildung 45 gestrichelt umrandet). Achtung: mehrere Gitterzellen können im selben Bucket liegen.

- **mehrere Buckets**, welche jeweils eine maximale Zahl von Datenrecords aufnehmen können.

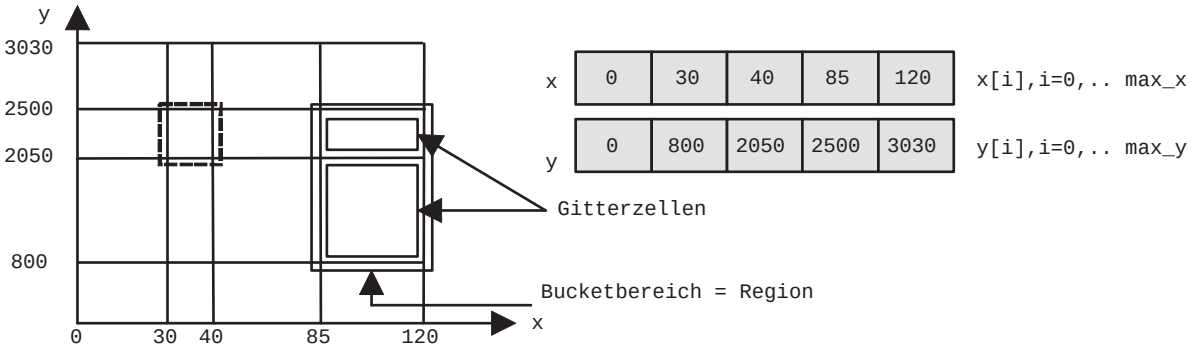


Abb. 45: Skalen und resultierende Gitterzellen

Beispiel für ein Lookup mit $x = 100, y = 1000$:
Suche in Skala x den letzten Eintrag $< x$. Er habe den Index $i = 3$.
Suche in Skala y den letzten Eintrag $< y$. Er habe den Index $j = 1$.
Lade den Teil des Grid-Directory in den Hauptspeicher, der $G[3, 1]$ enthält.
Lade Bucket mit Adresse $G[3, 1]$.

Beispiel für den Zugriff auf das Bucket-Directory (statisch, maximale Platznutzung):
Vorhanden seien 32.000 Datentupel, jeweils 5 passen in einen Block. Die X- und die Y-Achse habe jeweils 80 Unterteilungen. Daraus ergeben sich 6.400 Einträge für das Bucket-Directory G. Bei 4 Bytes pro Zeiger und 1024 Bytes pro Block passen 256 Zeiger auf einen Directory-Block. Also gibt es 25 Directory-Blöcke. D.h. $G[i, j]$ befindet sich auf Block $b = 5 * \lfloor j/16 \rfloor + \lfloor i/16 \rfloor$ an der Adresse $a = 16 * (j \ mod \ 16) + (i \ mod \ 16)$.

Bei einer *range query*, gegeben durch ein Suchrechteck, werden zunächst alle Gitterzellen bestimmt, die in Frage kommen, und dann die zugehörenden Buckets eingelesen.

5.5 Aufspalten und Mischen beim Grid File

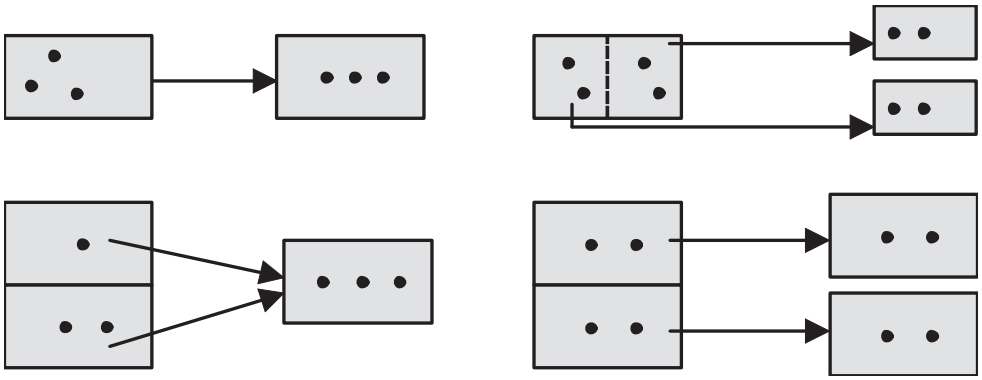


Abb. 46: Konsequenzen eines Bucket-Überlauf (mit und ohne Gitterverfeinerung)

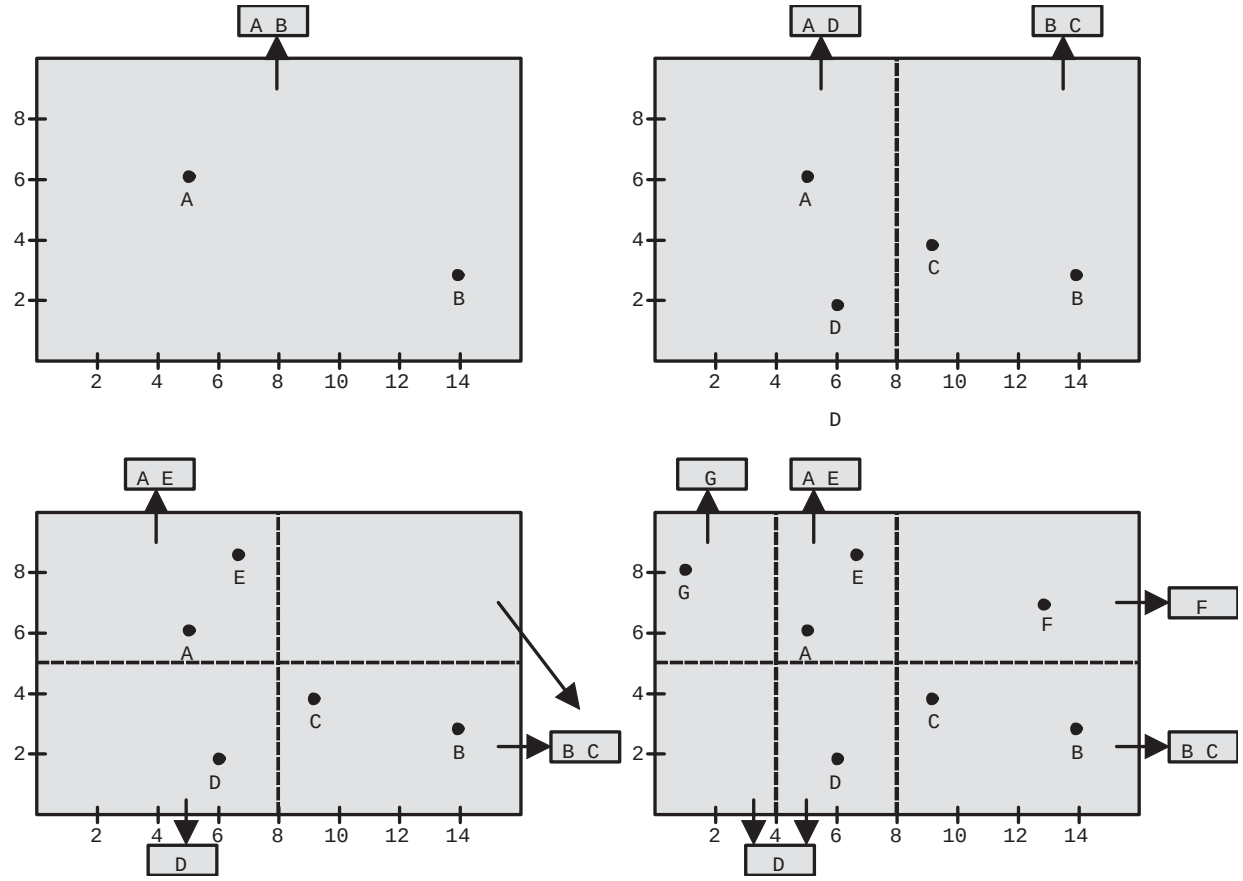


Abb. 47: Aufspalten der Regionen für Datenpunkte A, B, C, D, E, F, G

Die grundsätzliche Idee besteht darin, bei sich änderndem Datenbestand durch Modifikation der Skalen die Größen der Gitterzellen anzupassen.

Aufspalten von Regionen

Der Überlauf eines Buckets, dessen Region aus einer Zelle besteht, verursacht eine Gitterverfeinerung, die gemäß einer *Splitting Policy* organisiert wird. Im wesentlichen wird unter Abwechslung der Dimension die Region halbiert. Dieser Sachverhalt wird in der oberen Hälfte von Abbildung 46 demonstriert unter der Annahme, dass drei Datenrecords in ein Datenbucket passen. In der unteren Hälfte von Abbildung 46 ist zu sehen, dass bei Überlauf eines Buckets, dessen Region aus mehreren Gitterzellen besteht, keine Gitterverfeinerung erforderlich ist.

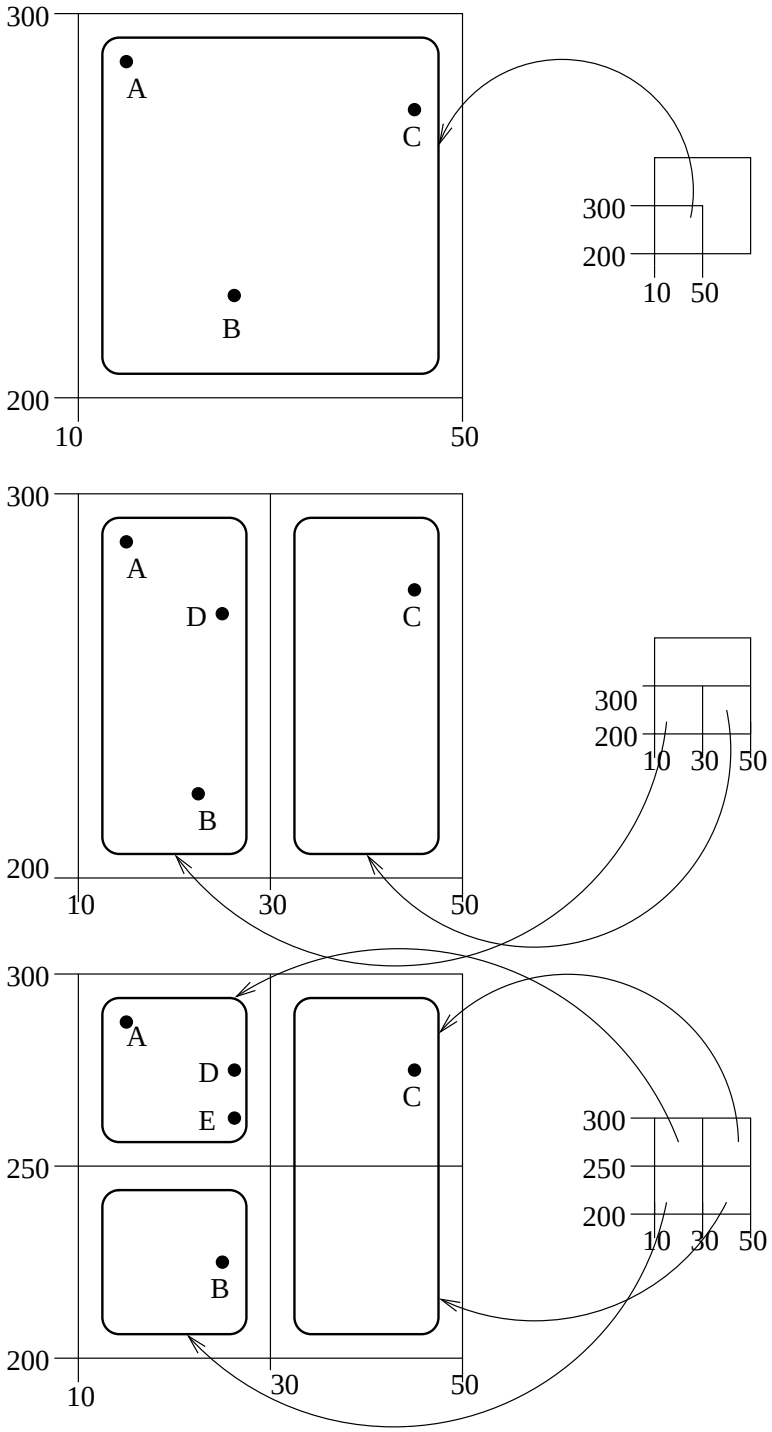


Abb. 48: Dynamik des Grid File beim Einfügen der Datenpunkte A,B,C,D,E

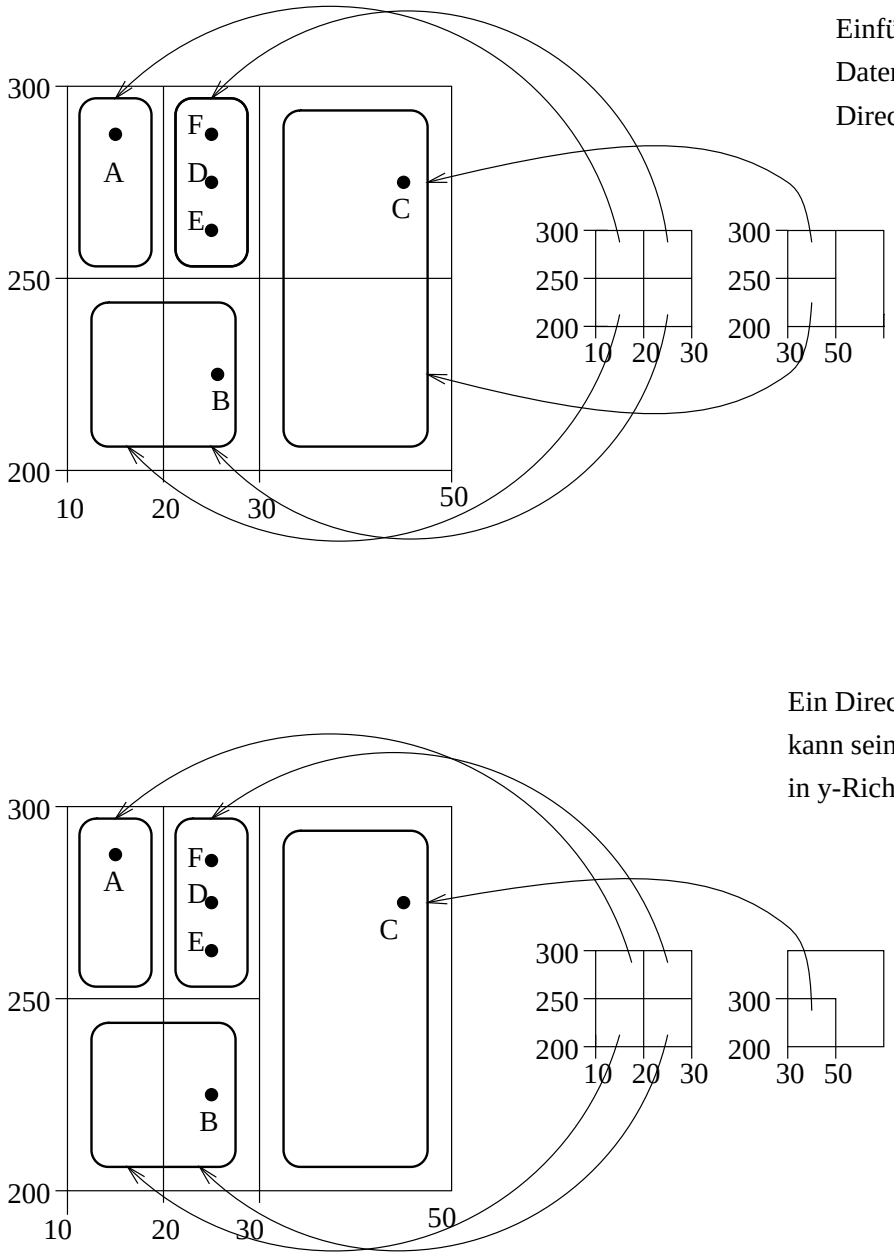


Abb. 49: Vergröberung des Grid Directory nach Aufspalten

Abbildung 47 zeigt die durch das sukzessive Einfügen in ein Grid File entwickelte Dynamik. Es handelt sich dabei um die in Kapitel 4 verwendeten Datenpunkte A, B, C, D, E, F, G. In dem Beispiel wird angenommen, dass 2 Datenrecords in einen Datenblock passen. Bei überlaufendem Datenblock wird die Region halbiert, wobei die Dimension abwechselt. Schließlich hat das Grid-Directory 6 Zeiger auf insgesamt 5 Datenblöcke. Die x-Skala hat drei Einträge, die y-Skala hat zwei Einträge.

Zu der dynamischen Anpassung der Skalen und Datenblöcke kommt noch die Buchhaltung der Directory-Blöcke. Dies wird in der Abbildung 5.11 demonstriert anhand der (neu positionierten) Datenpunkte A, B, C, D, E. Von den Directory-Blöcken wird angenommen, dass sie vier Adressen speichern können, in einen Datenblock mögen drei Datenrecords passen. Grundsätzlich erfolgt der Einstieg in den zuständigen Directory-Block über das sogenannte *Root-Directory*, welches im Hauptspeicher mit vergröberten Skalen liegt. Die durch das Einfügen verursachte Aufspaltung eines Datenblocks und die dadurch ausgelösten Verfeinerungen der Skalen ziehen auch Erweiterungen im Directory-Block nach. Abbildung 49 zeigt, wie beim Überlauf eines Directory-Blockes dieser halbiert und auf zwei Blöcke verteilt wird. Dabei kommt es zu einer Vergröberung der Skala.

Mischen von Regionen

Die beim Expandieren erzeugte Neustrukturierung bedarf einer Umordnung, wenn der Datenbestand schrumpft, denn nach dem Entfernen von Datenrecords können Datenblöcke mit zu geringer Auslastung entstehen, welche dann zusammengefasst werden sollten. Die *Merging Policy* legt den Mischpartner und den Zeitpunkt des Mischens fest:

- Mischpartner zu einem Bucket X kann nur ein Bucket Y sein, wenn die Vereinigung der beiden Bucketregionen ein Rechteck bildet (Abbildung 50). Grund: Zur effizienten Bearbeitung von Range-Queries sind nur rechteckige Gitter sinnvoll!
- Das Mischen wird ausgelöst, wenn ein Bucket höchstens zu 30 % belegt ist und wenn das vereinigte Bucket höchstens zu 70 $\%$ belegt sein würde (um erneutes Splitten zu vermeiden)

Abb. 50: Zusammenfassung von Regionen

5.6 Verwaltung geometrischer Objekte

In der bisherigen Anwendung repräsentierten die Datenpunkte im k-dimensionale Raum k-stellige Attributkombinationen. Wir wollen jetzt mithilfe der Datenpunkte geometrische Objekte darstellen und einfache geometrische Anfragen realisieren.

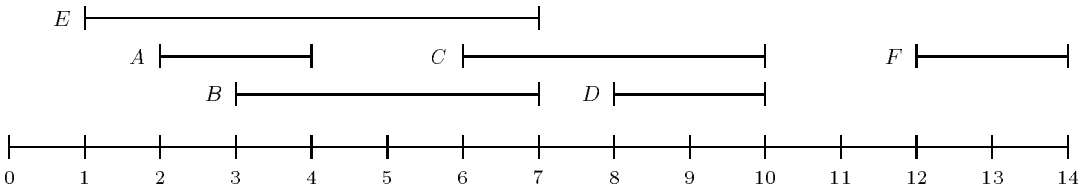


Abb. 51: Intervalle A,B,C,D,E,F über der Zahlengeraden

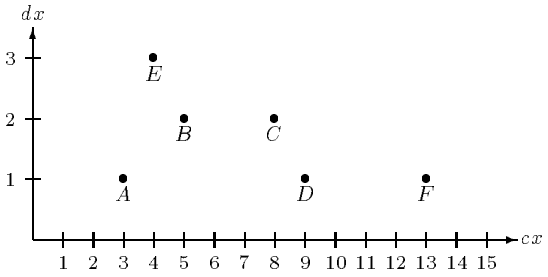


Abb. 52: Repräsentation von Intervallen durch Punkte

Abbildung 51 zeigt eine Ansammlung von Intervallen, die zu verwalten seien. Die Intervalle sollen durch Punkte im mehrdimensionalen Raum dargestellt werden. Wenn alle Intervalle durch ihre Anfangs- und Endpunkte repräsentiert würden, kämen sie auf der Datenfläche nur oberhalb der 45-Grad-Geraden zu liegen.

Abbildung 52 präsentiert eine wirtschaftlichere Verteilung, indem jede Gerade durch ihren Mittelpunkt und ihre halbe Länge repräsentiert wird.

Typische Queries an die Intervall-Sammlung lauten:

- Gegeben Punkt P, finde alle Intervalle, die ihn enthalten.
- Gegeben Intervall I, finde alle Intervalle, die es schneidet.

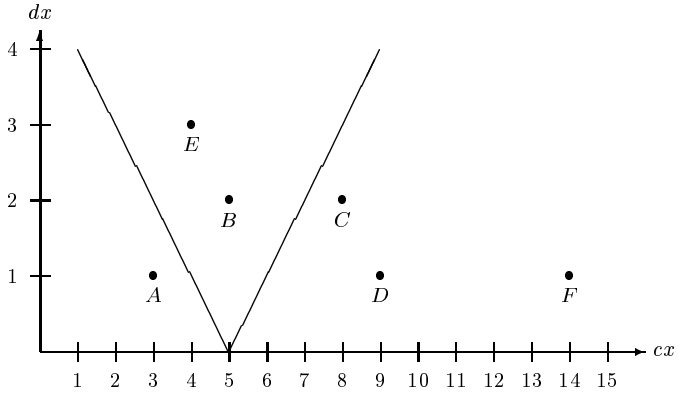


Abb. 53: Abfragekegel zu Punkt p=5

Abbildung 53 zeigt den kegelförmigen Abfragebereich zum Query-Punkt p=5, in dem alle Intervalle (repräsentiert durch Punkte) liegen, die den Punkt p enthalten. Grundlage ist die Überlegung, dass ein Punkt P genau dann im Intervall mit Mitte m und halber Länge d enhalten ist, wenn gilt: $m - d \leq p \leq m + d$

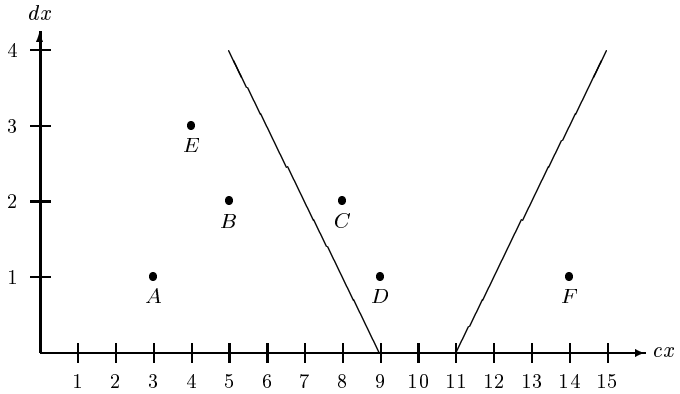


Abb. 54: Abfragekegel zu Intervall mit Mitte s=10 und halber Länge t=1

Abbildung 54 zeigt den kegelförmigen Abfragebereich zu dem Query-Intervall mit Mittelpunkt s=10 und halber Länge t=1, in dem alle Intervalle (repäsentiert durch Punkte) liegen, die das Query-Intervall schneiden. Grundlage ist die Überlegung, dass ein Intervall mit Mitte s und halber Länge t genau dann ein Intervall mit Mitte m und halber Länge d schneidet, wenn gilt: $m - d \leq s + t \text{ und } s - t \leq m + d$

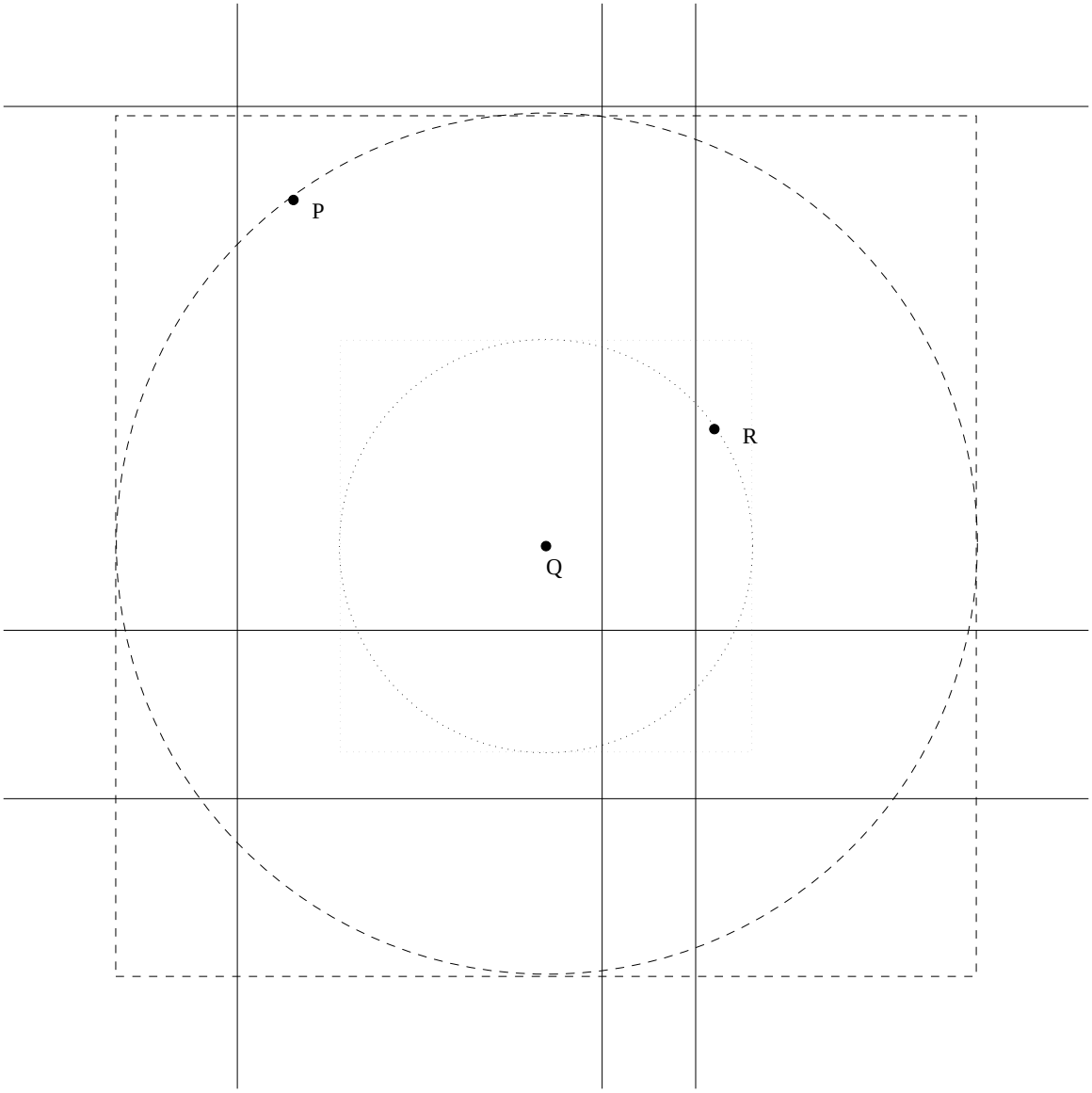


Abb. 55: Nearest-Neighbor-Suche zu Query-Punkt Q

Abbildung 55 zeigt die Vorgehensweise bei der Bestimmung des nächstgelegenen Nachbarn (englisch: *nearest neighbor*). Suche zunächst auf dem Datenblock, der für den Query-Point Q zuständig ist, den nächstgelegenen Punkt P. Bilde eine *Range-Query* mit Quadrat um den Kreis um Q mit Radius $|P - Q|$. Schränke Quadratgröße weiter ein, falls nähere Punkte gefunden werden.

Die erwähnten Techniken lassen sich auf höherdimensionierte Geometrie-Objekte wie Rechtecke oder Quader erweitern. Zum Beispiel bietet sich zur Verwaltung von orthogonalen Rechtecken in der Ebene folgende Möglichkeit an: Ein Rechteck wird repräsentiert als ein Punkt im 4-dimensionalen Raum, gebildet durch die beiden 2-dimensionalen Punkte für horizontale bzw. vertikale Lage. Zu einem Query-Rechteck, bestehend aus horizontalem Intervall P und vertikalem Intervall Q, lassen sich die schneidenden Rechtecke finden im Durchschnitt der beiden kegelförmigen Abfragebereiche zu den Intervallen P und Q.

6. Das Relationale Modell

6.1 Definition

Gegeben sind n nicht notwendigerweise unterschiedliche *Wertebereiche* (auch *Domänen* genannt) $D_1, \ldots, D_n$, welche nur *atomare* Werte enthalten, die nicht strukturiert sind, z.B. Zahlen oder Strings. Eine Relation R ist definiert als Teilmenge des kartesischen Produkts der n Domänen:

$$R \subseteq D_1 \times D_2 \times \ldots \times D_n$$

Es wird unterschieden zwischen dem *Schema* einer Relation, gegeben durch die n Domänen und der aktuellen *Ausprägung* (Instanz). Ein Element der Menge R wird als Tupel bezeichnet, dessen $Stelligkeit$ sich aus dem Relationenschema ergibt. Wir bezeichnen mit **sch**(*R*) oder mit $\mathcal{R} = A_1, \ldots, A_n$ die Menge der Attribute und mit R die aktuelle Ausprägung. Mit **dom**(*A*) bezeichnen wird die Domäne eines Attributs A. Also gilt

$$R \subseteq \mathrm{dom}(A_1) \times \mathrm{dom}(A_2) \times \ldots \times \mathrm{dom}(A_n)$$

Im Datenbankbereich müssen die Domänen außer einem Typ noch einen Namen haben. Wir werden Relationenschemata daher durch eine Folge von Bezeichner/Wertebereich - Tupeln spezifizieren, z.B.

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

Hierbei wird in den eckigen Klammern [. . .] angegeben, wie die Tupel aufgebaut sind, d.h. welche Attribute vorhanden sind und welchen Wertebereich sie haben. Ein Schlüsselkandidat wird unterstrichen. Die geschweiften Klammern { . . . } sollen ausdrücken, dass es sich bei einer Relationenausprägung um eine Menge von Tupeln handelt. Zur Vereinfachung wird der Wertebereich auch manchmal weggelassen:

Telefonbuch : { [Name, Adresse, TelefonNr] }

6.2 Umsetzung in ein relationales Schema

Das ER-Modell besitzt zwei grundlegende Strukturierungskonzepte:

- Entity-Typen
- Relationship-Typen

Abbildung 56 zeigt ein ER-Diagramm zum Universitätsbetrieb. Zunächst wollen wir die Generalisierung ignorieren, da es im relationalen Modell keine unmittelbare Umsetzung gibt. Dann verbleiben vier Entity-Typen, die auf folgende Schemata abgebildet werden:

Studenten :{[MatrNr : integer, Name : string, Semester : integer] }
Vorlesungen :{[VorlNr : integer, Titel : string, SWS : integer] }
Professoren :{[PersNr : integer, Name : string, Rang : string, Raum : integer] }
Assistenten :{[PersNr : integer, Name : string, Fachgebiet : string] }

Bei der relationalen Darstellung von Beziehungen richten wir im *Initial*-Entwurf für jeden Beziehungstyp eine eigene Relation ein. Später kann davon ein Teil wieder eliminiert werden. Grundsätzlich entsteht das Relationenschema durch die Folge aller Schlüssel, die an der Beziehung beteiligt sind sowie ggf. weitere Attribute der Beziehung. Dabei kann es notwendig sein, einige der Attribute umzubenennen. Die Schlüsselattribute für die referierten Entity-Typen nennt man *Fremdschlüssel*.

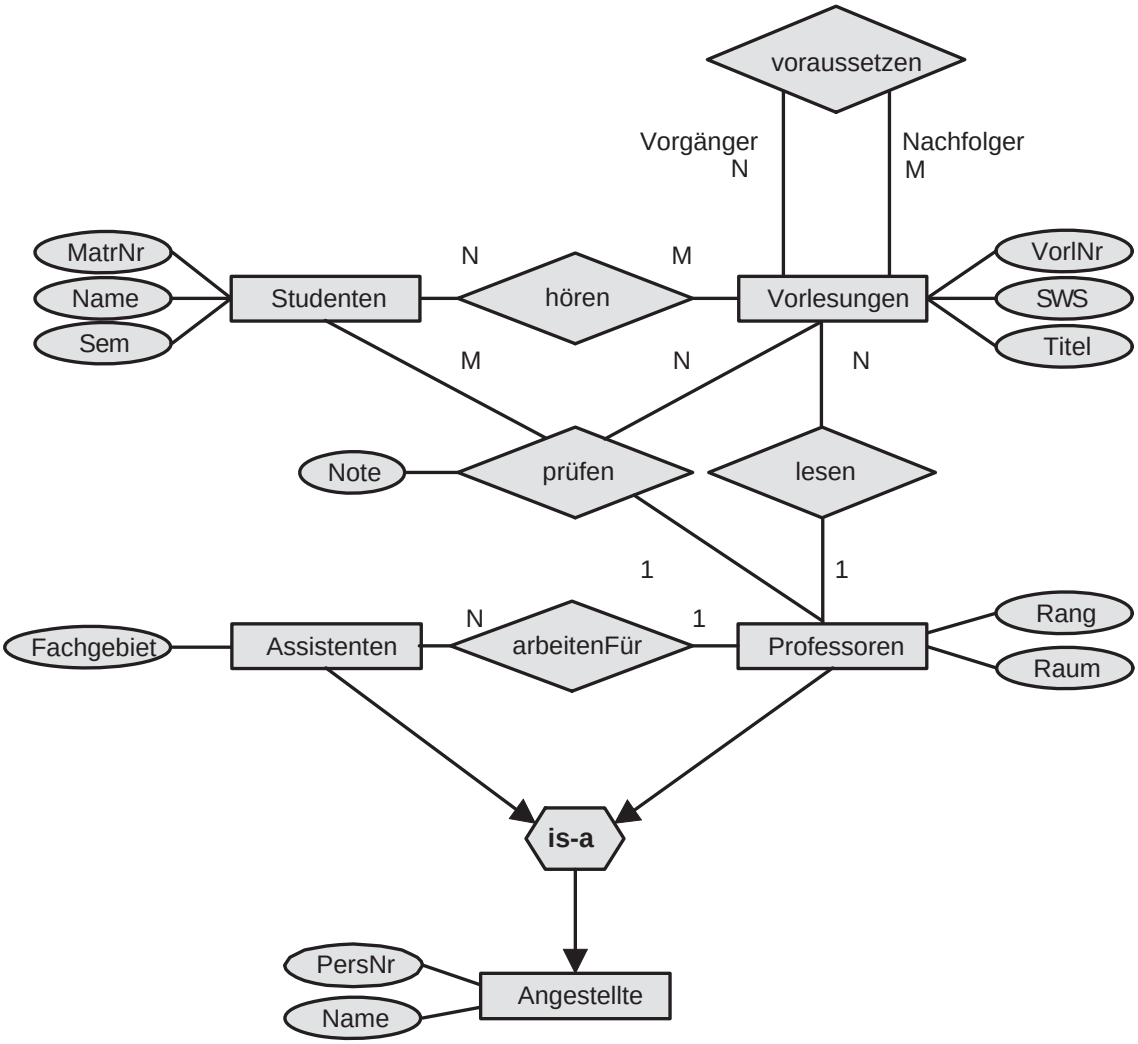


Abb. 56: Konzeptuelles Schema der Universität

Für das Universitätsschema entstehen aus den Relationships die folgenden Schemata:

hören :{[MatrNr : integer, VorlNr : integer] }
lesen :{[PersNr : integer, VorlNr : integer] }
arbeitenFür :{[AssiPersNr : integer, ProfPersNr : integer] }
voraussetzen:{[Vorgänger : integer, Nachfolger : integer] }
prüfen :{[MatrNr : integer, VorlNr : integer, PersNr : integer, Note : decimal] }

Unterstrichen sind jeweils die Schlüssel der Relation, eine *minimale* Menge von Attributen, deren Werte die Tupel eindeutig identifizieren.

Da die Relation *hören* eine $N : M$-Beziehung darstellt, sind sowohl die Vorlesungsnummern als auch die Matrikelnummern alleine keine Schlüssel, wohl aber ihre Kombination.

Bei der Relation *lesen* liegt eine 1:N-Beziehung vor, da jeder Vorlesung genau ein Dozent zugeordnet ist mit der partiellen Funktion

$$lesen : Vorlesungen \rightarrow Professoren$$

Also ist für die Relation *lesen* bereits das Attribut *VorlNr* ein Schlüsselkandidat, für die Relation *arbeitenFür* bildet die *AssiPersNr* einen Schlüssel.

Bei der Relation *prüfen* liegt wiederum eine partielle Abbildung vor:

$$prüfen : Studenten \times Vorlesungen \rightarrow Professoren$$

Sie verlangt, das *MatrNr* und *VorlNr* zusammen den Schlüssel bilden.

6.3 Verfeinerung des relationalen Schemas

Das im Initialentwurf erzeugte relationale Schema läst sich verfeinern, indem einige der $1:1$-, $1:N$- oder $N:1$-Beziehungen eliminiert werden. Dabei dürfen nur Relationen mit gleichem Schlüssel zusammengefast werden.

Nach dieser Regel können von den drei Relationen

Vorlesungen:{[VorlNr : integer, Titel : string, SWS : integer] }
Professoren :{[PersNr : integer, Name : string, Rang : string, Raum : integer] }
lesen :{[PersNr : integer, VorlNr : integer] }

die Relationen *Vorlesungen* und *lesen* zusammengefast werden. Somit verbleiben im Schema

Vorlesungen:{[VorlNr : integer, Titel : string, SWS : integer, gelesenVon : integer] }
Professoren :{[PersNr : integer, Name : string, Rang : string, Raum : integer] }

Das Zusammenlegen von Relationen mit unterschiedlichen Schlüsseln erzeugt eine Redundanz von Teilen der gespeicherten Information. Beispielsweise speichert die (unsinnige) Relation

Professoren':{[PersNr, liestVorl, Name, Rang, Raum] }

zu jeder von einem Professor gehaltenen Vorlesung seinen Namen, seinen Rang und sein Dienstzimmer:

Bei $1:1$-Beziehungen gibt es zwei Möglichkeiten, die ursprünglich entworfene Relation mit den beteiligten Entity-Typen zusammenzufassen.

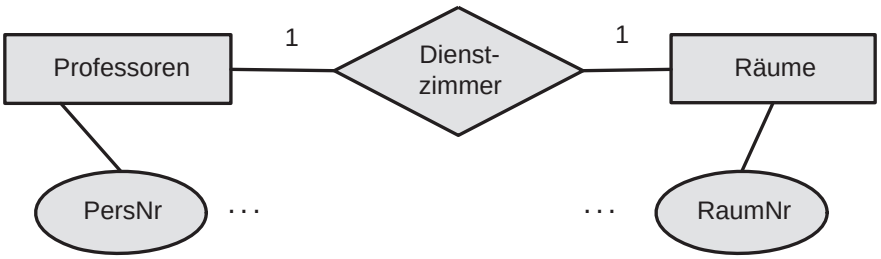


Abb. 57: Beispiel einer 1:1-Beziehung

Abbildung 57 zeigt eine mögliche Modellierung für die Unterbringung von Professoren in Räumen als $1:1$-Beziehung. Die hierzu gehörenden Relationen heisen

Professoren :{[PersNr, Name, Rang] }
Räume :{[RaumNr, Gröe, Lage] }
Dienstzimmer:{[PersNr, RaumNr] }

Da *Professoren* und *Dienstzimmer* denselben Schlüssel haben, kann zusammengefasst werden zu

Professoren:{[PersNr, Name, Rang, Raum] }
Räume :{[RaumNr, Gröse, Lage] }

Da das Attribut *RaumNr* innerhalb der Relation *Dienstzimmer* ebenfalls einen Schlüssel bildet, könnten als Alternative auch die Relationen *Dienstzimmer* und *Räume* zusammengefasst werden:

Professoren:{[PersNr, Name, Rang] }
Räume :{[RaumNr, Gröse, Lage, ProfPersNr] }

Diese Modellierung hat allerdings den Nachteil, das viele Tupel einen sogenannten Nullwert für das Attribut *ProfPersNr* haben, da nur wenige Räume als Dienstzimmer von Professoren genutzt werden.

Die in Abbildung 56 gezeigte Generalisierung von *Assistenten* und *Professoren* zu *Angestellte* könnte wie folgt durch drei Relationen dargestellt werden:

Angestellte :{[PersNr, Name] }
Professoren:{[PersNr, Rang, Raum] }
Assistenten :{[PersNr, Fachgebiet] }

Hierdurch wird allerdings die Information zu einem Professor, wie z.B.

[2125, Sokrates, C4, 226]

auf zwei Tupel aufgeteilt:

[2125, Sokrates] und [2125, C4, 226]

Um die vollständige Information zu erhalten, müssen diese beiden Relationen verbunden werden (Join). Folgende Tabellen zeigen eine Beispiel-Ausprägung der Universitäts-Datenbasis. Das zugrundeliegende Schema enthält folgende Relationen:

Studenten	:	{[MatrNr : integer, Name : string, Semester : integer] }
Vorlesungen	:	{[VorlNr : integer, Titel : string, SWS : integer, gelesenVon : integer] }
Professoren	:	{[PersNr : integer, Name : string, Rang : string, Raum : integer] }
Assistenten	:	{[PersNr : integer, Name : string, Fachgebiet : string, Boss : integer] }
hören	:	{[MatrNr : integer, VorlNr : integer] }
voraussetzen	:	{[Vorgänger : integer, Nachfolger : integer] }
prüfen	:	{[MatrNr : integer, VorlNr : integer, PersNr : integer, Note :decimal] }

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	52
2134	Augustinus	C3	309
2136	Curie	C4	36
2137	Kant	C4	7

Studenten		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12
26120	Fichte	10
26830	Aristoxenos	8
27550	Schopenhauer	6
28106	Carnap	3
29120	Theophrastos	2
29555	Feuerbach	2

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
5043	Erkenntnistheorie	3	2126
5049	Mäeutik	2	2125
4052	Logik	4	2125
5052	Wissenschaftstheorie	3	2126
5216	Bioethik	2	2126
5259	Der Wiener Kreis	2	2133
5022	Glaube und Wissen	2	2134
4630	Die 3 Kritiken	4	2137

voraussetzen	
Vorgänger	Nachfolger
5001	5041
5001	5043
5001	5049
5041	5216
5043	5052
5041	5052
5052	5259

hören	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
27550	5041
28106	4052
28106	5216
28106	5259
27550	4630
29120	5041
29120	5049
29555	5022
25403	5022
29555	5001

Assistenten			
PersNr	Name	Fachgebiet	Boss
3002	Platon	Ideenlehre	2125
3003	Aristoteles	Syllogistik	2125
3004	Wittgenstein	Sprachtheorie	2126
3005	Rhetikus	Planetenbewegung	2127
3006	Newton	Keplersche Gesetze	2127
3007	Spinoza	Gott und Natur	2134

prüfen			
MatrNr	VorlNr	PersNr	Note
28106	5001	2126	1
25403	5041	2125	2
27550	4630	2137	2

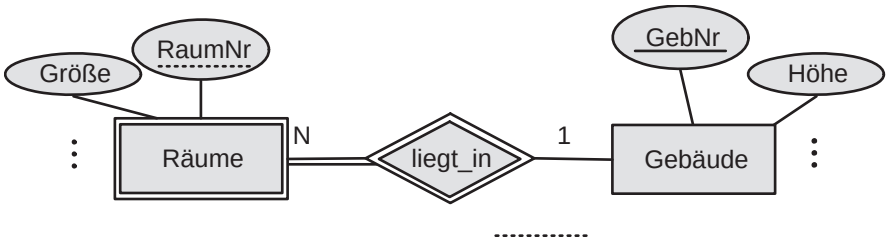


Abb. 58: Schwacher Entity-Typ

Zur Modellierung von schwachen Entity-Typen betrachten wir Abbildung 58, in der mittels der Relation *liegt_in* der schwache Entitity-Typ *Räume* dem Entity-Typ *Gebäude* untergeordnet wurde.

Wegen der $1:N$-Beziehung zwischen *Gebäude* und *Räume* kann die Beziehung *liegt_in* verlagert werden in die Relation Räume:

Räume : {[GebNr, RaumNr, Grösse] }

Eine Beziehung *bewohnt* zwischen *Professoren* und *Räume* benötigt als Fremdschlüssel zum einen die Personalnummer des Professors und zum anderen die Kombination von Gebäude-Nummer und Raum-Nummer:

bewohnt : {[PersNr, GebNr, RaumNr] }

Da *bewohnt* eine $1:1$-Beziehung darstellt, kann sie durch einen Fremdschlüssel beim Professor realisiert werden. Ist die beim *Gebäude* hinterlegte Information eher gering, käme auch, wie im Universitätsschema in Abbildung 56 gezeigt, ein Attribut *Raum* bei den *Professoren* infrage.

6.4 Abfragesprachen

Es gibt verschiedene Konzepte für formale Sprachen zur Formulierung einer Anfrage (Query) an ein relationales Datenbanksystem:

- **Relationenalgebra (prozedural):**
  Verknüpft konstruktiv die vorhandenen Relationen durch Operatoren wie $\cup, \cap, \ldots$:
- **Relationenkalkül (deklarativ):**
  Beschreibt Eigenschaften des gewünschten Ergebnisses mit Hilfe einer Formel der Prädikatenlogik 1. Stufe unter Verwendung von $\wedge, \vee, \neg, \exists, \forall$.
- **Query by Example (für Analphabeten):**
  Verlangt vom Anwender das Ausfüllen eines Gerüstes mit Beispiel-Einträgen.
- **SQL (kommerziell):**
  Stellt eine in Umgangssprache gegossene Mischung aus Relationenalgebra und Relationenkalkül dar.

6.5 Relationenalgebra

Die Operanden der Sprache sind Relationen. Als unabhängige Operatoren gibt es *Selektion, Projektion, Vereinigung, Mengendifferenz, Kartesisches Produkt, Umbenennung*; daraus lassen sich weitere Operatoren *Verbund, Durchschnitt, Division* ableiten.

Selektion : Es werden diejenigen Tupel ausgewählt, die das *Selektionsprädikat* erfüllen. Die Selektion wird mit σ bezeichnet und hat das Selektionsprädikat als Subskript.

Die Selektion

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

liefert als Ergebnis

$\sigma_{Semester>10}(Studenten)$		
MatrNr	Name	Semester
24002	Xenokrates	18
25403	Jonas	12

Das Selektionsprädikat wird beschrieben durch eine Formel F mit folgenden Bestandteilen:

- Attributnamen der Argumentrelation R oder Konstanten als Operanden
- arithmetische Vergleichsoperatoren $< = > \leq \neq \geq$
- logische Operatoren: $\wedge \ \vee \ \neg$ (und oder nicht)

Projektion : Bei der Projektion werden Spalten der Argumentrelation extrahiert. Das Operatorsymbol lautet Π, die gewünschten Attribute werden im Subskript aufgelistet:

$$\Pi_{Rang}(Professoren)$$

liefert als Ergebnis

$\Pi_{Rang}(Professoren)$
Rang
C4
C3

Die Attributmenge wird üblicherweise nicht unter Verwendung von Mengenklammern sondern als durch Kommata getrennte Sequenz gegeben. Achtung: Da das Ergebnis wiederum eine Relation ist, existieren per definitionem keine Duplikate! In der Praxis müssen sie dennoch algorithmisch entfernt werden.

Vereinigung : Zwei Relationen mit gleichem Schema können durch die Vereinigung, symbolisiert durch $\cup$, zusammengefast werden. Beispiel:

$$\Pi_{PersNr,Name}(Assistenten) \cup \Pi_{PersNr,Name}(Professoren)$$

Mengendifferenz : Für zwei Relationen R und S mit gleichem Schema ist die Mengendifferenz $R \ - \ S$ definiert als die Menge der Tupel, die in R aber nicht in S vorkommen. Beispiel

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

liefert die Matrikelnummern derjenigen Studenten, die noch nicht geprüft wurden.

Kartesisches Produkt : Das kartesische Produkt (Kreuzprodukt) zweier Relationen R und S wird mit $R \times S$ bezeichnet und enthält alle möglichen Paare von Tupeln aus R und S. Das Schema der Ergebnisrelation, also **sch**$(R \times S)$, ist die Vereinigung der Attribute aus **sch**(R) und **sch**(S).

Das Kreuzprodukt von *Professoren* und *hören* hat 6 Attribute und enthält $91\ (= 7 \cdot 13)$ Tupel.

Professoren $\times$ hören					
Professoren				hören	
PersNr	name	Rang	Raum	MatrNr	VorlNr
2125	Sokrates	C4	226	26120	5001
...	...	...	...	...	...
2125	Sokrates	C4	226	29555	5001
...	...	...	...	...	...
2137	Kant	C4	7	29555	5001

Haben beide Argumentrelationen ein gleichnamiges Attribut A, so kann dies durch Voranstellung des Relationennamen R in der Form $R.A$ identifiziert werden.

Umbenennung von Relationen und Attributen : Zum Umbenennen von Relationen und Attributen wird der Operator ρ verwendet, wobei im Subskript entweder der neue Relationenname steht oder die Kombination von neuen und altem Attributnamen durch einen Linkspfeil getrennt. Beispiele:

$$\rho_{Dozenten}(Professoren)$$
$$\rho_{Zimmer \leftarrow Raum}(Professoren)$$

Eine Umbenennung kann dann erforderlich werden, wenn durch das kartesische Produkt Relationen mit identischen Attributnamen kombiniert werden sollen. Als Beispiel betrachten wir das Problem, die Vorgänger der Vorgänger der Vorlesung mit der Nummer 5216 herausfinden. Hierzu ist nach Umbenennung ein kartesisches Produkt der Tabelle mit sich selbst erforderlich:

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

Die konstruierte Tabelle hat vier Spalten und enthält das Lösungstupel mit dem Wert 5001 als Vorgänger von 5041, welches wiederum der Vorgänger von 5216 ist:

V1		V2	
Vorgänger	Nachfolger	Vorgänger	Nachfolger
5001	5041	5001	5041
...	...	...	...
5001	5041	5041	5216
...	...	...	...
5052	5259	5052	5259

Natürlicher Verbund (Join) : Der sogenannte *natürliche Verbund* zweier Relationen R und S wird mit $R \bowtie S$ gebildet. Wenn R insgesamt $m + k$ Attribute $A_1, \ldots, A_m, B_1, \ldots, B_k$ und S insgesamt $n + k$ Attribute

$B_1, \ldots, B_k, C_1, \ldots, C_n$ hat, dann hat $R \bowtie S$ die Stelligkeit $m + k + n$. Hierbei wird vorausgesetzt, das die Attribute A_i und C_j jeweils paarweise verschieden sind. Das Ergebnis von $R \bowtie S$ ist definiert als

$$R \bowtie S \ := \ \Pi_{A_1,\ldots,A_m,R.B_1,\ldots,R.B_k,C_1,\ldots,C_n}(\sigma_{R.B_1=S.B_1 \wedge \ldots \wedge R.B_k=S.B_k}(R \times S))$$

Es wird also das kartesische Produkt gebildet, aus dem nur diejenigen Tupel selektiert werden, deren Attributwerte für gleichbenannte Attribute der beiden Argumentrelationen gleich sind. Diese gleichbenannten Attribute werden in das Ergebnis nur einmal übernommen.

Die Verknüpfung der *Studenten* mit ihren *Vorlesungen* geschieht durch

$$(Studenten \bowtie hören) \bowtie Vorlesungen$$

Das Ergebnis ist eine 7-stellige Relation:

$(Studenten \bowtie hören) \bowtie Vorlesungen$						
MatrNr	Name	Semester	VorlNr	Titel	SWS	gelesenVon
26120	Fichte	10	5001	Grundzüge	4	2137
25403	Jonas	12	5022	Glaube und Wissen	2	2137
28106	Carnap	3	4052	Wissenschaftstheorie	3	2126
...	...	...	...	...	...	...

Da der Join-Operator assoziativ ist, können wir auch auf die Klammerung verzichten und einfach schreiben

$$Studenten \bowtie hören \bowtie Vorlesungen$$

Wenn zwei Relationen verbunden werden sollen bzgl. zweier Attribute, die zwar die gleiche Bedeutung aber unterschiedliche Benennungen haben, so müssen diese vor dem Join mit dem ρ-Operator umbenannt werden. Zum Beispiel liefert

$$Vorlesungen \bowtie \rho_{gelesenVon \leftarrow PersNr}(Professoren)$$

die Relation $\{[VorlNr, Titel, SWS, gelesenVon, Name, Rang, Raum]\}$

Allgemeiner Join : Beim natürlichen Verbund müssen die Werte der gleichbenannten Attribute übereinstimmen. Der allgemeine Join-Operator, auch *Theta-Join* genannt, erlaubt die Spezifikation eines beliebigen Join-Prädikats θ. Ein Theta-Join $R \bowtie_\theta S$ über der Relation R mit den Attributen $A_1, A_2, \ldots, A_n$ und der Relation S mit den Attributen $B_1, B_2, \ldots, B_n$ verlangt die Einhaltung des Prädikats θ, beispielsweise in der Form

$$R \bowtie_{A_1 < B_1 \wedge A_2 = B_2 \wedge A_3 < B_5} S$$

Das Ergebnis ist eine $n + m$-stellige Relation und läst sich auch als Kombination von Kreuzprodukt und Selektion schreiben:

$$R \bowtie_\theta S := \sigma_\theta(R \times S)$$

Wenn in der Universitätsdatenbank die *Professoren* und die *Assistenten* um das Attribut *Gehalt* erweitert würden, so könnten wir diejenigen Professoren ermitteln, deren zugeordnete Assistenten mehr als sie selbst verdienen:

$$Professoren \bowtie_{Professoren.Gehalt < Assistenten.Gehalt \wedge Boss = Professoren.PersNr} Assistenten$$

Outer Join: Die bisher genannten Join-Operatoren werden auch innere Joins genannt (*inner join*). Bei ihnen gehen diejenigen Tupel der Argumentrelationen verloren, die keinen Join-Partner gefunden haben. Bei den äuseren Join-Operatoren (*outer joins*) werden - je nach Typ des Joins - auch partnerlose Tupel gerettet:

- left outer join: Die Tupel der linken Argumentrelation bleiben erhalten
- right outer join: Die Tupel der rechten Argumentrelation bleiben erhalten
- full outer join: Die Tupel beider Argumentrelationen bleiben erhalten

Somit lassen sich zu zwei Relationen L und R insgesamt vier verschiedene Joins konstruieren:

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

inner Join				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

left outer join				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-

right outer join				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
-	-	c_3	d_2	e_2

left outer join				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	-	-
-	-	c_3	d_2	e_2

Mengendurchschnitt : Der Mengendurchschnitt (Operatorsymbol $\cap$) ist anwendbar auf zwei Argumentrelationen mit gleichem Schema. Im Ergebnis liegen alle Tupel, die in beiden Argumentrelationen liegen. Beispiel:

$$\Pi_{PersNr}(\rho_{PersNr \leftarrow gelesenVon}(Vorlesungen)) \cap \Pi_{PersNr}(\sigma_{Rang=C4}(Professoren))$$

liefert die Personalnummer aller C4-Professoren, die mindestens eine Vorlesung halten.

Der Mengendurchschnitt läst sich mithilfe der Mengendifferenz bilden:

$$R \cap S \;=\; R \setminus (R \setminus S)$$

Division : Sei R eine r-stellige Relation, sei S eine s-stellige Relation, deren Attributmenge in R enthalten ist.

Mit der Division

$$Q \;:=\; R \;\div\; S \;:=\; \{t = t_1, t_2, \ldots, t_{r-s} \mid \forall u \in S \;:\; tu \in R\}$$

sind alle die Anfangsstücke von R gemeint, zu denen sämtliche Verlängerungen mit Tupeln aus S in der Relation R liegen.

Beispiel:

R	
M	*V*
m_1	v_1
m_1	v_2
m_1	v_3
m_2	v_2
m_2	v_3

$\div$

S
V
v_1
v_2

=

R $\div$ *S*
M
m_1

Die Division von R durch S läßt sich schrittweise mithilfe der unabhängigen Operatoren nachvollziehen (zur Vereinfachung werden hier die Attribute statt über ihre Namen über ihren Index projiziert):

$T := \pi_{1,\ldots,r-s}(R)$	alle Anfangsstücke
$T \times S$	diese kombiniert mit allen Verlängerungen aus S
$(T \times S) \backslash R$	davon nur solche, die nicht in R sind
$V := \pi_{1,\ldots,r-s}((T \times S) \backslash R)$	davon die Anfangsstücke
$T \backslash V$	davon das Komplement

Beispiel für eine Query, die mit dem Divisionsoperator gelöst werden kann:

Liste die Namen der Studenten, die jeweils alle 4-stündigen Vorlesungen hören:

$$\Pi_{Name}(Studenten \bowtie (hoeren \div \Pi_{VorlNr}(\rho_{SWS=4}(Vorlesungen))))$$

6.6 Relationenkalkül

Ausdrücke in der Relationenalgebra spezifizieren konstruktiv, wie das Ergebnis der Anfrage zu berechnen ist. Demgegenüber ist der *Relationenkalkül* stärker *deklarativ* orientiert. Er beruht auf dem mathematischen Prädikatenkalkül erster Stufe, der quantifizierte Variablen zuläst. Es gibt zwei unterschiedliche, aber gleichmächtige Ausprägungen des Relationenkalküls:

- Der relationale Tupelkalkül
- Der relationale Domänenkalkül

6.7 Der relationale Tupelkalkül

Im *relationalen Tupelkalkül* hat eine Anfrage die generische Form

$$\{t \mid P(t)\}$$

wobei t eine sogenannte Tupelvariable ist und P ist ein Prädikat, das erfüllt sein mus, damit t in das Ergebnis aufgenommen werden kann. Das Prädikat P wird formuliert unter Verwendung von $\vee, \wedge, \neg, \forall, \exists, \Rightarrow$.

Query: Alle C4-Professoren:

$$\{p \mid p \in Professoren \wedge p.Rang = 'C4'\}$$

Query: Alle Professorennamen zusammen mit den Personalnummern ihrer Assistenten:

$$\{[p.Name, a.PersNr] \mid p \in Professoren \wedge a \in Assistenten \wedge p.PersNr = a.Boss\}$$

Query: Alle diejenigen Studenten, die sämtliche vierstündigen Vorlesungen gehört haben:

$$\{s \mid s \in Studenten \wedge \forall v \in Vorlesungen \\ (v.SWS = 4 \Rightarrow \exists h \in hören(h.VorlNr = v.VorlNr \wedge h.MatrNr = s.MatrNr))\}$$

Für die Äquivalenz des Tupelkalküls mit der Relationenalgebra ist es wichtig, sich auf sogenannte *sichere* Ausdrücke zu beschränken, d.h. Ausdrücke, deren Ergebnis wiederum eine Teilmenge der Domäne ist. Zum Beispiel ist der Ausdruck

$$\{n \mid \neg(n \in Professoren)\}$$

nicht sicher, da er unendlich viele Tupel enthält, die nicht in der Relation *Professoren* enthalten sind.

6.8 Der relationale Domänenkalkül

Im *relationalen Domänenkalkül* werden Variable nicht an Tupel, sondern an Domänen, d.h. Wertemengen von Attributen, gebunden. Eine Anfrage hat folgende generische Struktur:

$$\{[v_1, v_2, \ldots, v_n] \mid P(v_1, v_2, \ldots, v_n)\}$$

Hierbei sind die v_i Domänenvariablen, die einen Attributwert repräsentieren. P ist eine Formel der Prädikatenlogik 1. Stufe mit den freien Variablen $v_1, v_2, \ldots, v_n$.

Join-Bedingungen können implizit durch die Verwendung derselben Domänenvariable spezifiziert werden. Beispiel:

Query: Alle Professorennamen zusammen mit den Personalnummern ihrer Assistenten:

$$\{[n, a] \mid \exists p, r, t([p, n, r, t] \in Professoren \\ \wedge \exists v, w([a, v, w, p] \in Assistenten))\}$$

Wegen des Existenz- und Allquantors ist die Definition des *sicheren Ausdruckes* etwas aufwendiger als beim Tupelkalkül. Da sich diese Quantoren beim Tupelkalkül immer auf Tupel einer vorhandenen Relation bezogen, war automatisch sichergestellt, das das Ergebnis eine endliche Menge war.

6.9 Query by Example

Query-by-Example (QBE) beruht auf dem relationalen Domänenkalkül und erwartet vom Benutzer das beispielhafte Ausfüllen eines Tabellenskeletts.

Liste alle Vorlesungen mit mehr als 3 SWS:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
		p._t	> 3	

Die Spalten eines Formulars enthalten Variablen, Konstanten, Bedingungen und Kommandos. Variablen beginnen mit einem Unterstrich (_), Konstanten haben keinen Präfix. Der Druckbefehl **p._t** veranlast die Ausgabe von _t.

Im Domänenkalkül lautet diese Anfrage

$$\{[t]|\exists v, s, r([v, t, s, r] \in \ Vorlesungen \wedge s > 3)\}$$

Ein Join wird durch die Bindung einer Variablen an mehrere Spalten möglich:

Liste alle Professoren, die Logik lesen:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
		Logik		_x

Professoren	PersNr	Name	Rang	Raum
	_x	p._n		

Über eine *condition box* wird das Einhalten von Bedingungen erzwungen:

Liste alle Studenten, die in einem höheren Semester sind als Feuerbach:

Studenten	MatrNr	Name	Semester
		p._s	_a
		Feuerbach	_b

conditions
_a> _b

Das Kommando zur Gruppierung lautet **g. ,** hierdurch werden alle Tupel einer Relation zusammengefast (gruppiert), die bezüglich eines Attributes gleiche Werte haben. Innerhalb einer Gruppe kann dann über Aggregatfunktionen summiert, minimiert, maximiert, der Durchschnitt gebildet oder einfach nur gezählt werden. Die Schlüsselworte dafür heisen **sum.**, **avg.**, **min.**, **max.** und **cnt.**. Standardmäsig werden innerhalb einer Gruppe die Duplikate eliminiert. Die Duplikateliminierung wird durch **all.** unterdrückt:

Liste die Summe der SWS der Professoren, die überwiegend lange Vorlesungen halten:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
			p.sum.all._x	p.g.

conditions
avg.all._x>2

Einfügen, Ändern und Löschen geschieht mit den Kommandos **i.**, **u.**, **d.**.

Füge neuen Studenten ein:

Studenten	MatrNr	Name	Semester
i.	4711	`Wacker`	5

Setze die Semesterzahlen von Feuerbach auf 3:

Studenten	MatrNr	Name	Semester
u.		`Feuerbach`	u.3

Entferne Sokrates und alle seine Vorlesungen:

Professoren	PersNr	Name	Rang	Raum
d.	_x	`Sokrates`		

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
d.	_y			_x

hören	VorlNr	MatrNr
d.	_y	

6.10 SQL

SQL wird ausführlich behandelt in Kapitel 7. Hier sei nur vorab auf die offensichtliche Verwandtschaft mit der Relationenalgebra und dem Relationenkalkül hingewiesen:

Query: Die Namen der Studenten, die 4-stündige Vorlesungen hören:

```
select s.name
from studenten s, hoeren h, vorlesungen v
where s.matrnr = h.matrnr
and   h.vorlnr = v.vorlnr
and   v.sws = 4
```

Query: Die Namen der Studenten, die jeweils alle 4-stündigen Vorlesungen hören:

Wir erinnern uns an die entsprechende Formulierung im relationalen Tupelkalkül:

$$\{s.name \mid s \in Studenten \wedge \forall v \in Vorlesungen$$
$$(v.SWS = 4 \Rightarrow \exists h \in hören(h.VorlNr = v.VorlNr \wedge h.MatrNr = s.MatrNr))\}$$

SQL kennt keinen All-Quantor (wohl aber einen Existenz-Quantor) und auch keinen Implikationsoperator. Wir nutzen daher folgende Äquivalenzen:

$\forall t \in R(P(t))$ ist äquivalent zu $\neg(\exists t \in R(\neg(t)))$

$A \Rightarrow B$	ist äquivalent zu	$\neg A \vee B$
$\neg(A \vee B)$	ist äquivalent zu	$\neg A \wedge \neg B$

Daher lässt sich der obige Ausdruck umformen in

$$\{s.name \mid s \in Studenten \wedge \neg(\exists v \in Vorlesungen \\ (v.SWS = 4 \wedge \neg \exists h \in hören(h.VorlNr = v.VorlNr \wedge h.MatrNr = s.MatrNr)))\}$$

Daraus entsteht unmittelbar die SQL-Query

```
select s.name from Studenten s
 where not exists
    (select * from vorlesungen v
     where sws=4 and not exists
       (select * from hoeren h
        where h.vorlnr = v.vorlnr
          and h.matrnr = s.matrnr))
```

7. SQL

Wer die Daten für die folgenden Beispiele als MYSQL-Datenbank selbst erzeugen möchte, kann folgendes Script benutzen.

```
/*
        Dieses Script für MySQL ist nahezu identisch zu dem Originalscript
        für den Microsoft SQL Server 2000 (siehe
        http://www-lehre.inf.uos.de/~dbs/2005/skript/SQL-Server/sql-script.txt.html).
        Allerdings ist in MySQL das Datumsformat anders: Während in Microsoft SQL
       ein Datum in der Form 'TT.MM.JJJJ' gespeichert wird, benutzt MySQL das Format 'JJJJ-MM-TT'.
       Ausserdem wird eine Datenbank 'dbs11' erstellt, in der dann die Tabellen angelegt werden.
*/

 CREATE DATABASE dbs11;
 USE dbs11;

 CREATE TABLE Studenten
        (MatrNr         INTEGER PRIMARY KEY,
         Name           VARCHAR(20) NOT NULL,
         Semester       INTEGER,
         GebDatum       DATE);

 CREATE TABLE Professoren
        (PersNr         INTEGER PRIMARY KEY,
         Name           VARCHAR(20) NOT NULL,
         Rang           CHAR(2) CHECK (Rang in ('C2', 'C3', 'C4')),
         Raum           INTEGER UNIQUE,
         Gebdatum       DATE);

 CREATE TABLE Assistenten
        (PersNr         INTEGER PRIMARY KEY,
         Name           VARCHAR(20) NOT NULL,
         Fachgebiet     VARCHAR(20),
         Boss           INTEGER REFERENCES Professoren,
         GebDatum       DATE);

 CREATE TABLE Vorlesungen
        (VorlNr         INTEGER PRIMARY KEY,
         Titel          VARCHAR(20),
         SWS            INTEGER,
         gelesenVon     INTEGER REFERENCES Professoren);

 CREATE TABLE hoeren
        (MatrNr         INTEGER REFERENCES Studenten   ON UPDATE CASCADE
                                                       ON DELETE CASCADE,
         VorlNr         INTEGER REFERENCES Vorlesungen ON UPDATE CASCADE,
         PRIMARY KEY    (MatrNr, VorlNr));

 CREATE TABLE voraussetzen
        (Vorgaenger     INTEGER REFERENCES Vorlesungen  ON UPDATE CASCADE,
         Nachfolger     INTEGER REFERENCES Vorlesungen,
         PRIMARY KEY    (Vorgaenger, Nachfolger));

 CREATE TABLE pruefen
        (MatrNr         INTEGER REFERENCES Studenten   ON UPDATE CASCADE
                                                       ON DELETE CASCADE,
         VorlNr         INTEGER REFERENCES Vorlesungen ON UPDATE CASCADE,
         PersNr         INTEGER REFERENCES Professoren,
         Note           NUMERIC(3,1) CHECK (Note between 0.7 and 5.0),
         PRIMARY KEY    (MatrNr, VorlNr));

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (24002, 'Xenokrates', 18, '1975-10-23');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (25403, 'Jonas', 12,'1973-09-18');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (26120, 'Fichte', 10,'1967-12-04');
```

```
INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (26830, 'Aristoxenos', 8,'1943-08-05');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (27550, 'Schopenhauer', 6,'1954-06-22');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (28106, 'Carnap', 3,'1979-10-02');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (29120, 'Theophrastos', 2,'1948-04-19');

INSERT INTO Studenten(MatrNr, Name, Semester, GebDatum)
VALUES (29555, 'Feuerbach', 2,'1961-02-12');



INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2125, 'Sokrates', 'C4', 226,'1923-08-23');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2126, 'Russel', 'C4', 232,'1934-07-10');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2127, 'Kopernikus', 'C3', 310,'1962-03-12');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2133, 'Popper', 'C3', 052,'1949-09-03');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2134, 'Augustinus', 'C3', 309,'1939-04-19');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2136, 'Curie', 'C4', 036,'1929-05-10');

INSERT INTO Professoren(PersNr, Name, Rang, Raum, GebDatum)
VALUES (2137, 'Kant', 'C4', 007,'1950-04-04');



INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3002, 'Platon', 'Ideenlehre', 2125, '1966-08-14');

INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3003, 'Aristoteles', 'Syllogistik', 2125,'1970-12-23');

INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3004, 'Wittgenstein', 'Sprachtheorie', 2126,'1968-08-02');

INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3005, 'Rhetikus', 'Planetenbewegung', 2127,'1962-12-20');

INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3006, 'Newton', 'Keplersche Gesetze', 2127,'1961-11-10');

INSERT INTO Assistenten(PersNr, Name, Fachgebiet, Boss, GebDatum)
VALUES (3007, 'Spinoza', 'Gott und Natur', 2134,'1963-02-08');



INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5001, 'Grundzüge', 4, 2137);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5041, 'Ethik', 4, 2125);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5043, 'Erkenntnistheorie', 3, 2126);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5049, 'Mäeutik', 2, 2125);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (4052, 'Logik', 4, 2125);
```

```
INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5052, 'Wissenschaftstheorie', 3, 2126);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5216, 'Bioethik', 2, 2126);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5259, 'Der Wiener Kreis', 2, 2133);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (5022, 'Glaube und Wissen', 2, 2134);

INSERT INTO Vorlesungen(VorlNr, Titel, SWS, gelesenVon)
VALUES (4630, 'Die 3 Kritiken', 4, 2137);



INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (26120, 5001);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (27550, 5001);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (27550, 4052);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (27550, 5041);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (28106, 4052);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (28106, 5216);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (28106, 5259);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (27550, 4630);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (29120, 5041);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (29120, 5049);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (29555, 5022);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (25403, 5022);

INSERT INTO hoeren(MatrNr, VorlNr)
VALUES (29555, 5001);



INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5001, 5041);

INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5001, 5043);

INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5001, 5049);

INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5041, 5216);

INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5043, 5052);
```

```
INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5041, 5052);

INSERT INTO voraussetzen(Vorgaenger, Nachfolger)
VALUES (5052, 5259);

INSERT INTO pruefen(MatrNr, VorlNr, PersNr, Note)
VALUES (28106, 5001, 2126, 1.0);

INSERT INTO pruefen(MatrNr, VorlNr, PersNr, Note)
VALUES (25403, 5041, 2125, 2.0);

INSERT INTO pruefen(MatrNr, VorlNr, PersNr, Note)
VALUES (27550, 4630, 2137, 2.0);
```

7.1 MySQL

Stellvertretend für die zahlreichen am Markt befindlichen relationalen Datenbanksysteme wird in diesem Kapitel das System *MySQL 5.1* verwendet. MySQL ist Open Source und für viele Plattformen erhältlich, darunter Linux, Microsoft Windows und Mac OS X. MySQL hatte lange Zeit nicht den gleichen Funktionsumfang wie kommerzielle Datenbanksysteme (Oracle, IBM DB2, MS SQL-Server). Mit Erscheinen der Version 5.0 konnte der MySQL-Server aber stark aufholen und beherrscht nun unter anderem Views, Trigger und Stored Procedures.

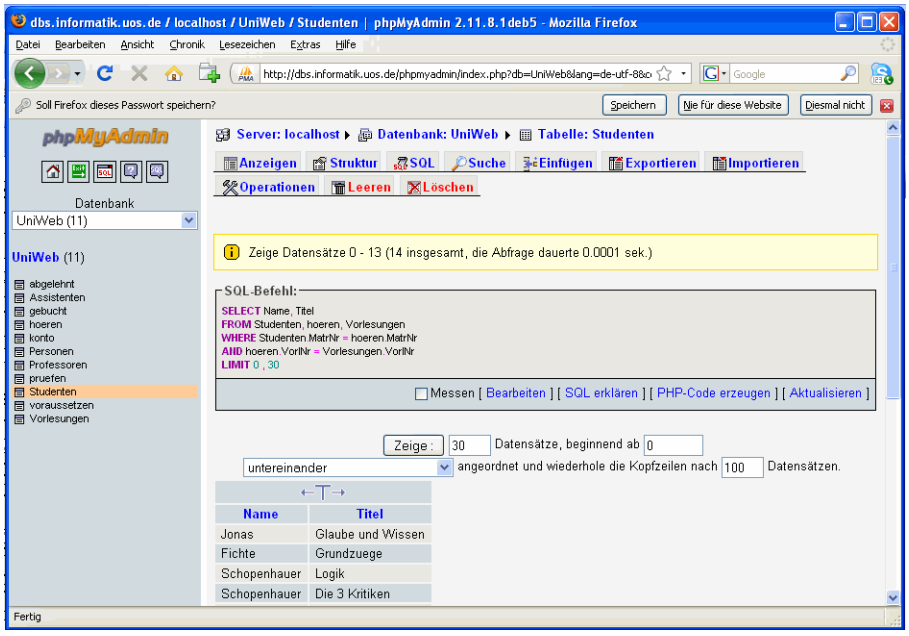


Abb. 59: Abfrage mit phpmyadmin

Zur Verwaltung von Datenbanken dienen die mitgelieferten Kommandozeilen-Clients `mysql` und `mysqladmin`. Eine Alternative zur Kommandozeile stellt die Open Source Anwendung *phpMyAdmin* dar, die eine grafische Benutzeroberfläche zur Verfügung stellt. phpMyAdmin ist in PHP geschrieben und läuft im Webbrowser. Es ist dadurch plattform-unabhängig und in der Lage, auch Datenbanken zu administrieren, die auf anderen Rechnern laufen.

Für die Administation einer Datenbank mit Hilfe von phpMyAdmin sind keine Kenntnisse in SQL notwendig. phpMyAdmin bietet allerdings auch eine Schnitstelle, um eigene SQL-Anfragen zu stellen.

Zum Zugriff auf eine MySQL-Datenbank eignet sich die MySQLWorkbench (siehe Abbildungen 60 und 61).

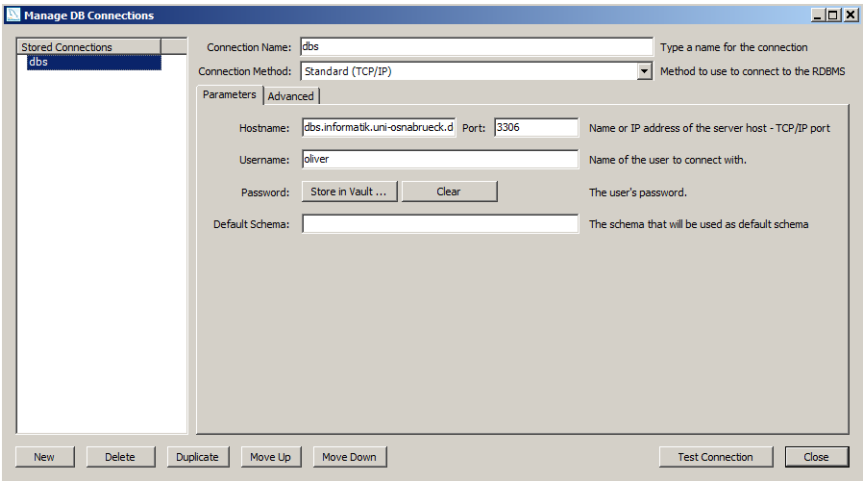


Abb. 60: Einrichten der Datenbankverbindung in MySQLWorkbench

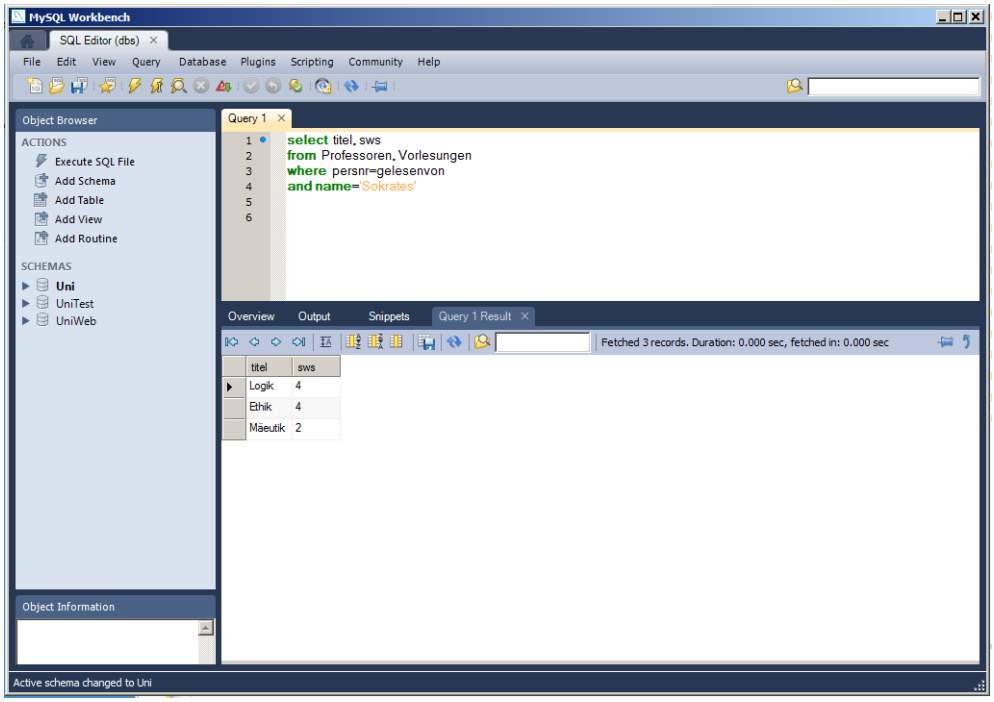


Abb. 61: Abfrage in MySQLWorkbench

7.2 Sprachphilosophie

Die Relationale Algebra und der Relationenkalkül bilden die Grundlage für die Anfragesprache SQL. Zusätzlich zur Manipulation von Tabellen sind Möglichkeiten zur Definition des relationalen Schemas, zur Formulierung von Integritätsbedingungen, zur Vergabe von Zugriffsrechten und zur Transaktionskontrolle vorgesehen.

Relationale Datenbanksysteme realisieren keine Relationen im mathematischen Sinne, sondern Tabellen, die durchaus doppelte Einträge enthalten können. Bei Bedarf müssen die Duplikate explizit entfernt werden.

SQL geht zurück auf den von IBM Anfang der 70er Jahre entwickelten Prototyp *System R* mit der Anfragesprache *Sequel*. Der zur Zeit aktuelle Standard lautet SQL-92, auch SQL 2 genannt. Er wird weitgehend vom relationalen Datenbanksystem *SQL-Server* unterstützt.

7.3 Datentypen

Der MySQL-Server verwendet unter anderem folgende Datentypen:

Typ	Bytes	Wertebereich
`bigint`	8	ganze Zahlen von -2^{63} bis $+2^{63}$
`int`	4	ganze Zahlen von -2^{31} bis $+2^{31}$
`mediumint`	3	ganze Zahlen von -2^{23} bis $+2^{23}$
`smallint`	2	ganze Zahlen von -2^{15} bis $+2^{15}$
`tinyint`	1	ganze Zahlen von -128 bis 127
`bool, boolean`	1	aliase für tinyint(1)
`decimal(n,k)`	n	numerische Daten mit fester Genauigkeit mit n Stellen, davon k Nachkommastellen
`numeric(n,k)`	n	entspricht decimal
`float`	4	Gleitkommazahlen von -10^{38} bis $+10^{38}$
`double`	8	Gleitkommazahlen von -10^{308} bis $+10^{308}$
`real`	8	alias für double
`date`	3	Datumsangaben von 01.01.1000 bis 31.12.9999
`datetime`	8	Zeitangaben von 01.01.1000 00:00:00 Uhr bis 31.12.9999 23:59:59 Uhr
`timestamp`	4	Zeitangaben von 01.01.1970 bis 2037
`time`	3	Zeitintervall mit Stunden, Minuten, Sekunden von -838:59:59 bis 838:59:59
`year`	1	Jahresangabe von 1901 bis 2155
`char(n)`	n	String fester Länge mit n <= 255 Zeichen
`varchar(n)`	n	String variabler Länge mit n <= 65535 Zeichen
`text`	n	String variabler Länge mit n <=65535 Zeichen
`binary`	n	Binärdaten fester Länge mit n <= 255 Bytes
`varbinary`	n	Binärdaten variabler Länge mit n <= 65535 Bytes
`blob`	n	Binärdaten variabler Länge mit n <= 65535 Bytes

`NULL` bezeichnet nicht besetzte Attributwerte

`default` bezeichnet vorbesetzte Attributwerte.

`numeric(n,m)`-Werte werden mit n Dezimalstellen angezeigt, davon m Nachkommastellen. Sie werden in einem Binärformat mit jeweils 4 Bytes pro 9 Dezimalstellen (Basis 10) kodiert.

`bigint, int, mediumint, smallint` und `tinyint` sind standardmäsig vorzeichenbehaftet. Durch den Zusatz `unsigned` wird bei ihnen der Wertebereich in die positiven Zahlen verschoben un beginnt bei 0. Durch Angabe eines Parameters kann auserdem die Anzeigebreite angegeben werden, die übrigens Bits werden mit Nullen aufgefüllt. `tinyint(1)` speichert ein Bit.

`bigint, int, mediumint, smallint` und `tinyint` können mit dem Zusatz `auto_increment` versehen werden. Dadurch werden sie automatisch mit einem eindeutigen Wert initialisiert. Standardmäßig betragen Startwert und Schrittweite jeweils 1,; andere Werte können für die Tabelle spezifiert werden.

`datetime` enthlt Angaben zum Datum und zur Uhrzeit. Das Datum YYYY-MM-DD wird als YYYY*10.000 + MM*100 + DD kodiert, die Zeit HH-MM-SS wird als 10.000*HH + MM*100 + SS kodiert.

Spalten vom Typ `timestamp` wird automatisch für die betreffende Zeile bei einer INSERT- oder UPDATE-Operation der aktuelle Zeitstempel zugewiesen. Der Wert wird kodiert als Zahl der Sekunden nach dem 01.01.1970.

Spalten vom Typ `blob` und `text` speichern nur die ersten 255 Bytes innerhalb des Zeilentupels und verteilten ggf. weitere Daten in Blöcken zu je 2.000 Bytes in verborgenen Tabellen.

7.4 SQL-Statements zur Schemadefinition

SQL-Statements zum Anlegen, Erweitern, Verkürzen und Entfernen einer Tabelle:

1. Tabelle anlegen:

```
create table Personen (
  persNr      int primary key auto_increment, -- ganze Zahl, automatisch vergeben
  name        varchar(15) not null,           -- max. 15 Zeichen langer Wert
  geschlecht  boolean default 0,              -- boolscher Wert, vorbesetzt mit 0
  note        decimal (3,2),                  -- 3 Gesamt-, 2 Nachkommastellen
  groesse     float,                          -- einfache Genauigket
  gewicht     double,                         -- doppelte Genauigkeit
  gebDatum    datetime,                       -- Datum
  abschluss   year,                           -- Abschlussjahr
  marathon    time,                           -- Laufzeit fuer Marathon
  bemerkung   text,                           -- laengerer Text
  kombination set ('rot','gruen','blau'),     -- Menge von 3 Farben
  zugriff     timestamp                       -- Zeitstempel fuer Zeilenzugriff
) auto_increment = 100000                     -- beginne bei 1000000
```

2. Tabelle um eine Spalte erweitern:

```
alter table Person
add   Vorname varchar(15)
```

3. Tabellenspalte ändern:

```
alter table Person
modify Vorname varchar(20)
```

4. Tabelle um eine Spalte verkürzen:

```
alter table  Person
drop  column Vorname
```

5. Tabelle entfernen:

```
drop table Person
```

7.5 Aufbau einer SQL-Query zum Anfragen

Eine SQL-Query zum Abfragen von Relationen hat den folgenden generischen Aufbau:

`SELECT`	Spalten-1
`FROM`	Tabellen
`WHERE`	Bedingung-1
`GROUP BY`	Spalten-2
`HAVING`	Bedingung-2
`ORDER BY`	Spalten-3

Nur die Klauseln `SELECT` und `FROM` sind erforderlich, der Rest ist optional.

Es bedeuten ...

Spalten-1	Bezeichnungen der Spalten, die ausgegeben werden
Tabellen	Bezeichnungen der verwendeten Tabellen

Bedingung-1	Auswahlbedingung für die auszugebenden Zeilen; verwendet werden
	`AND OR NOT = > < != <= >= IS NULL BETWEEN IN LIKE`
Spalten-2	Bezeichnungen der Spalten, die eine Gruppe definieren.
	Eine Gruppe bzgl. Spalte x sind diejenigen Zeilen, die bzgl. x
	identische Werte haben.
Bedingung-2	Bedingung zur Auswahl einer Gruppe
Spalten-3	Ordnungsreihenfolge für `<Spalten-1>`

Vor `<Spalten-1>` kann das Schlüsselwort `DISTINCT` stehen, welches identische Ausgabezeilen unterdrückt.

Sogenannte *Aggregate Functions* fassen die Werte einer Spalte oder Gruppe zusammen.

Es liefert ...

`COUNT (*)`	Anzahl der Zeilen
`COUNT (DISTINCT x)`	Anzahl der verschiedenen Werte in Spalte x
`SUM (x)`	Summe der Werte in Spalte x
`SUM (DISTINCT x)`	Summe der verschiedenen Werte in Spalte x
`AVG (x)`	Durchschnitt der Werte in Spalte x
`AVG (DISTINCT x)`	Durchschnitt der verschiedenen Werte in Spalte x
`MAX (x)`	Maximum der Werte in Spalte x
`MIN (x)`	Minimum der Werte in Spalte x

jeweils bezogen auf solche Zeilen, welche die `WHERE`-Bedingung erfüllen. Null-Einträge werden bei `AVG`, `MIN`, `MAX` und `SUM` ignoriert.

Spalten der Ergebnisrelation können umbenannt werden mit Hilfe der `AS`-Klausel.

7.6 SQL-Queries zum Anfragen

Folgende Beispiele beziehen sich auf die Universitätsdatenbank, wobei die Relationen *Professoren, Assistenten* und *Studenten* jeweils um ein Attribut *GebDatum* vom Typ *Datetime* erweitert worden sind.

1. Liste alle Studenten:

```
select * from Studenten
```

2. Liste Personalnummer und Name der C4-Professoren:

```
select PersNr, Name
from   Professoren
where  Rang='C4'
```

3. Zähle alle Studenten:

```
select count(*)
from   Studenten
```

4. Liste Namen und Studiendauer in Jahren von allen Studenten,

```
select Name, Semester/2 as Studienjahr
from   Studenten
where  Semester is not null
```

5. Liste alle Studenten mit Semesterzahlen zwischen 1 und 4:

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

alternativ

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

alternativ

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

6. Liste alle Vorlesungen, die im Titel den String `Ethik` enthalten, klein oder gros geschrieben:

```
select *
from   Vorlesungen
where  Titel like '%ETHIK'
```

7. Liste Personalnummer, Name und Rang aller Professoren, absteigend sortiert nach Rang, innerhalb des Rangs aufsteigend sortiert nach Name:

```
select   PersNr, Name, Rang
from     Professoren
order by Rang desc, Name asc
```

8. Liste alle verschiedenen Einträge in der Spalte Rang der Relation Professoren:

```
select distinct Rang
from            Professoren
```

9. Liste alle Geburtstage mit ausgeschriebenem Monatsnamen:

```
select  Name,
        Day(Gebdatum)       as Tag,
        Monthname(GebDatum) as Monat,
        Year(GebDatum)      as Jahr
from    Studenten
```

10. Liste das Alter der Studenten in Jahren:

```
select  Name, Year(Now()) - Year(GebDatum) as Jahre
from    Studenten
```

11. Liste die Wochentage der Geburtsdaten der Studenten:

```
select  Name, Dayname(GebDatum) as Wochentag
from    Studenten
```

12. Liste die Kalenderwochen der Geburtsdaten der Studenten:

```
select  Name, Week(GebDatum) as Kalenderwoche
from      Studenten
```

13. Liste den Dozenten der Vorlesung Logik:

```
select  Name, Titel
```

```
from    Professoren, Vorlesungen
where   PersNr = gelesenVon and Titel = 'Logik'
```

14. Liste die Namen der Studenten mit ihren Vorlesungstiteln:

```
select  Name, Titel
from    Studenten, hoeren, Vorlesungen
where   Studenten.MatrNr = hoeren.MatrNr
and     hoeren.VorlNr    = Vorlesungen.VorlNr
```

alternativ:

```
select  s.Name, v.Titel
from    Studenten s, hoeren h, Vorlesungen v
where   s.MatrNr = h.MatrNr
and     h.VorlNr = v.VorlNr
```

15. Liste die Namen der Assistenten, die für denselben Professor arbeiten, für den Aristoteles arbeitet:

```
select  a2.Name
from    Assistenten a1, Assistenten a2
where   a2.boss  =  a1.boss
and     a1.name  = 'Aristoteles'
and     a2.name != 'Aristoteles'
```

16. Liste die durchschnittliche Semesterzahl:

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

17. Liste Geburtstage der Gehaltsklassenältesten (ohne Namen!):

```
select   Rang, min(GebDatum) as Aeltester
from     Professoren
group by Rang
```

18. Liste Summe der SWS pro Professor:

```
select   gelesenVon as PersNr, sum(SWS) as Lehrbelastung
from     Vorlesungen
group by gelesenVon
```

19. Liste Summe der SWS pro Professor, sofern seine Durchschnitts-SWS gröser als 3 ist:

```
select   gelesenVon as PersNr, sum(SWS) as Lehrbelastung
from     Vorlesungen
group by gelesenVon
having   avg(SWS) > 3
```

20. Liste Summe der SWS pro C4-Professor, sofern seine Durchschnitts-SWS gröser als 3 ist:

```
select   Name, sum(SWS)
from     Vorlesungen, Professoren
where    gelesenVon = PersNr and Rang='C4'
group by gelesenVon, Name
having   avg(cast(SWS as decimal)) > 3.0
```

21. Liste alle Prüfungen, die als Ergebnis die schlechteste Note haben:

```
select *
from   pruefen
where  Note = (select max(Note) from pruefen)
```

22. Liste alle Professoren zusammen mit ihrer Lehrbelastung:

```
select PersNr, Name, (select sum(SWS)
                      from Vorlesungen
```

```
                    where gelesenVon = PersNr) as Lehrbelastung
from Professoren
```

23. Liste alle Studenten, die älter sind als der jüngste Professor:

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

Alternativ:

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

24. Liste alle Assistenten, die für einen jüngeren Professor arbeiten:

```
select a.*
from   Assistenten a, Professoren p
where  a.Boss = p.PersNr
and    a.GebDatum < p.GebDatum
```

25. Liste alle Studenten mit der Zahl ihrer Vorlesungen, sofern diese Zahl gröser als 2 ist:

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

26. Liste die Namen und Geburtstage der Gehaltsklassenältesten:

```
select p.Rang, p.Name, p.gebdatum
from   Professoren p,
       (select Rang, min(GebDatum) as maximum
        from Professoren
        group by Rang) tmp
where  p.Rang = tmp.Rang and p.GebDatum = tmp.maximum
```

27. Liste Vorlesungen zusammen mit Marktanteil, definiert als = Hörerzahl/Gesamtzahl:

```
select h.VorlNr, h.AnzProVorl, g.GesamtAnz,
       h.AnzProVorl / g.GesamtAnz as Marktanteil
from   (select VorlNr, count(*) as AnzProVorl
        from hoeren group by VorlNr) h,
       (select count(*) as GesamtAnz
        from Studenten) g
```

28. Liste die Vereinigung von Professoren- und Assistenten-Namen:

```
(select Name from Assistenten)
union
(select Name from Professoren)
```

29. Liste die Differenz von Professoren- und Assistenten-Namen (nur SQL-92):

```
(select Name from Assistenten)
minus
(select Name from Professoren)
```

30. Liste den Durchschnitt von Professoren- und Assistenten-Namen (nur SQL-92):

```
(select Name from Assistenten)
intersect
```

```
(select Name from Professoren)
```

31. Liste alle Professoren, die keine Vorlesung halten:

```
select Name
from   Professoren
where  PersNr not in ( select gelesenVon from Vorlesungen )
```

Alternativ:

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

32. Liste Studenten mit gröster Semesterzahl:

```
select Name
from   Studenten
where  Semester >= all ( select Semester
                         from Studenten )
```

33. Liste Studenten, die nicht die gröste Semesterzahl haben:

```
select Name
from   Studenten
where  Semester < some ( select Semester
                         from Studenten )
```

34. Liste solche Studenten, die alle 4-stündigen Vorlesungen hören:

```
select s.*
from   Studenten s
where  not exists
  (select *
  from Vorlesungen v
  where v.SWS = 4 and not exists
    (select *
     from hoeren h
     where h.VorlNr = v.VorlNr and h.MatrNr = s.MatrNr
    )
  )
```

Um alle direkten und indirekten Voraussetzungen für die Vorlesung 'Der Wiener Kreis' zu ermitteln, muss die transitive Hülle der rekursiven Relation `voraussetzen` berechnet werden. Hierzu gibt es in verschiedenen Programmiersprachen und SQL-Dialekten unterschiedliche Ansätze:

1. Definition der transitiven Hülle einer rekursiven Relation in Prolog:

```
Trans(V,N) :- voraussetzen(V,N).
Trans(V,N) :- Trans(V,Z), voraussetzen(Z,N)
```

2. Berechnung der transitiven Hülle einer rekursiven Relation in DB2:

```
with Trans(vorgaenger, nachfolger)
as
(select vorgaenger, nachfolger from voraussetzen
 union all
 select t.vorgaenger, v.nachfolger
 from Trans t, voraussetzen v
 where t.nachfolger = v.vorgaenger
)

select titel from Vorlesungen where vorlnr in
  (select vorgaenger from Trans where nachfolger in
    (select vorlnr from Vorlesungen
```

```
     where titel='Der Wiener Kreis'))
```

3. Berechnung der transitiven Hülle einer rekursiven Relation in Oracle:

```
select Titel
from   Vorlesungen
where  VorlNr in (
        select Vorgaenger
        from voraussetzen
        connect by Nachfolger = prior Vorgaenger
        start with Nachfolger = (
                select VorlNr
                from Vorlesungen
                where Titel = 'Der Wiener Kreis'
        )
)
```

4. Idee zur Berechnung der transitiven Hülle einer rekursiven Relation in MySQL:

```
create temporary table nach (nr integer)
create temporary table vor (nr integer)

insert into nach
    select vorlnr from Vorlesungen
    where titel = 'Der Wiener Kreis'

-- mehrfach iterieren:

insert into vor
    select v.vorgaenger
    from voraussetzen v
    where v.nachfolger in (select * from nach)

delete from nach

insert into nach
    select distinct nr from vor
```

7.7 SQL-Statements zum Einfügen, Modifizieren und Löschen

1. Füge neue Vorlesung mit einigen Angaben ein:

```
insert into Vorlesungen (VorlNr, Titel, gelesenVon)
values (4711,'Selber Atmen', 2125)
```

2. Schicke alle Studenten in die Vorlesung *Selber Atmen*:

```
insert into hoeren
select MatrNr, VorlNr
from   Studenten, Vorlesungen
where  Titel = 'Selber Atmen'
```

3. Erweitere die neue Vorlesung um ihre Semesterwochenstundenzahl:

```
update Vorlesungen
set    SWS=6
where  Titel='Selber Atmen'
```

4. Entferne alle Studenten aus der Vorlesung *Selber Atmen*:

```
delete from hoeren
where vorlnr =
     (select VorlNr from Vorlesungen
      where  Titel = 'Selber Atmen')
```

5. Entferne die Vorlesung *Selber Atmen*:

```
delete from Vorlesungen
where titel = 'Selber Atmen'
```

7.8 SQL-Statements zum Anlegen von Sichten

Die mangelnden Modellierungsmöglichkeiten des relationalen Modells in Bezug auf Generalisierung und Spezialisierung können teilweise kompensiert werden durch die Verwendung von Sichten. Nicht alle Sichten sind *update-fähig*, da sich eine Änderung ihrer Daten nicht immer auf die Originaltabellen zurückpropagieren läst

1. Lege Sicht an für Prüfungen ohne Note:

```
create view pruefenSicht as
select MatrNr, VorlNr, PersNr
from   pruefen;
```

2. Lege Sicht an für Studenten mit ihren Professoren:

```
create view StudProf (Sname, Semester, Titel, PName) as
select s.Name, s.Semester, v.Titel, p.Name
from   Studenten s, hoeren h, Vorlesungen v, Professoren p
where  s.MatrNr     = h.MatrNr
and    h.VorlNr     = v.VorlNr
and    v.gelesenVon = p.PersNr;
```

3. Lege Sicht an mit Professoren und ihren Durchschnittsnoten:

```
create   view ProfNote (PersNr, Durchschnittsnote) as
select   PersNr, avg (Note)
from     pruefen
group by PersNr;
```

4. Entferne die Sichten wieder:

```
drop view PruefenSicht;
drop view StudProf;
drop view ProfNote;
```

5. Lege Untertyp als Verbund von Obertyp und Erweiterung an:

```
create table Angestellte   (PersNr      integer not null,
                            Name        varchar(30) not null);

create table ProfDaten     (PersNr      integer not null,
                            Rang        character(2),
                            Raum        integer);

create table AssiDaten     (PersNr      integer not null,
                            Fachgebiet  varchar(30),
                            Boss        integer)

create view Profs as
   select a.persnr, a.name, d.rang, d.raum
   from   Angestellte a, ProfDaten d
   where  a.PersNr = d.PersNr;

create view Assis as
   select a.persnr, a.name, d.fachgebiet, d.boss
   from   Angestellte a, AssiDaten d
   where  a.PersNr = d.PersNr;
```

6. Entferne die Tabellen und Sichten wieder:

```
drop table Angestellte;
drop table AssiDaten;
drop table ProfDaten;
drop view  Profs;
drop view  Assis;
```

7. Lege Obertyp als Vereinigung von Untertypen an (zwei der drei Untertypen sind schon vorhanden):

```
create table AndereAngestellte (PersNr      integer not null,
                                Name        varchar(30) not null);

create view Angestellte as
  (select PersNr, Name from Professoren) union
  (select PersNr, Name from Assistenten) union
  (select PersNr, Name from AndereAngestellte);
```

8. Entferne die Tabelle und die Sichten wieder:

```
drop table andereAngestellte;
drop view Angestellte;
```

7.9 SQL-Statements zum Anlegen von Indizes

1. Lege einen Index an für die aufsteigend sortierten Titel der Tabelle Vorlesung:

```
create index titelindex
on Vorlesungen(titel asc);
```

2. Lege einen Index an für die Semesterzahlen der Tabelle Studenten:

```
create index semesterindex
on Studenten(semester asc);
```

3. Entferne die Indizes titelindex und semesterindex:

```
drop index titelindex on Vorlesungen;
drop index semesterindex on Studenten;
```

7.10 Load data infile

Gegeben sei eine tabellenartig strukturierte Textdatei auf dem Rechner des Datenbankservers:

```
4711;Willi;C4;339;1951.03.24
4712;Erika;C3;222;1962.09.18
```

Durch den Befehl `load data infile` kann der Inhalt einer Datei komplett in eine SQL-Server-Tabelle eingefügt werden, wobei das Trennzeichen zwischen Feldern und Zeilen angegeben werden muss:

```
LOAD DATA INFILE '/tmp/prof.txt'
   INTO TABLE Professoren
   FIELDS TERMINATED BY ';';
   LINES TERMINATED BY '\n';
```

7.11 SQL-Scripte

Der MySQL-Server bietet eine prozedurale Erweiterung von SQL an, genannt *SQL-Scripte* oder auch *Stored Procedures*. Hiermit können SQL-Statements zu Prozeduren oder Funktionen zusammengefasst und ihr Ablauf durch Kontrollstrukturen gesteuert werden.

Sei eine Tabelle *konto* mit Kontonummern und Kontoständen angelegt durch

```
create table konto (nr int, stand int);
insert into konto values (1, 100);
insert into konto values (2, 100);
insert into konto values (3, 100);
```

Listing 7.2 zeigt eine benannte Stored Procedure, welche versucht, innerhalb der Tabelle *konto* eine Überweisung durchzuführen und danach das Ergebnis in zwei Tabellen festhält:

```
create table gebucht (datum DATE, nr_1 int, nr_2 int, betrag int);
create table abgelehnt (datum DATE, nr_1 int, nr_2 int, betrag int);


create procedure ueberweisung               -- lege Prozedur an
  (x int,                                   -- Konto-Nr. zum Belasten
   y int,                                   -- Konto-Nr. fuer Gutschrift
   betrag int)                              -- Ueberweisungsbetrag
begin
  set @s = (select stand                    -- lokale Variable mit
              from konto where nr = x);     -- Kontostand von x initialisieren
  if (@s < betrag) then                     -- falls Konto ueberzogen
    insert into abgelehnt                   -- notiere Fehlschlag in
        values (now(), x, y, betrag);       -- der Tabelle 'abgelehnt'
  else
    update konto set stand = stand - betrag -- setze in der Tabelle konto
        where nr = x;                       -- neuen Betrag fuer Kto-Nr. x
    update konto set stand = stand + betrag -- setze in der Tabelle konto
        where nr = y;                       -- neuen Betrag fuer Kto-Nr. y
    insert into gebucht                     -- notiere Buchung in
         values (now(), x, y, betrag);      -- der Tabelle gebucht
  end if;
end $$
```

Listing 7.2 stored procedure ueberweisung

Im Gegensatz zu einem konventionellen Benutzerprogramm wird eine *stored procedure* in der Datenbank gespeichert. Sie wird aufgerufen und (später) wieder entfernt durch

```
call ueberweisung (2, 3, 50);
drop procedure ueberweisung;
```

In Listing 7.3 wird eine Funktion `f2c` definiert, die eine übergebene Zahl als Temperatur in Fahrenheit auffasst und den Wert nach Celsius umrechnet.

```
create function f2c (fahrenheit int)        -- lege Funktion mit Parameter an
  returns int                               -- Rueckgabewert ist ein int
begin
  set @celsius = (fahrenheit - 32)/9.0*5.0; -- berechne Celsius-Wert
  return @celsius;                          -- liefere Wert zurueck
end $$
```

Listing 7.3 stored function f2c

Der Aufruf der Funktion erfolgt innerhalb einer SQL-Abfrage

```
select f2c(122);
```

Oft besteht das Ergebnis eines Select-Statements aus einer variablen Anzahl von Tupeln. Diese können nacheinander verarbeitet werden mit Hilfe eines sogenannten *Cursor*. Listing 7.4 zeigt den typischen Einsatz in einer *stored procedure*.

Zunächst wird der Cursor durch `declare` mit einer SQL-Query assoziiert, welche die Professorennamen mit ihren Vorlesungstiteln ermittelt. Dann wird er mit `open` für die Abarbeitung geöffnet. Mittels `fetch` wird das

nächste Tupel aus der Trefferliste geholt und mit `into` in lokalen Variablen abgespeichert. Ein *continue handler* überwacht, ob noch Tupel aus der SQL-Abfrage abzuarbeiten sind. Zum Abschluss wird der Cursor geschlossen und deallokiert.

```
create procedure berechneVorlesungen() -- speichere pro Professor
begin                                  -- die Zahl seiner Vorlesungen
  declare done int default 0;
  declare prof_name CHAR(16);
  declare prof_cursor cursor for
    select p.name
    from Professoren p, Vorlesungen v
    where p.persnr = v.gelesenvon;
  declare continue handler for sqlstate '02000' set done = 1;
  update Professoren set anzVorlesungen = 0;
  open prof_cursor;
  repeat
    fetch prof_cursor into prof_name;
    if not done then
      update Professoren set anzVorlesungen = anzVorlesungen + 1
      where name = prof_name;
    end if;
  until done end repeat;
  close prof_cursor;
end
```

Listing 7.4 Umgang mit einem Cursor beim Zählen der Veranstaltungen eines Professors

Folgendes Script degradiert jeden Professor auf die Gehaltsstufe C2, falls die Zahl seiner Hörer kleiner ist als 2:

```
CREATE PROCEDURE zuwenigHoerer() -- setze Professor auf Rang C2
begin                            -- falls weniger als 2 Hoerer
  declare done int default 0;
  declare nr, anzahl int;
  declare prof_cursor cursor for
    select gelesenvon, count(*) from Vorlesungen v, hoeren h
    where v.vorlnr = h.vorlnr
    group by gelesenvon;
  declare continue handler for sqlstate '02000' set done = 1;
  open prof_cursor;
  fetch prof_cursor into nr, anzahl;
  while not done do
    if (anzahl <2) then
      update Professoren set rang = 'C2' where persnr = nr;
    end if;
    fetch prof_cursor into nr, anzahl;
  end while;
  close prof_cursor;
end
```

Listing 7.5 Umgang mit einem Cursor beim Zählen seiner Hörer

8. Datenintegrität

8.1 Grundlagen

In diesem Kapitel werden *semantische Integritätsbedingungen* behandelt, also solche, die sich aus den Eigenschaften der modellierten Welt ableiten lassen. Wir unterscheiden zwischen statischen und dynamischen Integritätsbedingungen. Eine statische Bedingung mus von jedem Zustand der Datenbank erfüllt werden (z. B. Professoren haben entweder den Rang C2, C3 oder C4). Eine dynamische Bedingung betrifft eine Zustandsänderung (z. B. Professoren dürfen nur befördert, aber nicht degradiert werden).

Einige Integritätsbedingungen wurden schon behandelt:

- Die Definition des Schlüssels verhindert, das zwei Studenten die gleiche Matrikelnummer haben.
- Die Modellierung der Beziehung *lesen* durch eine 1:N-Beziehung verhindert, dass eine Vorlesung von mehreren Dozenten gehalten wird.
- Durch Angabe einer Domäne für ein Attribut kann z. B. verlangt werden, dass eine Matrikelnummer aus maximal 5 Ziffern besteht (allerdings wird nicht verhindert, daß Matrikelnummern mit Vorlesungsnummern verglichen werden).

8.2 Referentielle Integrität

Seien R und S zwei Relationen mit den Schemata $\mathcal{R}$ und $\mathcal{S}$. Sei κ Primärschlüssel von $\mathcal{R}$.

Dann ist $\alpha \subset \mathcal{S}$ ein Fremdschlüssel, wenn für alle Tupel $s \in S$ gilt:

1. s.α enthält entweder nur Nullwerte oder nur Werte ungleich Null
2. Enthält $s.\alpha$ keine Nullwerte, existiert ein Tupel $r \in R$ mit $s.\alpha = r.\kappa$

Die Erfüllung dieser Eigenschaft heist *referentielle Integrität*. Die Attribute von Primär- und Fremdschlüssel haben jeweils dieselbe Bedeutung und oft auch dieselbe Bezeichnung (falls möglich). Ohne Überprüfung der referentiellen Integrität kann man leicht einen inkonsistenten Zustand der Datenbasis erzeugen, indem z. B. eine Vorlesung mit nichtexistentem Dozenten eingefügt wird. Zur Gewährleistung der referentiellen Integrität mus also beim Einfügen, Löschen und Ändern immer sichergestellt sein, daß gilt

$$\pi_\alpha(S) \subseteq \pi_\kappa(R)$$

Erlaubte Änderungen sind daher:

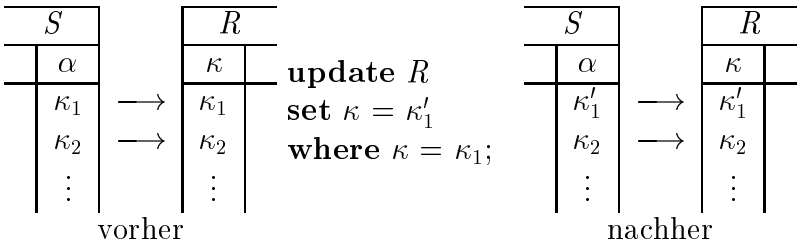
- Einfügen eines Tupels in S verlangt, daß der Fremdschlüssel auf ein existierendes Tupel in R verweist.
- Ändern eines Tupels in S verlangt, daß der neue Fremdschlüssel auf ein existierendes Tupel in R verweist.
- Ändern eines Primärschlüssels in R verlangt, daß kein Tupel aus S auf ihn verwiesen hat.
- Löschen eines Tupels in R verlangt, daß kein Tupel aus S auf ihn verwiesen hat.

8.3 Referentielle Integrität in SQL

SQL bietet folgende Sprachkonstrukte zur Gewährleistung der referentiellen Integrität:

- Ein Schlüsselkandidat wird durch die Angabe von `unique` gekennzeichnet.
- Der Primärschlüssel wird mit `primary key` markiert. Seine Attribute sind automatisch `not null`.
- Ein Fremdschlüssel heist `foreign key`. Seine Attribute können auch `null` sein, falls nicht explizit `not null` verlangt wird.
- Ein `unique foreign key` modelliert eine 1:1 Beziehung.
- Innerhalb der Tabellendefinition von S legt die Klausel α `integer references R` fest, das der Fremdschlüssel α (hier vom Typ Integer) sich auf den Primärschlüssel von Tabelle R bezieht. Ein Löschen von Tupeln aus R wird also zurückgewiesen, solange noch Verweise aus S bestehen.
- Durch die Klausel `on update cascade` werden Veränderungen des Primärschlüssels auf den Fremdschlüssel propagiert (oberer Teil von Abbildung 62).
- Durch die Klausel `on delete cascade` zieht das Löschen eines Tupels in R das Entfernen des auf ihn verweisenden Tupels in S nach sich (unterer Teil der Abbildung 62).
- Durch die Klauseln `on update set null` und `on delete set null` erhalten beim Ändern bzw. Löschen eines Tupels in R die entsprechenden Tupel in S einen Nulleintrag (Abbildung 63).

(a) **create table** $S\,(\ldots,\alpha$ **integer references** R **on update cascade**);



(b) **create table** $S\,(\ldots,\alpha$ **integer references** R **on delete cascade**);

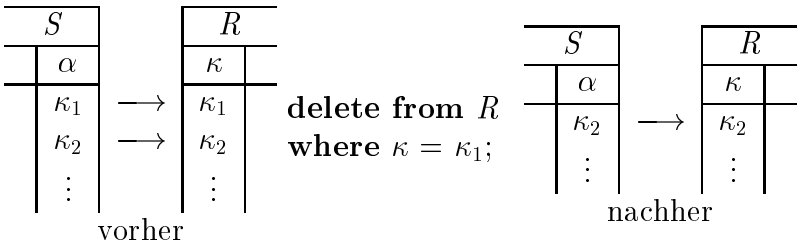
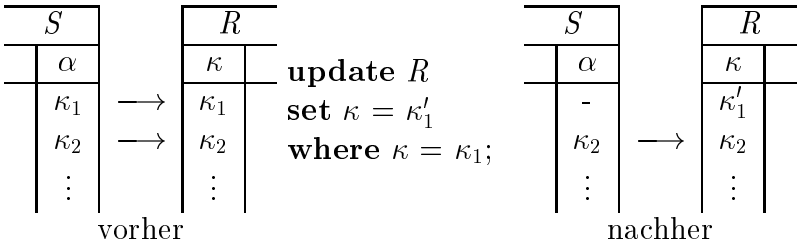


Abb. 62: Referentielle Integrität durch Kaskadieren

a) **create table** $S\,(\ldots,\alpha$ **integer references** R **on update set null**);



(b) **create table** $S\,(\ldots,\alpha$ **integer references** R **on delete set null**);

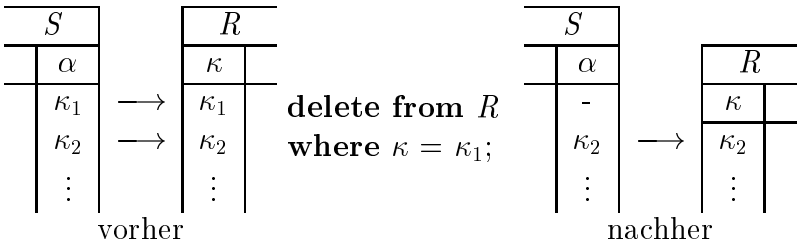
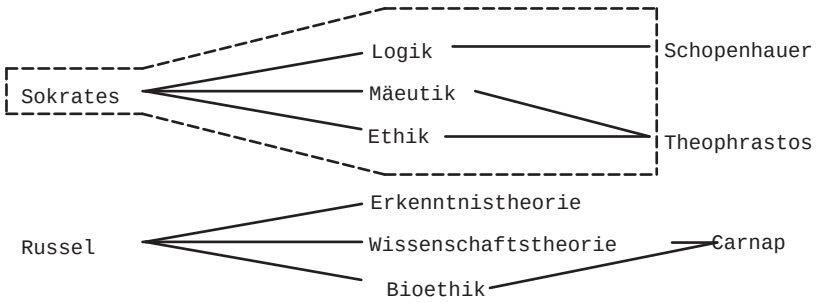


Abb. 63: Referentielle Integrität durch Nullsetzen

Kaskadierendes Löschen kann ggf. eine Kettenreaktion nach sich ziehen. In Abbildung 64 wird durch das Löschen von `Sokrates` der gestrichelte Bereich mit drei Vorlesungen und drei *hören*-Beziehungen entfernt, weil der Fremdschlüssel *gelesenVon* die Tupel in *Professoren* mit `on delete cascade` referenziert und der Fremdschlüssel *VorlNr* in *hören* die Tupel in *Vorlesungen* mit `on delete cascade` referenziert.




```
create table Vorlesung
(...,
gelesenVon integer
            references Professoren
            on delete cascade);
```

```
create table hoeren
   (...,
    VorlNr integer
            references   Vorlesungen
           on delete cascade);
```

Abb. 64: Kaskadierende Löschoperationen

8.4 Statische Integrität in SQL

Durch die *check-Klausel* können einem Attribut Bereichseinschränkungen auferlegt werden.

Zum Beispiel erzwingen

```
... check Semester between 1 and 13 ...
... check Rang in ('C2', 'C3', 'C4') ...
```

gewisse Vorgaben für die Semesterzahl bzw. den Professorenrang. Listing 8.1 zeigt die Formulierung der Uni-Datenbank mit den Klauseln zur Überwachung von statischer und referentieller Integrität. Hierbei wurden folgende Restriktionen verlangt:

- Ein Professor darf solange nicht entfernt oder sein Primärschlüssel geändert werden, wie noch Verweise auf ihn existieren.
- Eine Vorlesung darf solange nicht entfernt werden, wie noch Verweise auf sie existieren. Eine Änderung ihres Primärschlüssels ist erlaubt und zieht das Ändern der Sekundärschlüssel nach sich.
- Ein Student darf entfernt werden und zieht dabei das Entfernen der Zeilen nach sich, die über Sekundärschlüssel auf ihn verweisen. Auch sein Primärschlüssel darf geändert werden und zieht das Ändern der Sekundarschlüssel nach sich.

```
CREATE TABLE Studenten(
  MatrNr         INTEGER PRIMARY KEY,
  Name           VARCHAR(20) NOT NULL,
  Semester       INTEGER,
  GebDatum       DATE
);


CREATE TABLE Professoren(
  PersNr         INTEGER PRIMARY KEY,
  Name           VARCHAR(20) NOT NULL,
  anzVorlesungen INTEGER,
  Rang           CHAR(2) CHECK (Rang in ('C2','C3','C4')),
  Raum           INTEGER UNIQUE,
  Gebdatum       DATE
);


CREATE TABLE Assistenten (
  PersNr         INTEGER PRIMARY KEY,
  Name           VARCHAR(20) NOT NULL,
  Fachgebiet     VARCHAR(20),
  Boss           INTEGER,
  GebDatum       DATE,
  FOREIGN KEY (Boss) REFERENCES Professoren (PersNr)
);


CREATE TABLE Vorlesungen (
  VorlNr         INTEGER PRIMARY KEY,
  Titel          VARCHAR(20),
  SWS            INTEGER,
  gelesenVon     INTEGER,
  FOREIGN KEY (gelesenVon) REFERENCES Professoren (PersNr)
);


CREATE TABLE hoeren (
  MatrNr         INTEGER,
  VorlNr         INTEGER,

  PRIMARY KEY (MatrNr, VorlNr),
```

```
  FOREIGN KEY (MatrNr) REFERENCES Studenten (MatrNr)
      ON UPDATE CASCADE
      ON DELETE CASCADE,
  FOREIGN KEY (VorlNr) REFERENCES Vorlesungen (VorlNr)
      ON UPDATE CASCADE
);


CREATE TABLE voraussetzen (
  Vorgaenger     INTEGER,
  Nachfolger     INTEGER,
  PRIMARY KEY (Vorgaenger, Nachfolger),
  FOREIGN KEY (Vorgaenger) REFERENCES Vorlesungen (VorlNr)
      ON UPDATE CASCADE,
  FOREIGN KEY (Nachfolger) REFERENCES Vorlesungen (VorlNr)
      ON UPDATE CASCADE
);


CREATE TABLE pruefen (
  MatrNr         INTEGER,
  VorlNr         INTEGER,
  PersNr         INTEGER,
  Note           NUMERIC(3,1) CHECK (Note between 0.7 and 5.0),
  PRIMARY KEY (MatrNr, VorlNr),
  FOREIGN KEY (MatrNr) REFERENCES Studenten (MatrNr)
      ON UPDATE CASCADE
      ON DELETE CASCADE,
  FOREIGN KEY (VorlNr) REFERENCES Vorlesungen (VorlNr)
      ON UPDATE CASCADE,
  FOREIGN KEY (PersNr) REFERENCES Professoren (PersNr)
);
```

Listing 8.1 Universitätsschema mit Integritätsbedingungen

8.5 Trigger

Die allgemeinste Konsistenzsicherung geschieht durch einen *Trigger*. Dies ist eine benutzerdefinierte Prozedur, die automatisch bei Erfüllung einer bestimmten Bedingung vom DBMS gestartet wird. Trigger können `BEFORE` oder `AFTER` einer Operation ausgeführt werden; als Operationen sind `INSERT`, `DELETE` oder `UPDATE` möglich. Hilfreich sind zwei vom System gefüllte Variablen `NEW` und `OLD`, in der solche Zeilen gespeichert sind, die neu hinzugekommen bzw. gerade gelöscht wurden.

Listing 8.2 zeigt einen AFTER-INSERT-Trigger für die Tabelle hoeren, der nach jedem Einfügen eines Tupels in der Tabelle `hoeren` aufgerufen wird und dann den Wert `anzhoerer` in der Tabelle `Professoren` bei dem zuständigen Professor um eins hochzählt.

```
create trigger zaehlehoerer
after insert on hoeren
for each row
begin
  update Professoren p
  set p.anzhoerer = p.anzhoerer + 1
  where NEW.vorlnr in
    (select vorlnr from Vorlesungen
     where gelesenvon = p.persnr);
end
```

Listing 8.2 Trigger zum Hochzählen der Hörerzahl

Listing 8.3 zeigt einen AFTER-INSERT-Trigger für die Tabelle `hoeren`, der nach jedem Einfügen eines Tupels feststellt, welcher Professor nun mehr als 2 Hörer hat und diesen dann in den Rang C4 befördert.

```
create trigger befoerderung
after insert on hoeren
```

```
for each row
  update Professoren set rang='C4'
  where persnr in
    (select gelesenvon
     from hoeren h, Vorlesungen v
     where h.vorlnr = v.vorlnr
     group by gelesenvon
     having count(*) > 2)
```

Listing 8.3 Trigger zur Beförderung bei mehr als 2 Hörern

Listing 8.4 zeigt einen AFTER-INSERT-Trigger, der nach dem Einfügen eines Tupels in die Tabelle *Professoren* einen Protokolleintrag in der Tabelle `Protokoll` einfügt.

```
create trigger schreibprotokoll
after insert on Professoren
for each row
  insert into Protokoll (bemerkung)
  values (CONCAT(NEW.persnr, ' wurde eingefuegt'))
```

Listing 8.4 Trigger zum Schreiben eines Protokolleintrags

Sei durch `create table Geburtstagstabelle (name varchar(30), jahre int)` eine Tabelle mit Namen und Angaben zum Lebensalter gegeben. Listing 8.5 zeigt einen AFTER-INSERT-Trigger, der nach dem Einfügen eines Namens mit Lebensalter in die Tabelle `Geburtstagstabelle` daraus das Geburtsjahr errechnet und dieses dann zusammen mit dem Namen in die Tabelle `Personen` einfügt.

```
create trigger Geburtstageintragen
after insert on Geburtstagstabelle
for each row
  insert into Personen (name, gebDatum)
  values(NEW.name, DATE_ADD(NOW(), interval -NEW.jahre year))
```

Listing 8.5 Trigger zum Einfügen eines errechneten Geburtsdatums

Das Schlüsselwort `drop` entfernt einen Trigger: `drop trigger Geburtstag`

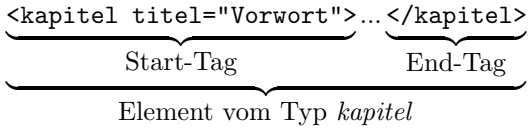
9. XML-Technologien

9.1 XML

Die *Extensible Markup Language* oder kurz *XML* ist eine vom World-Wide-Web-Konsortitum (W3C)[1] herausgegebene Spezifikation[2] zur Beschreibung strukturierter Dokumente. Im Gegensatz zu zahlreichen anderen Formatspezifikationen handelt es sich bei XML um ein Meta-Format, das lediglich den grundsätzlichen formalen Aufbau der Dateien festlegt, aber keine konkreten Strukturen und Bedeutungen vorgibt. Jeder Entwickler kann nun auf Grundlage der XML-Spezifikation konkrete XML-Formate entwerfen und damit festlegen, wie sein Dateiformat genau aussieht, welche Struktur es besitzt und was die einzelnen Bestandteile bedeuten.

Elemente

Zentraler Bestandteil von XML-Dokumenten sind die sogenannten *Elemente*. Sie bestehen aus einem Start- und einem End-Tag sowie dem dazwischenliegenden Elementrumpf. Die Tags sind wie in folgendem Beispiel aufgebaut:



Ein Start-Tag beginnt immer mit einer öffnenden spitzen Klammer (Kleinerzeichen) gefolgt vom *Elementtyp*. Danach können optional beliebig viele *Attribute* folgen, die durch Whitespace (Leerzeichen, Tabulatoren, Newlines, Returns) voneinander getrennt werden. Beendet wird das Start-Tag mit einer schließenden spitzen Klammer (Größerzeichen).

Attribute haben immer die Form *Attributname* = "*Attributwert*". Anstelle der doppelten Anführungszeichen (") dürfen auch einfache (') verwendet werden. Die Reihenfolge, in der die Attribute im Start-Tag angegeben werden, spielt keine Rolle. Allerdings darf jedes Attribut pro Element nur einmal verwendet werden. Folgende Konstruktion ist also nicht erlaubt:

```
<buch autor="Willi Wacker" autor="Wilma Wacker">...</buch>
```

Für Elementtypen und Attributnamen gelten folgende Regeln:

- Bezeichner muss mit einem Buchstaben beginnen
- Bezeichner darf folgende Zeichen enthalten: Buchstaben, Ziffern, Unterstrich (_), Klammeraffe (@), Punkt (.), Minuszeichen (-)
- Groß-/Kleinschreibung ist signifikant, d.h. *kapitel* ist ein anderer Elementtyp als *Kapitel*

Alle End-Tags beginnen ebenfalls mit einer öffnenden spitzen Klammer. Danach folgen ein Slash (/), der Elementtyp und eine schließende spitze Klammer. Der Bereich zwischen Start- und End-Tag (in den vorangehenden Beispielen durch drei Punkte gekennzeichnet) wird *Elementrumpf* oder *Elementinhalt* genannt.

Elementinhalt

Der Elementinhalt kann aus einer Folge von weiteren Elementen und reinem Text bestehen. Beides darf beliebig aneinandergereiht werden. Wichtig ist lediglich, dass die Elemente korrekt geschachtelt werden. Die Start- und End-Tags verschiedener Elemente dürfen sich nicht überlappen:

1 http://www.w3.org
2 http://www.w3.org/XML

falsch

```
<kapitel>
  <titel>Vorwort</titel>
  <absatz>
    Ich gratuliere Ihnen zum Kauf
<fett>dieses
    <kursiv>wunderbaren</fett></kursiv>
Buchs.
  </absatz>
</kapitel>
```

richtig

```
<kapitel>
  <titel>Vorwort</titel>
  <absatz>
    Ich gratuliere Ihnen zum Kauf
<fett>dieses
    <kursiv>wunderbaren</kursiv></fett>
Buchs.
  </absatz>
</kapitel>
```

Wenn ein Element keinen Text und keine weiteren Elemente enthält, der Elementrumpf also leer ist, wird es als *leeres Element* bezeichnet. Da diese Elemente keine klammernden Tags benötigen, sondern ein einzelnes Tag ausreicht, definiert die XML-Spezifikation folgende Kurzschreibweise:

`<uhrzeit zeitzone="GMT"/>` ist gleichbedeutend mit `<uhrzeit zeitzone="GMT"></uhrzeit>`

Wird das Start-Tag also mit `/>` beendet, gilt das Element als komplett und das End-Tag entfällt.

Wurzelelement

Wie gerade gesehen, dürfen Elementrümpfe aus einer beliebigen Folge weiterer Elemente bestehen. Das gilt allerdings nicht für das XML-Dokument selbst, denn jedes XML-Dokument muss ein eindeutiges äußeres Element, das *Wurzelelement* besitzen. Es bildet quasi den Rahmen, in dem sich alle Daten des Dokuments befinden.

falsch

```
<kapitel titel="Einatmen">
...
</kapitel>
<kapitel titel="Ausatmen">
...
</kapitel>
<anhang titel="Schnappatmung für
Fortgeschrittene">
...
</anhang>
```

richtig

```
<buch titel="Selber Atmen">
  <kapitel titel="Einatmen">
  ...
  </kapitel>
  <kapitel titel="Ausatmen">
  ...
  </kapitel>
  <anhang titel="Schnappatmung für
Fortgeschrittene">
  ...
  </anhang>
</buch>
```

Kommentare

Neben Element- und Textbausteinen erlaubt die XML-Spezifikation auch Kommentare, die bei der Verarbeitung von XML-Dateien normalerweise ignoriert werden. Sie können dem Autor dazu dienen, Beschreibungen oder Anmerkungen direkt in das Dokument einzufügen. Für Kommentare gelten folgende Regeln:

- sie werden durch die Zeichenfolge `<!--` eingeleitet
- sie werden durch die Zeichenfolge `-->` beendet
- sie dürfen nicht geschachtelt werden, d.h. die Zeichenfolge `<!--` ist innerhalb eines Kommentars nicht erlaubt.
- sie dürfen sowohl innerhalb von Elementrümpfen als auch vor und hinter dem Wurzelelement stehen

```
<!-- Mein erstes richtiges Buch -->
<buch titel="Selber Atmen">
  <kapitel titel="Einatmen">
  ...
  </kapitel>
```

```
  <kapitel titel="Ausatmen">
  ...
  </kapitel>
  <!-- Jetzt kommen noch ein paar Anhänge für Profis. -->
  <anhang titel="Schnappatmung für Fortgeschrittene">
  ...
  </anhang>
</buch>
<!-- Das wird ein Renner! -->
```

Vordefinierte Entities

Einige Zeichen haben in XML-Dokumenten eine besondere Bedeutung und dürfen deshalb nicht als reguläre Textbestandteile verwendet werden. Das gilt beispielsweise für das Kleinerzeichen, das u.a. zur Einleitung von Tags und Kommentaren benötigt wird. Möchte man diese Spezialzeichen im Text verwenden, muss auf sogenannte *Entities* zurückgegriffen werden. Dabei handelt es sich um Bezeichner, die für ein einzelnes Zeichen oder auch eine Zeichenfolge stehen können.

In XML-Dokumenten sind die folgenden fünf Entities vordefiniert:

Zeichen	Entity
<	<
>	>
&	&
"	"
'	'

Beispiel:

falsch

```
<aufgabe nr="1" gruppe="A & B">
  Vereinfachen Sie die Ungleichung 5x+6
 < 7.
</aufgabe>
```

richtig

```
<aufgabe nr="1" gruppe="A & B">
  Vereinfachen Sie die Ungleichung 5x+6
 < 7.
</aufgabe>
```

Wohlgeformte XML-Dokumente

Ein XML-Dokument wird *wohlgeformt* genannt, wenn es die Vorgaben der XML-Spezifikation einhält. Wohlgeformte XML-Dateien können von einem XML-Parser fehlerfrei eingelesen und in einen Objektbaum überführt werden. Die Wohlgeformtheit sagt nichts über die korrekte Verwendung der zur Verfügung stehenden Elemente aus sondern garantiert lediglich die Fehlerfreiheit der äußeren Form. Das folgende XML-Dokument ist wohlgeformt, obwohl es inhaltlich wenig Sinn macht:

```
<personen>
  <person>
    <kapitel titel="Einatmen">
    </kapitel>
    <uhrzeit/>
  </person>
  <kursiv><bild url="http://www.uni-osnabrueck.de/images/LogoGelbLinks.gif"/></kursiv>
</personen>
```

9.2 DTD und Schema

(Text in Arbeit)

9.3 XPath

Zur Navigation in XML-Bäumen und zur gezielten Auswahl von XML-Knoten hat das W3C die Lokatorsprache *XPath* entwickelt. Für fast alle Programmiersprachen gibt es Bibliotheken, die es dem Programmierer erlauben, mit Hilfe von XPath-Ausdrücken Teile der bearbeiteten XML-Dokumente auszuwählen und anschließend zu verarbeiten. Während die Verwendung von XPath hierbei eher optionalen Charakter hat, spielt sie im Rahmen von XQuery und XSLT eine zentrale Rolle, denn XPath gehört dort zum elementaren Bestandteil der Sprachen.

Dieses Kapitel gibt einen Überblick über Syntax und Semantik von XPath-Ausdrücken.

Knotentests

Auf den ersten Blick erinnern einfache XPath-Ausdrücke an Pfandangaben zur Navigation im UNIX-Dateisystem, außer dass anstelle der Verzeichnis- und Dateinamen die Typen von Elementknoten angegeben werden. Tatsächlich wurde die Syntax an UNIX-Pfadangaben angelehnt und umfangreich erweitert.

Eine Pfadangabe besteht in XPath aus einer beliebigen Folge so genannter *Knotentests*, die durch Slashes voneinander getrennt werden. Bei jedem dieser Tests wird geprüft, ob der XML-Baum an der angegebenen Position einen oder mehrere Knoten mit der gewünschten Eigenschaft besitzt oder nicht. Das Ergebnis eines solchen Tests ist immer eine geordnete Menge (Sequenz) aller passenden Knoten. Werden keine passenden Knoten gefunden, ist die Ergebnissequenz leer.

Eine **absolute Pfadangabe** beginnt immer bei der Dokumentwurzel. Da die Wurzel des aktuellen XML-Dokuments durch einen Slash (/) adressiert wird, beginnen absolute XPath-Ausdrücke, die sich auf das aktuelle Dokument beziehen, logischerweise immer mit einem Slash. Anschließend folgen die Knotentests, die im einfachsten Fall schrittweise die Eigenschaften der Kindknoten prüfen. Abhängig vom Knotentyp stellt XPath nun verschiedene Tests bereit.

Elementknoten werden durch Angabe ihres Typs ausgewählt. Abbildung 65 zeigt eine absolute Pfadangabe, bei der in drei Schritten von der Dokumentwurzel zum zweiten B-Element navigiert wird. Der aus drei Knotentests bestehende XPath-Ausdruck `/A/B/B` wird dabei folgendermaßen schrittweise ausgewertet:

1. liefere die Dokumentwurzel /
2. suche im Dokumentknoten ein Element vom Typ A und liefere es als Ergebnis zurück
3. suche im gefundenen A-Knoten ein Element vom Typ B und liefere es als Ergebnis zurück
4. suche im gefundenen B-Knoten ein Element vom Typ B und liefere es als Ergebnis zurück

Das Ergebnis ist also ein Elementknoten vom Typ B. Falls nur ein einziger Knotentest in einer solchen Pfadangabe fehlschlägt, ist das Ergebnis die leere Menge.

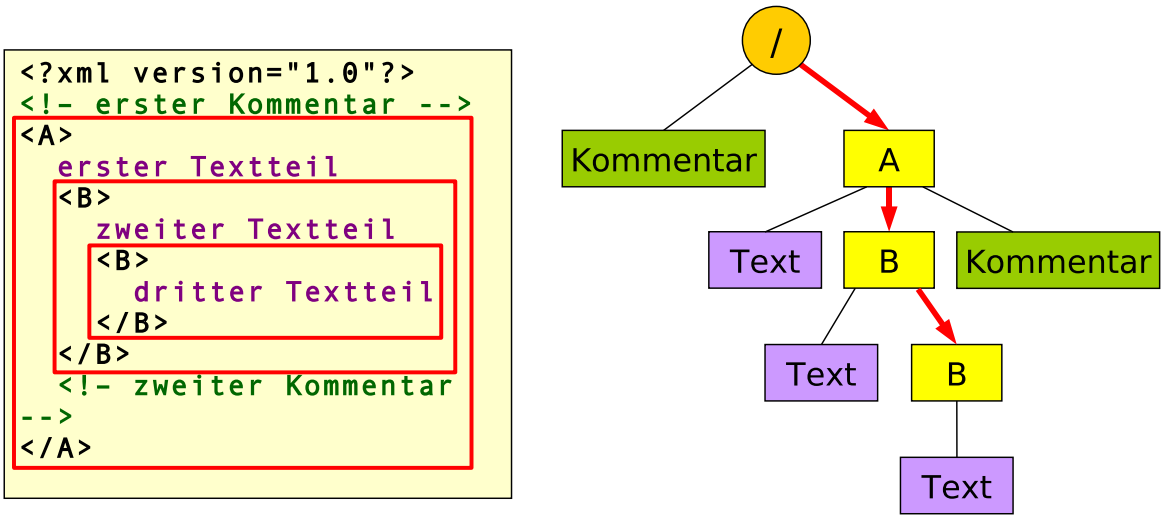


Abb. 65: Schrittweise Navigation zum Knoten <code class='escaped'>/A/B/B</code>

Möchte man nicht im aktuellen sondern in einem anderen XML-Dokument nach Knoten suchen, kann dies durch Verwendung des Knotentests `doc(...)` anstelle des führenden Slashs erreicht werden. So sucht der Ausdruck `doc('knoten.xml')/A` in der Datei knoten.xml nach dem Wurzelelement A und liefert es im Erfolgsfall zurück.

Die folgende Tabelle zeigt die Syntax der Knotentests für alle zentralen Knotentypen:

Knotentyp	**Syntax des Knotentests**
aktuelles Dokument	/
anderes Dokument	doc("*URI*")
Element	*elementtyp*
Attribut	*@attributname*
Text	text()
Kommentar	comment()
alle Knotentypen	node()

Zur Veranschaulichung der Knotentests `text()` und `comment()` hier noch ein paar Beispiele:

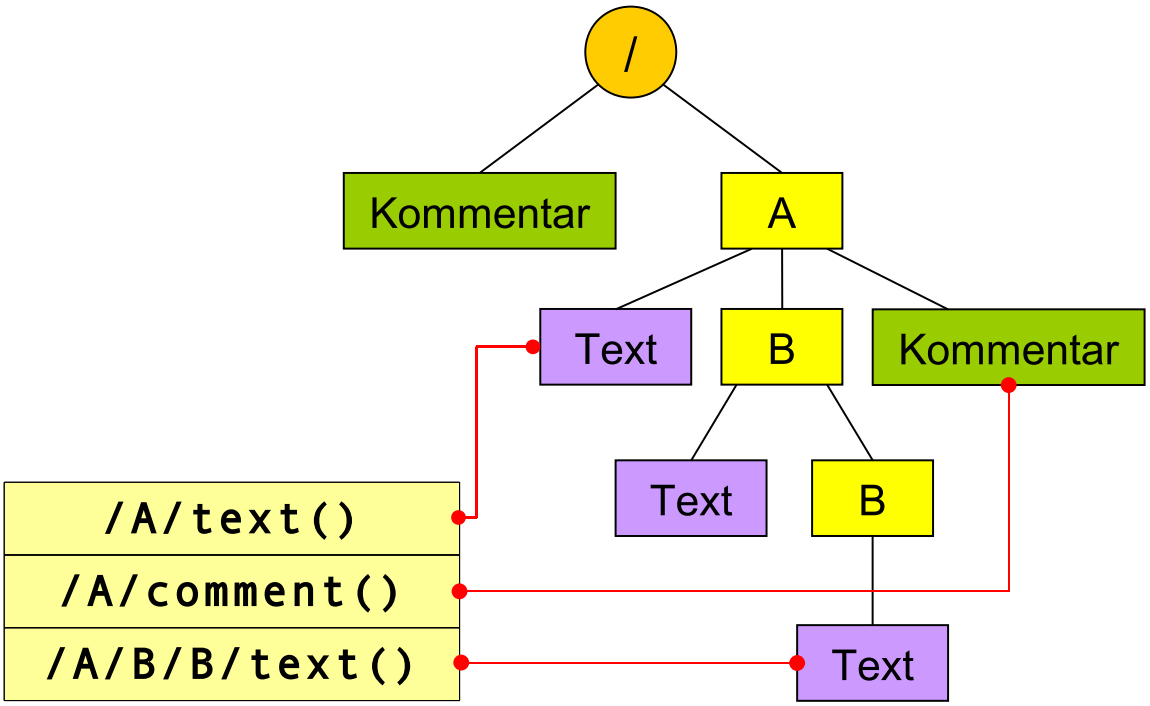


Abb. 66: Navigation zu Text- und Kommentarknoten

Knotensequenzen

In den vorangehenden Beispielen liefert jeder Knotentest einen eindeutigen Knoten zurück. Dies ist in der Praxis aber eher die Ausnahme als die Regel, denn oft enthalten XML-Elemente mehr als nur einen Knoten vom selben Typ. Passen auf einen Knotentest mehrere Knoten, so werden sie alle zurückgeliefert und zwar in der Reihenfolge, in der sie im XML-Dokument auftauchen. Eine solche geordnete Menge von Knoten wird als *Knotensequenz* bezeichnet. Der Ausdruck `/A/B` in Abbildung 67 liefert somit beide B-Kindelemente vom Wurzelelement A. Wird auf diese Sequenz anschließend der Knotentest `text()` angewendet, erhält man alle Textknoten, die in den durch `/A/B` adressierten B-Elementen enthalten sind.

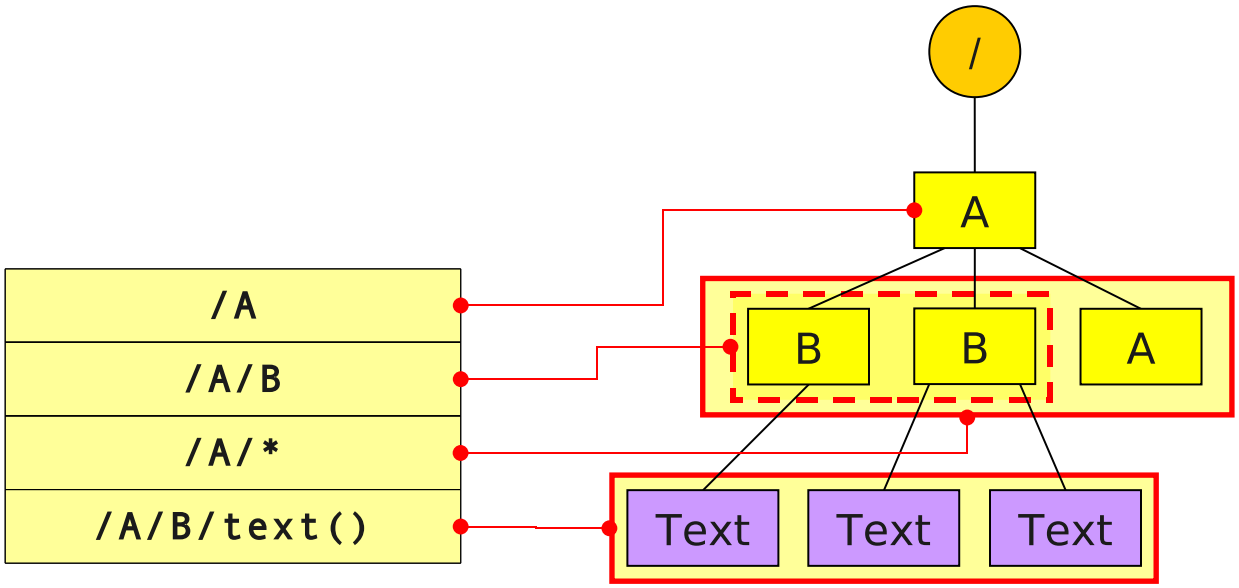


Abb. 67: Auswahl von Knotensequenzen

In einigen Fällen kann es erforderlich sein, nicht nur Elemente eines bestimmten Typs sondern alle in einem Knoten enthaltenen Elemente oder Attribute auszuwählen. Dafür stellt XPath die beiden Knotentests `*` und `@*`

zur Verfügung. Angewendet auf das folgende XML-Dokument, liefert der Ausdruck `/veranstaltungen/*/titel/text()` die beiden Veranstaltungstitel. `/veranstaltungen/seminar/@*` selektiert die beiden Attribute des Seminar-Elements und mit `/veranstaltungen/*/@*` erhält man alle vier Attribute.

```
<veranstaltungen>
  <!-- Hauptstudiumsveranstaltungen -->
  <vorlesung nr="1.234" max-teilnehmer="150">
    <titel>Einführung in die manuelle Addition natürlicher Zahlen</titel>
    <dozent>
      <vname>Manfred</vname>
      <nname>Müller</nname>
    </dozent>
  </vorlesung>
  <seminar nr="1.235" max-teilnehmer="20">
    <titel>Topologien leerer Unterräume</titel>
    <dozent>
      <titel>Prof. Dr.</titel>
      <vname>Maria</vname>
      <nname>Meier</nname>
    </dozent>
  </seminar>
</veranstaltungen>
```

Achsen

Wenn nicht anders angegeben, bezieht sich ein Knotentest immer auf die Kinder der aktuell adressierten Knoten. `/A/B/C` bedeutet: "Suche im Elementknoten A nach B-Elementen, danach in den B-Elementen nach C-Elementen und liefere letztere als Resultat zurück." Mit jedem weiteren rechts angefügten Knotentest steigt man also eine Ebene weiter abwärts. Dementsprechend sind solche Knotentests relativ kurzsichtig, da der Blick nur in Richtung der unmittelbaren Kindknoten geht.

XPath bietet nun eine Erweiterung der Knotentests an, mit denen die Blickrichtung geändert werden kann, denn manchmal ist es notwendig, nicht die Kinder eines Elements auszuwählen, sondern z.B. die Geschwister oder einen der Vorfahren. Diese Blickrichtungsänderung wird mit Hilfe der 12 so genannten *Achsen* realisiert. Abbildung 68 gibt einen Überblick über die von den verschiedenen Achsen jeweils erreichbaren Knoten.

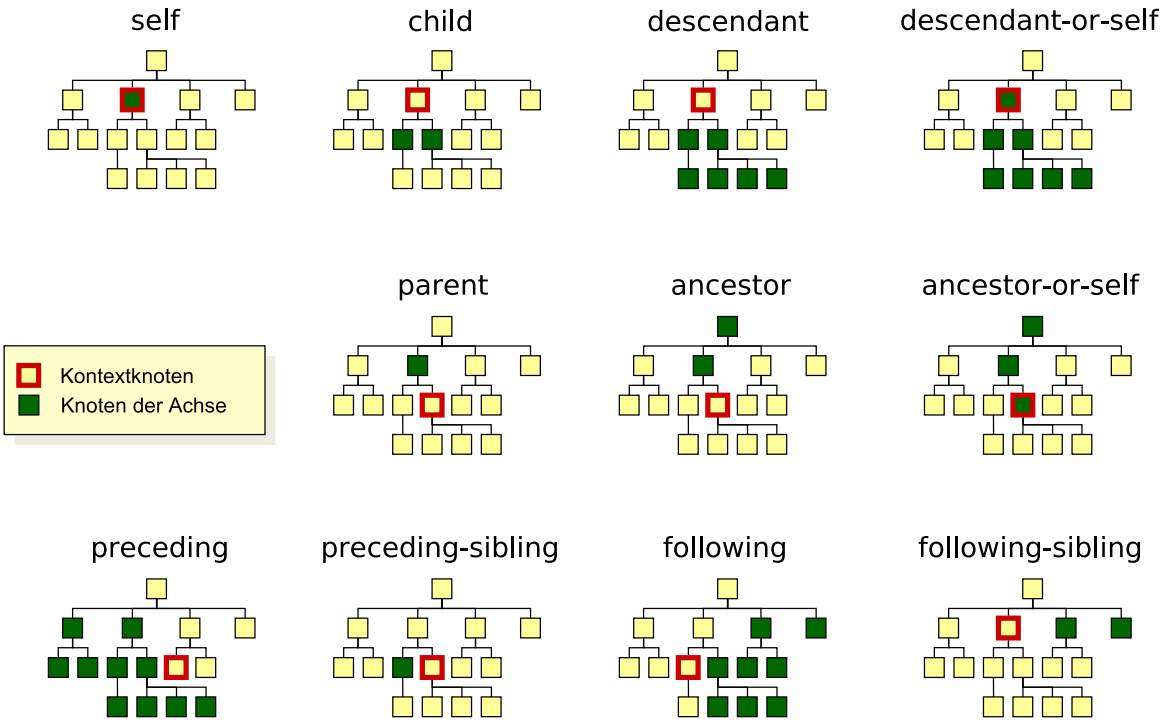


Abb. 68: Die verfügbaren Achsen von XPath

Jeder Knotentest kann durch einen vorangestellten Achsennamen gefolgt von zwei Doppelpunkten erweitert werden. Dadurch bezieht sich der Test dann auf die von der Achse erreichbaren Knoten. Wird beispielsweise statt `/A` der Ausdruck `/descendant::A` verwendet, dann wird nicht nur der A-Wurzelknoten sondern sämtliche A-Elemente, die sich irgendewo im XML-Baum befinden, zurückgeliefert, denn wie Abbildung 68 zu entnehmen ist, erweitert die descendant-Achse den Suchbereich auf alle Nachfahren. Auch hier entspricht die Reihenfolge, in der die Knoten in der Ergebnissequenz angeordnet werden, der Dokumentreihenfolge.

Ein weiteres Beispiel: Angewendet auf das vorangehende XML-Dokument erhält man durch Angabe von `/*/vorlesung/following-sibling::*/titel` das *titel*-Element mit Inhalt "Topologien leerer Unterräume", denn der Knotentest `following-sibling::*` setzt die Suche nicht bei den Kindern des bis dahin ausgewählten *vorlesung*-Elements sondern bei den nachfolgenden/rechten Geschwistern fort. In diesem Fall ist das *seminar*-Element das einzige nachfolgende/rechte Geschwisterelement.

Einige Achsen werden in der Praxis besonders häufig benötigt. Aus diesem Grund definiert die XPath-Spezifikation din in folgender Tabelle auflisteten Kurzschreibweisen für ausgewählte Knotentests.

Kurzform	**Langform**
A	child::A
@A	attribute::A
.	self::node()
..	parent::node()
//	/descendant-or-self::node()/

Der XPath-Ausdruck `//A/B/@C` ist also eine Kurzform für `/descendant-or-self::node()/child::A/child::B/attribute::C`.

Relative XPath-Ausdrücke und Kontextknoten

XPath-Ausdrücke, die mit einem Slash beginnen, starten immer bei der Wurzel des aktuellen XML-Dokuments und werden deshalb als absolute Pfadangaben bezeichnet. Weitaus wichtiger sind in der Praxis allerdings

relative Ausdrücke, die sich auf einen zuvor ausgewählten Knoten im XML-Baum, den so genannten *Kontextknoten*, beziehen. Besonders ausgiebig wird davon in der Transformationssprache XSLT, aber auch bei den unten beschriebenen XPath-Prädikaten Gebrauch gemacht. Abbildung 69 zeigt ein paar Beispiele für relative XPath-Ausdrücke.

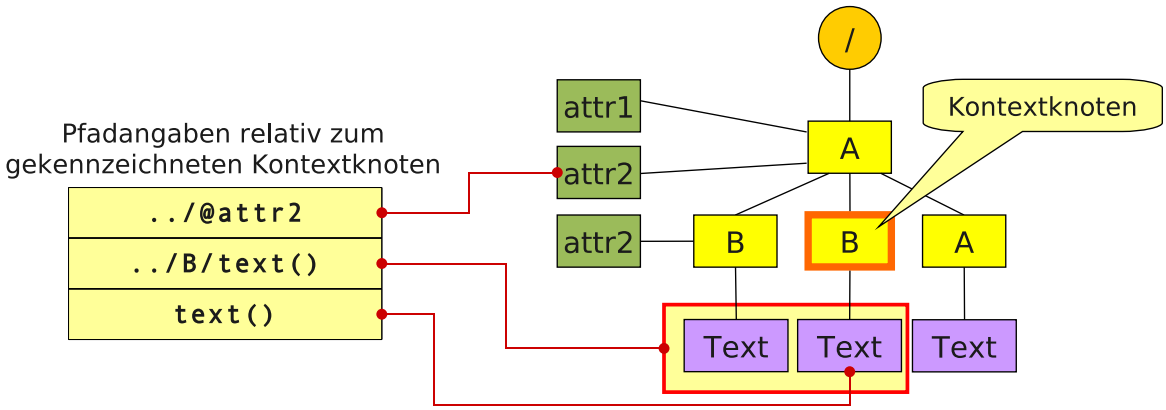


Abb. 69: Relative XPath-Ausdrücke beziehen sich auf einen Kontextknoten

Wie der Kontextknoten ausgewählt wird und wo genau er sich im Baum befindet, spielt dabei zunächst keine Rolle. Entscheidend ist lediglich, dass Klarheit darüber herrscht, von welchem Typ der Kontextknoten ist oder in welchem Knotenumfeld er sich befindet.

Zur Veranschaulichung betrachten wir noch einmal das oben abgebildete XML-Dokument mit dem "Vorlesungsverzeichnis". Angenommen, wir wissen, dass durch irgendeine vorangehende Aktion ein *dozent*-Element ausgewählt wurde, der Kontextknoten also ein *dozent*-Element ist. Dann liefert der relative Ausdruck `../titel` das *titel*-Element der Veranstaltung, in der sich der Kontextknoten befindet: Ausgehend von einem beliebigen *dozent*-Element in obigem XML-Dokument selektiert `..` den Elternknoten (*vorlesung* oder *seminar*), und der angehängte Knotentest `titel` wählt das darin erhaltenen *titel*-Elemet aus.

Der schlichte Ausdruck `nname` liefert ausgehend von demselben Kontextknoten das Element mit dem Nachnamen des Dozenten.

Prädikate

Mit den bisher vorgestellten Knotentests lassen sich die verschiedenen Knoten nur relativ grob auswählen. Beispielsweise ist es bisher nicht möglich, gezielt ein Element oder ein Attribut mit bestimmtem Inhalt zu adressieren. Vielleicht möchte man im obigen XML-Dokument nach den Veranstaltungen eines bestimmten Dozenten suchen oder alle Veranstaltungen mit mehr als 50 maximalen Teilnehmern ausgeben. Solche Verfeinerungen der Knotentests lassen sich mit Hilfe von *Prädikaten* erreichen. Dabei handelt es sich um boolsche Ausdrücke, die in eckigen Klammern hinter einem Knotentest angegeben werden können. Alle Knoten, die die Bedingung nicht erfüllen, werden aus der Ergebnissequenz entfernt.

Betrachten wir noch einmal das vorangehende XML-Dokument und wenden darauf den XPath-Ausdruck `/*/*[@max-teilnehmer > 20]` an. Der XPath-Prozessor führt dabei der Reihe nach folgende Schritte aus:

1. wähle den Dokumentknoten aus
2. suche im Dokumentknoten nach dem Wurzelelement (egal von welchem Typ es ist) und liefere es zurück
3. suche im Wurzelelement nach beliebigen Elementknoten und liefere sie zurück (liefert das *vorlesung*- und *seminar*-Element)
4. suche in den ausgewählten Elementen (*vorlesung* und *seminar*) jeweils nach dem Attribut *max-teilnehmer*; liefere die Elemente zurück, die ein solches Attribut besitzen und dessen Wert größer als 20 ist

Durch die Einschränkung `@max-teilnehmer > 20` wird nur die Vorlesung von Manfred Müller ausgewählt, da das *seminar*-Element die angegebene Eigenschaft nicht erfüllt.

Zur Formulierung von Prädikaten erlaubt XPath zahlreiche vordefinierte Operatoren und Funktionen. Neben den Vergleichsoperatoren =, !=, <, >, <= und >= sind auch die mathematischen bzw. logischen Operationen +, -, *mul*, *div*, *mod*, *or* und *and* erlaubt. Zum Testen von String-Eigenschaften gibt es ebenfalls einige Funktionen. Hier eine kleine Auswahl:

Funktion	**Ergebnistyp**	**Bedeutung**
concat(s1,...,sn)	string	verknüpft die Strings s1 bis sn zu einem neuen String
contains(s1, s2)	bool	prüft, ob String s2 in String s1 enthalten ist
normalize-space(s)	string	entfernt Whitespace vom Anfang und Ende des Strings und ersetzt alle anderen Whitespacefolgen durch einfache Leerzeichen
string-length(s)	number	liefert die Länge von String s
substring(s, n, l)	string	liefert Teilstring der Länge l von s beginnend bei Zeichen n (Nummerierung beginnt bei 1)
substring-before(s1, s2)	string	sucht in String s1 nach String s2 und liefert alle Zeichen, die sich vor dem ersten Treffer befinden
substring-after(s1, s2)	string	sucht in String s1 nach String s2 und liefert alle Zeichen, die sich hinter dem ersten Treffer befinden

Als Parameter erwarten diese Funktionen größtenteils Strings. Diese können entweder als Konstanten in einfachen oder doppelte Anführungszeichen angegeben oder auch aus Knoten des XML-Dokuments abgeleitet werden. Falls erforderlich, führt der XPath-Prozessor dabei automatisch Typumwandlungen durch. So können statt Strings auch Knotentests oder ganze XPath-Ausdrücke angegeben werden, wie z.B. `string-length(/A/B)`. In diesen Fällen wird der Textinhalt des adressierten Knotens verwendet. Zu beachten ist dabei, dass die XPath-Ausdrücke eindeutige Knoten beschreiben müssen. Knotensequenzen mit mehr als einem Knoten produzieren hier einen Fehler.

Prädikate müssen nicht zwingend aus expliziten boolschen Ausdrücken bestehen, sondern können u.a. auch durch Strings und XPath-Ausdrücke gebildet werden. Strings der Länge 0 werden dabei als *false*, alle anderen als *true* interpretiert. Ganz ähnlich verhält es sich bei XPath-Ausdrücken und den dadurch beschriebenen Knotensequenzen: leere Sequenzen gelten als *false*, alle anderen als *true*.

Relative Pfadangaben beziehen sich innerhalb von Prädikaten immer auf die durch den vorangehenden Teilausdruck selektierten Knoten. Beispielsweise prüft `/A/B[C]`, ob die ausgewählten B-Elemente mindestens ein C-Element enthalten. `/A/B` stellt also den Kontext für den Knotentest `C` dar.

Knotentests in Prädikaten lassen sich ebenfalls durch Prädikate weiter eingrenzen. Der Ausdruck `/A/B[C[@D]]` z.B. wählt alle B-Kindelemente vom Wurzelelement A aus, die ein C-Element mit D-Attribut enthalten.

Knotennummerierung und Positionstests

Eine besondere Bedeutung haben Prädikate, die nur aus einer ganzen Zahl bestehen. Sie werden automatisch zu einem Positionstest erweitert. So ist der Ausdruck `A[1]` eine Kurzschreibweise für `A[position() = 1]`. Er prüft offensichtlich, ob ein ausgewähltes A-Element die Position 1 besitzt. Doch was für eine Position liefert die Funktion *position*?

Wie bereits beschrieben, ist das Ergebnis eines Knotentests immer eine Knotensequenz, also eine geordnete Menge von Knoten. Die Knoten besitzen innerhalb der Sequenz eine eindeutige Position, die über einen Index angesprochen werden kann. Abbildung 70 zeigt die Nummerierung der A-Elemete beim Knotentest `A` relativ zum markierten Kontextknoten. Die Ergebnissequenz enthält somit vier Knoten, die bei 1 beginnend aufsteigend nummeriert werden. Die Nummer eines Knotens in der Ergebnissequenz kann mit der Funktion *position* abgefragt werden. Der Ausdruck `A[1]` liefert in diesem Beispiel also das erste A-Kindelement des Kontextknotens.

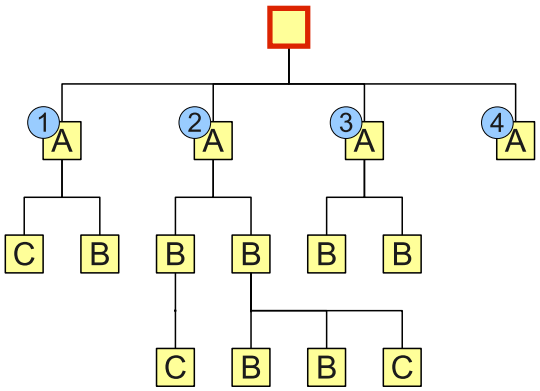


Abb. 70: Nummerierung der A-Knoten bei Auswertung des Ausdrucks <code class='escaped'>A</code> relativ zum rot markierten Kontextknoten

Ganz wichtig ist dabei zu beachten, dass sich die Nummerierung immer nur auf die gerade betrachtete Achse. Im obigen Beispiel wurde durch Weglassen der Achsenangabe implizit die *child*-Achse verwendet. Da der Kontextknoten vier passende Kindelemente enthält, wird von 1 bis 4 durchnummeriert.

Was passiert nun, wenn ein Knotentest Kindknoten von unterschiedlichen Eltern zurückliefert, wie z.B. vom Ausdruck `A/B`? In diesem Fall beginnt die Nummerierung der Kinder für jeden Elternknoten bei 1. Der Ausdruck `A/B[1]` ist also mehrdeutig und liefert eine Sequenz bestehend aus drei Knoten zurück (vgl. Abb. 71)

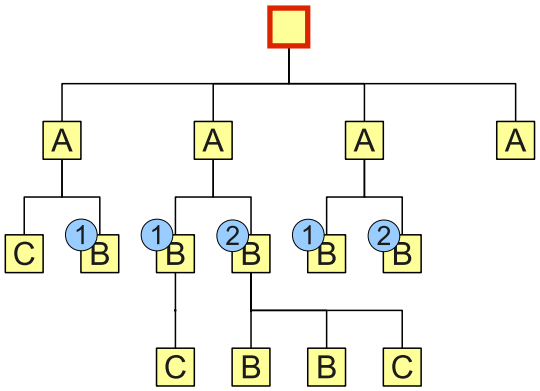


Abb. 71: Nummerierung der B-Knoten bei Auswertung des Ausdrucks <code class='escaped'>A/B</code> relativ zum rot markierten Kontextknoten

Um gezielt einen der B-Knoten über Positionsangaben auszuwählen, kann z.B. ein konkreter A-Knoten herausgefischt werden. Mit dem Ausdruck `A[3]/B[1]` erhält man den ersten B-Kindknoten des dritten A-Elements. Dasselbe Ergebnis lässt sich auch mit dem Ausdruck `(A/B[1])[3]` produzieren. Das angehängte Prädikat `[3]` extrahiert aus der von `A/B[1]` gelieferten Knotensequenz das dritte Element.

Die Zuweisung einer Positionsnummer ist also von der gewählten Achse, dem Kontextknoten und dem durchgeführten Knotentest abhängig. Wird einer dieser Parameter geändert, ändert sich auch die Nummerierung der ermittelten Knoten. Im Zusammenhang mit den verschiedenen Achsen ist zu beachten, dass die Nummerierung der Knoten nicht zwangsläufig der Dokumentreihenfolge entspricht: bei allen Achsen, die in Richtung Dokumentanfang orientiert sind (*ancestor*, *ancestor-or-self*, *parent*, *preceding* und *preceding-sibling*) wächst die Positionsnummer in umgekehrter Dokumentreihenfolge. Im Falle der ancestor-Achse bekommt also der Vaterknoten die Nummer 1, der Großvaterknoten die Nummer 2 usw. (vgl. Abb. 72). Als Faustregel kann man sich merken, dass die Nummern immer mit zunehmender Entfernung vom aktuellen Knoten wachsen.

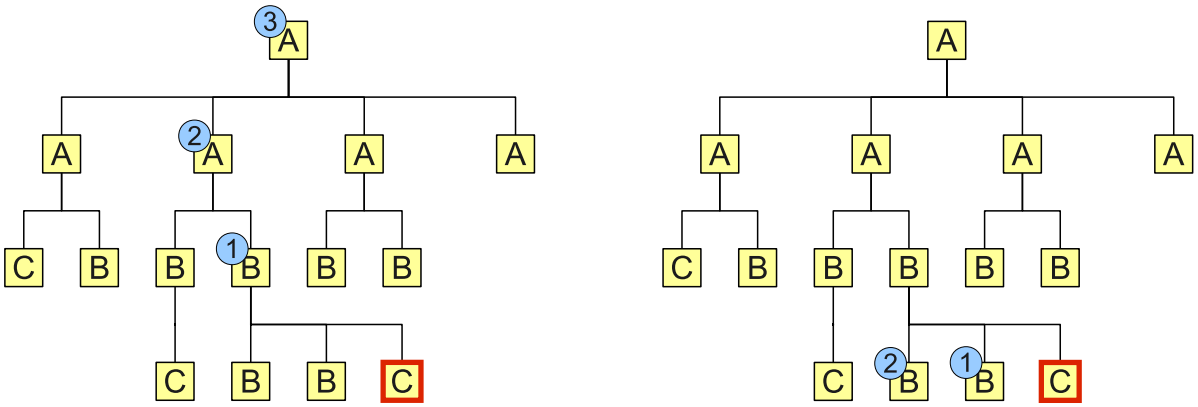


Abb. 72: Nummerierung der Knoten bei Auswertung des Ausdrucks <code class='escaped'>ancestor::*</code> (links) bzw. <code class='escaped'>preceding-sibling::*</code> (rechts).

9.4 XQuery

Bei XQuery[3] handelt es sich um eine vom W3C spezifizierte deklarative Programmiersprache, die auf das Extrahieren von Daten aus XML-Daten spezialisiert ist. Im Gegensatz zu SQL ist XQuery eine reine Abfragesprache. Sie enthält standardmäßig keine Anweisungen zum Ändern von XML-Dokumenten. Erst die optionale, sich noch in der Entwicklung befindende Spracherweiterung *XQuery Update Facility*[4] ermöglicht eine Modifikation der Daten. Nicht alle XQuery-Prozessoren stellen diese Erweiterung bereit, so dass dies bei der Wahl der Entwicklungswerkzeuge berücksichtigt werden muss.

XQuery 1.0 enthält XPath 2.0 als Untermenge, d.h. alle XPath-Ausdrücke sind auch gültige XQuery-Ausdrücke. Darüber hinaus erweitert XQuery den Sprachumfang u.a. um Variablen- und Funktionsdefinitionen, um Konstrukte zum Iterieren über Sequenzen sowie um Methoden zum Erzeugen neuer Elemente.

XQuery wird aus Performance-Gründen in der Regel in Verbindung mit XML-Datenbanken verwendet, kann aber auch direkt auf XML-Dateien angewendet werden. Dafür stehen stehen mittlerweile zahlreiche kommerzielle und kostenlose Tools zur Verfügung. Hier eine kleine Auswahl verschiedener XML-Datenbanken und XQuery-Tools aus dem Open-Source-Bereich:

- Saxon[5] (XQuery- und XSLT-Prozessor)
- Zorba[6] (XQuery-Prozessor, inkl. update facility)
- eXist[7] (XML-Datenbank, inkl. update facility)
- BaseX[8] (XML-Datenbank mit GUI)
- XQilla[9] (XQuery-Prozessor, inkl. update facility)

Sequenzen

Sequenzen sind uns schon im Zusammenhang mit XPath begegnet: Immer wenn ein XPath-Ausdruck mehr als einen passenden Knoten findet, wird eine Knotensequenz zurückgeliefert. Dabei handelt es sich um eine geordnete Menge von Knoten. In XQuery müssen Sequenzen nicht zwingend aus Knoten bestehen, sondern können auch Zahlen, Strings und/oder boolsche Werte enthalten. Sequenzen können u.a. indirekt als Ergebnis eines XPath-Audrucks aber auch durch explizit Auflisten aller enthaltenen Bestandteile erzeugt werden:

3 http://www.w3.org/TR/xquery
4 http://www.w3.org/TR/2009/CR-xquery-update-10-20090609
5 http://saxon.sourceforge.net
6 http://www.zorba-xquery.com
7 http://exist.sourceforge.net
8 http://www.inf.uni-konstanz.de/dbis/basex
9 http://xqilla.sourceforge.net/HomePage

```
(1, 2, 3, 4, 'Hallo', 'Welt')
```

Geschachtelte Sequenzen werden immer automatisch aufgelöst, so dass eine einzige eindimensionale Sequenz entsteht. Bildlich gesprochen bedeutet das, dass alle inneren Klammern entfernt werden. In XQuery gibt es somit keine geschachtelten Sequenzen:

`(1, (2, (3, 4)), ('Hallo', 'Welt'))` wird zu `(1, 2, 3, 4, 'Hallo', 'Welt')`

Zum schnellen Erzeugen ganzzahliger Sequenzen stellt XQuery den Operator *to* zur Verfügung:

`10 to 20` ist eine Kurzform für `(10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)`

FLWOR-Ausdrücke

Zu den zentralen Bestandteilen von XQuery gehören die sogenannten FLWOR-Ausdrücke. Das wie engl. *flower* ausgesprochene Akronym steht für die fünf Sprachbestandteile *for*, *let*, *where*, *order by* und *return*. Neben der Kleinschreibung ist hierbei die Reihenfolge der Schlüsselwörter zu beachten. Zwar muss ein FLWOR-Ausdruck nicht zwingend aus allen fünf Anweisungen bestehen, die verwendeten Anweisungen müssen aber in der genannten Reihenfolge angeordnet werden, also *for* vor *let*, *let* vor *order by* usw.

Variablendefinitionen mit *let*

Die Anweisung *let* wird zur Definition von Variablen verwendet. Variablen beginnen in XQuery immer mit einem Dollarzeichen ($), gefolgt von dem Variablennamen. Für Variablennamen gelten die gleichen Regeln wie für XML-Elementtypen (müssen mit einem Buchstaben beginnen, danach können Kombinationen aus Buchstaben, Ziffern und den Zeichen _, -, @ folgen).

Eine Variablendefinition via *let* hat die Form

`let <variable> := <wert>`, also z.B. `let $n := 5`.

Im Gegensatz zu imperativen Programmiersprachen, wie C++ oder Java gibt es in XQuery keinen Zuweisungsoperator. Variablen können also nachträglich nicht mehr verändert werden. Zwar sind Konstruktionen der Form

```
let $n := 5
let $n := 10
```

erlaubt, sie bewirken aber eine Neudefinition und keine Aktualisierung der Variablen. Insbesondere wird vom Standard nicht garantiert, dass iterative Ausdrücke, wie z.B.

```
let $n := 5
let $n := $n + 1
```

wie in imperativen Sprachen ausgewertet werden. Solche Konstruktionen sind unbedingt zu vermeiden. Variablen müssen in XQuery vielmehr wie Variablen in mathematischen Termen oder Gleichungen behandelt werden. Sie sind Platzhalter für jeweils einen Wert, der innerhalb des Ausdrucks "stabil" bleibt. Eine einmal definierte Variable steht also im ganzen Ausdruck für denselben Wert, so wie *x* und *y* an jeder Stelle des folgenden Gleichungssystems für jeweils dieselbe Zahl stehen. Es gibt insbesondere keine *Seiteneffekte*, die eine der Variablen um 1 vergrößern.

$$\begin{aligned} x^2 &= 10 - y^2 \\ xy &= y + 2x \end{aligned}$$

Anders als die vorangehenden Beispiele vielleicht zunächst vermuten lassen, können Variablen nicht nur einfache Objekte, wie Zahlen und Strings sondern auch Ergebnisse von XPath- oder ganzen XQuery-Ausdrücken

aufnehmen. Davon wird z.B. häufig Gebrauch gemacht, wenn lange oder rechenintensive Ausdrücke mehrfach benötigt werden. Da Variablen normalerweise mit konstanter Laufzeit ausgelesen werden, können sie bei sinnvollem Gebrauch neben der Lesbarkeit auch die Ausführgeschwindigkeit des Ausdrucks erhöhen.

Eine Variablendefinition könnte also z.B. folgendermaßen aussehen:

```
let $hauptstaedte := //land[substring(name,1,1)='A' and hauptstadt/einwohnerzahl
> 5000000]/hauptstadt
```

Resultate zurückliefern mit *return*

Jeder XQuery-Ausdruck muss ein Ergebnis, nämlich das Resultat der Anfrage produzieren. Eine isolierte Variablendefinition, wie beispielsweise die vorangehende, ist somit nicht erlaubt, da sie im Gegensatz zu einem isolierten XPath-Ausdruck kein Ergebnis zurückliefert. Aus diesem Grund muss jeder FLWOR-Ausdruck mit einem 'return'-Statement abschließen, durch den das gewünschte Ergebnis des Ausdrucks festgelegt wird.

Um die obige *let*-Anweisung zu einem gültigen FLWOR-Ausdruck zu vervollständigen, muss also ein *return*-Statement hinzugefügt werden:

```
let $hauptstaedte := //land[substring(name,1,1)='A' and hauptstadt/einwohnerzahl >
5000000]/hauptstadt
return $hauptstaedte
```

Dies ist gleichbedeutend mit

```
let $laender := //land[substring(name,1,1)='A' and hauptstadt/einwohnerzahl > 5000000]
return $laender/hauptstadt
```

und ebenfalls mit

```
let $laender := //land
let $spezielle_laender := $laender[substring(name,1,1)='A' and hauptstadt/einwohnerzahl > 5000000]
return $spezielle_laender/hauptstadt
```

Iterationen mit *for*

Die *for*-Anweisung wird immer dann benötigt, wenn die Komponenten einer Sequenz manipuliert, transformiert oder sortiert werden müssen. Zum besseren Verständnis betrachten wir zunächst das folgende XML-Dokument:

```
<personen>
   <person geschlecht="w">
      <vname>Wilma</vname>
      <nname>Wacker</nname>
   </person>
   <person geschlecht="w">
      <vname>Claire</vname>
      <nname>Grube</nname>
   </person>
   <person geschlecht="m">
      <vname>Willi</vname>
      <nname>Wacker</nname>
   </person>
   <person geschlecht="w">
      <vname>Anna</vname>
      <nname>Conda</nname>
```

```
   </person>
   <person geschlecht="m">
      <vname>Jim</vname>
      <nname>Panse</nname>
   </person>
   <person geschlecht="m">
      <vname>Bernhard</vname>
      <nname>Diener</nname>
   </person>
</personen>
```

Gesucht ist ein XQuery-Ausdruck, der die Namen aller Personen in der Form *Nachname, Vorname* untereinander ausgibt. Mit einem XPath-Ausdruck allein ist das nicht zu bewerkstelligen, da mit dem XPath-Vokabular zwar gleichartige Knoten selektiert, diese aber nicht weiter transformiert werden können. Abhilfe schafft hierbei die *for*-Anweisung, die in ihrer einfachsten Gestalt eine Iterationsvariable und eine Sequenz erwartet:

FLWOR-Ausdruck

```
for $p in //person
return concat($p/nname, ', ',
$p/vname, '
')
```

Resultat

```
Wacker, Wilma
Grube, Claire
Wacker, Willi
Conda, Anna
Panse, Jim
Diener, Bernhard
```

Der XPath-Ausdruck `//person` erzeugt eine Sequenz mit allen *person*-Elementen in Dokumentreihenfolge. Die Iterationsvariable $p nimmt der Reihe nach alle Komponenten der Sequenz an, so dass damit den nachfolgenden Anweisungen auf die *person*-Elemente zugegriffen werden kann. Das *return*-Statement

```
concat($p/nname, ', ', $p/vname, '
')
```

liefert für jedes *person*-Element einen String zurück, der sich aus dem Nachnamen, einem Komma, dem Vornamen sowie einem abschließenden "newline"-Zeichen (= Zeichen mit ASCII-Code 10) zusammensetzt. All diese Strings werden zu einer Sequenz zusammenfasst und als Resultat des FLWOR-Ausdrucks zurückgeliefert. Da die *return*-Anweisung den Abschluss des FLWOR-Ausdrucks bildet, verliert die Iterationsvariable $p dort automatisch ihre Gültigkeit, d.h. hinter dem *return* kann nicht mehr auf $p zugegriffen werden.

Jeder *for*-Anweisung kann zusätzlich zur Iterationsvariablen eine *Positionsvariable* hinzugefügt werden. Ihr wird bei jedem Iterationsschritt automatisch die ganzzahlige Position zugewiesen, an der sich das aktuelle Element innerhalb der Sequenz befindet. Da in der XML-Welt alle Nummerierungen bei 1 und nicht bei 0 beginnen, befindet sich auch hier das erste Sequenzelement auf Position 1.

FLWOR-Ausdruck

```
for $p at $i in //person
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
1: Wacker, Wilma
2: Grube, Claire
3: Wacker, Willi
4: Conda, Anna
5: Panse, Jim
6: Diener, Bernhard
```

Achtung: Eine Positionsvariable ist keine Laufvariablen, deren Wert bei jedem Iterationsschritt automatisch um 1 erhöht wird. Auch wenn sich die Reihenfolge, in der über die Sequenz iteriert wird, ändert, bleibt die Position der Sequenzelemente immer gleich. Näheres dazu in den folgenden Abschnitten.

Filtern mit *where*

Möchte man die Auswahl der Elemente, über die iteriert werden soll, noch etwas verfeinern, kann dies entweder mit Prädikaten im XPath-Ausdruck oder mit einer *where*-Anweisung im FLWOR-Ausdruck realisiert werden. Beide Varianten filtern Komponenten der Sequenz anhand boolscher Ausdrücke heraus -- allerdings zu unterschiedlichen Zeitpunkten.

Wie im XPath-Kapitel beschrieben, entfernt der XPath- oder XQuery-Prozessor alle nicht-passende Knoten aus der Ergebnissequenz, sprich: die Ergebnissequenz enthält genau die Knoten, die auf den XPath-Ausdruck inklusive aller Prädikate passen.

Bei Verwendung einer *where*-Klausel wird die aktuelle Sequenz hingegen nicht verändert, sondern alle nicht-passenden Sequenzkomponenten bei der Iteration übersprungen. In Verbindung mit Positionsvariablen ist dies ein wichtiger Unterschied, da sich die Nummerierung bei beiden Varianten unterscheidet, wie auch die beiden folgenden Beispiele zeigen:

FLWOR-Ausdruck

```
for $p at $i in //person[ @geschlecht = 'w']
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
1: Wacker, Wilma
2: Grube, Claire
3: Conda, Anna
```

Hier werden alle weiblichen Personen durch Angabe eines Prädikats im XPath-Ausdruck herausgefiltert. Der FLWOR-Ausdruck iteriert somit über drei *person*-Elemente, die von 1 bis 3 durchnummeriert sind. Wird statt eines Prädikats eine *where*-Klausel verwendet, iteriert der Ausdruck über alle sechs, von 1 bis 6 durchnummerierten *person*-Elemente. Das Ergebnis weicht deshalb von der ersten Variante ab:

FLWOR-Ausdruck

```
for $p at $i in //person
where $p/@geschlecht = 'w'
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
1: Wacker, Wilma
2: Grube, Claire
4: Conda, Anna
```

Sortieren mit *order by*

Mit Hilfe der Anweisung *order by* lassen sich die Komponenten der Sequenz, über die iteriert wird, sortieren. Genauer gesagt, ändert die Anweisung die Iterationsreihenfolge. Die Position der Sequenzbestandteile bleibt weiterhin unverändert.

Falls vorhanden, muss die Anweisung *order by* hinter der *where*-Klausel angegeben werden. Die Anweisung selbst erwartet mindestens ein Sortierkriterium, wie in folgendem Beispiel den Nachnamen der Person:

FLWOR-Ausdruck

```
for $p at $i in //person
order by $p/nname
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
4: Conda, Anna
6: Diener, Bernhard
2: Grube, Claire
5: Panse, Jim
1: Wacker, Wilma
3: Wacker, Willi
```

Um die beiden letzten Namen in die richtige Reihenfolge zu bringen, muss die Sortierung bei identischem Nachnamen auf den Vorname ausgedehnt werden. Dies lässt sich durch Angabe eines zweiten Sortierkriteriums hinter dem ersten erreichen:

FLWOR-Ausdruck

```
for $p at $i in //person
order by $p/nname, $p/vname
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
4: Conda, Anna
6: Diener, Bernhard
2: Grube, Claire
5: Panse, Jim
3: Wacker, Willi
1: Wacker, Wilma
```

Falls nicht anders angegeben, sorgt das *order by*-Statement, wie in den Beispielen gesehen, für eine aufsteigende Sortierung. Durch Angabe des Modifizierers *descending* kann die Sortierreihenfolge für jedes Sortierkriterium getrennt umgekehrt werden:

FLWOR-Ausdruck

```
for $p at $i in //person
order by $p/nname descending, $p/vname
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
3: Wacker, Willi
1: Wacker, Wilma
5: Panse, Jim
2: Grube, Claire
6: Diener, Bernhard
4: Conda, Anna
```

FLWOR-Resultate weiterverarbeiten

Abschließend soll noch kurz erwähnt werden, dass die Ergebnisse eines FLWOR-Ausdrucks nicht sofort als Gesamtresultat des XQuery-Ausdrucks zurückgegeben werden müssen, sondern zur Weiterverarbeitung auch in Variablen abgelegt werden können. Dazu wird der gesamte FLWOR-Ausdruck einfach als Wert einer übergeordneten *let*-Anweisung aufgefasst:

XQuery-Ausdruck

```
let $personen :=
  for $p in //person
  order by $p/nname descending, $p/vname
descending
  return $p
for $p at $i in $personen
return concat($i, ': ', $p/nname, ', ',
$p/vname, '
')
```

Resultat

```
1: Wacker, Wilma
2: Wacker, Willi
3: Panse, Jim
4: Grube, Claire
5: Diener, Bernhard
6: Conda, Anna
```

In diesem Beispiel sorgt der eingerückte FLWOR-Ausdruck dafür, dass die *person*-Elemente absteigend sortiert werden. Die Ergebnissequenz wird in der Variable *$personen* abgelegt. Der anschließend folgende FLWOR-Ausdruck iteriert über die bereits sortierte Sequenz und gibt so ein aufsteigend nummeriertes Ergebnis zurück.

Element-Konstruktoren

Von einer auf XML spezialisierten Programmiersprache wird nicht nur erwartet, dass mit ihr XML-Dokumente eingelesen und verarbeitet, sondern darüber hinaus auch erzeugt werden können. Dabei geht es nicht um das Erstellen neuer XML-Dateien, sondern um die textuelle Ausgabe von XML-Schnipseln oder ganzer XML-Dokumente.

In den vorangehenden Beispielen wurden mit XQuery-Ausdrücken verschiedene Stringsequenzen erzeugt. Wenn vom *return*-Statement statt Strings ganze Elemente zurückgegeben werden, werden diese immer inklusive Tags und Inhalt ausgegeben. Der XQuery-Prozessor sorgt also automatisch dafür, dass Elementknoten in die korrekte Textdarstellung überführt werden.

Oft ist es aber nötig, neue Elemente zu erzeugen oder den Typ vorhandener Elemente auszutauschen. Für diese Fälle gibt es in XQuery die sogenannten *Element-Konstruktoren*, die im einfachsten Fall aus eingestreuten Start- und End-Tags bestehen, die korrekt geschachtelt werden müssen. Befinden sich zwischen Start- und End-Tag XQuery-Anweisungen, die ausgewertet werden sollen, müssen diese wie im folgenden Beispiel durch geschweifte Klammern umschlossen werden:

XQuery-Ausdruck

```
<vornamen>{
  for $v in
//vname
```

Resultat

```
<vornamen>

<vname>Anna</vname>
```

```
  order by $v
  return $v
}
</vornamen>
```

```
<vname>Bernhard</vname>

<vname>Claire</vname>
  <vname>Jim</vname>

<vname>Willi</vname>

<vname>Wilma</vname>
</vornamen>
```

Zum Extrahieren des Textinhalts eines Elements oder eines Attributs stellt XQuery die Funktion *data* zur Verfügung. Sie entfernt, bildlich gesprochen, die Tags bzw. Attributkennzeichner vom Argument und liefert die reinen Textinhalte als String zurück. In Verbindung mit Element-Konstruktoren können auf diese Weise Elementtypen ausgetauscht werden. Zum Abschluss dieses Kapitels dazu noch ein etwas umfangreicheres Beispiel, das ein HTML-Dokument mit einer Tabelle generiert:

XQuery-Ausdruck

```
<html>
<head><title>Personentabelle</title></head>
<body>
<table cellpadding="5">{
let $farben := ('red', 'lightblue', 'lightgreen',
'yellow')
for $p at $i in //person
let $farbe1 := $farben[(($i - 1) mod 4)+1]
let $farbe2 := $farben[4-(($i - 1) mod 4)]
order by $p/nname, $p/vname
return
  <tr>
    <td bgcolor="{$farbe1}">{data($p/vname)}</td>
    <td bgcolor="{$farbe2}">{data($p/nname)}</td>
  </tr>
}
</table>
</body>
</html>
```

Resultat

Anna	Conda
Bernhard	Diener
Claire	Grube
Jim	Panse
Willi	Wacker
Wilma	Wacker

9.5 XSLT

(Text in Arbeit)

10. Datenbankapplikationen

10.1 ODBC

Open Database Connectivity (ODBC) ist eine von Microsoft entwickelte standardisierte Anwendungs-Programmierungs-Schnittstelle (Application Programming Interface, API), über die Anwendungen auf Datenbanken unterschiedlicher Hersteller zugreifen können (Oracle, Informix, Microsoft, MySQL, ...). Hierzu wird auf dem Rechner des Anwenders unter Verwendung des für den jeweiligen Hersteller vorgeschriebenen Treibers eine sogenannte Datenquelle installiert, auf die dann das Anwendungsprogramm zugreifen kann, ohne die proprietären Netzwerkkommandos zu beherrschen.

Abbildung 73 zeigt das Hinzufügen einer ODBC-Datenquelle in der Systemsteuerung von Windows.

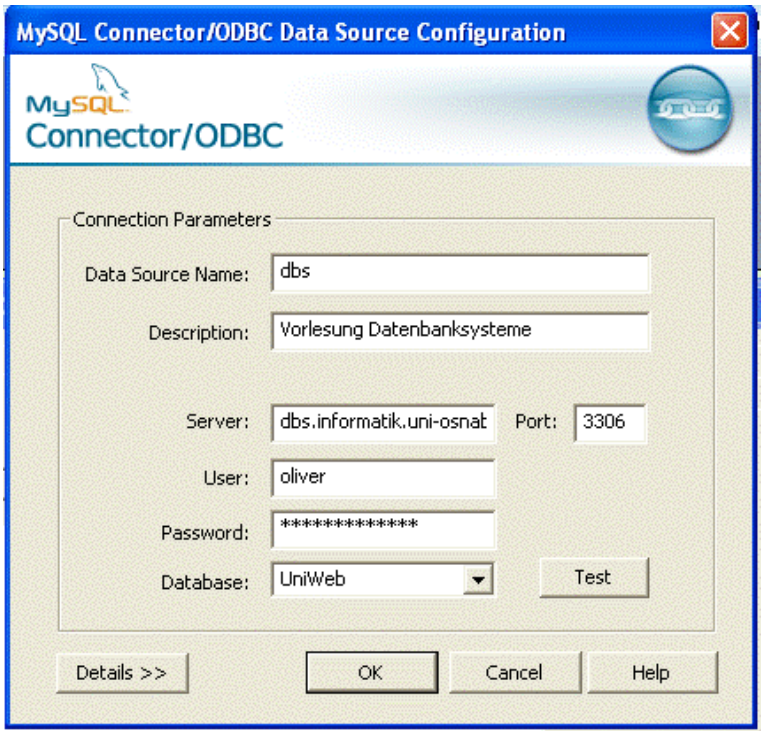


Abb. 73: Hinzufügen einer ODBC-Datenquelle

10.2 Microsoft Visio

Microsoft Visio ist ein Vektorgrafikprogramm zum Erstellen von Grafiken mit vorgefertigten Schablonen. Über den Menupunkt *Database Reverse Engineering* kann das Datenbankschema einer ODBC-fähigen Datenbank eingelesen und grafisch aufbereitet werden (Abbildung 74).

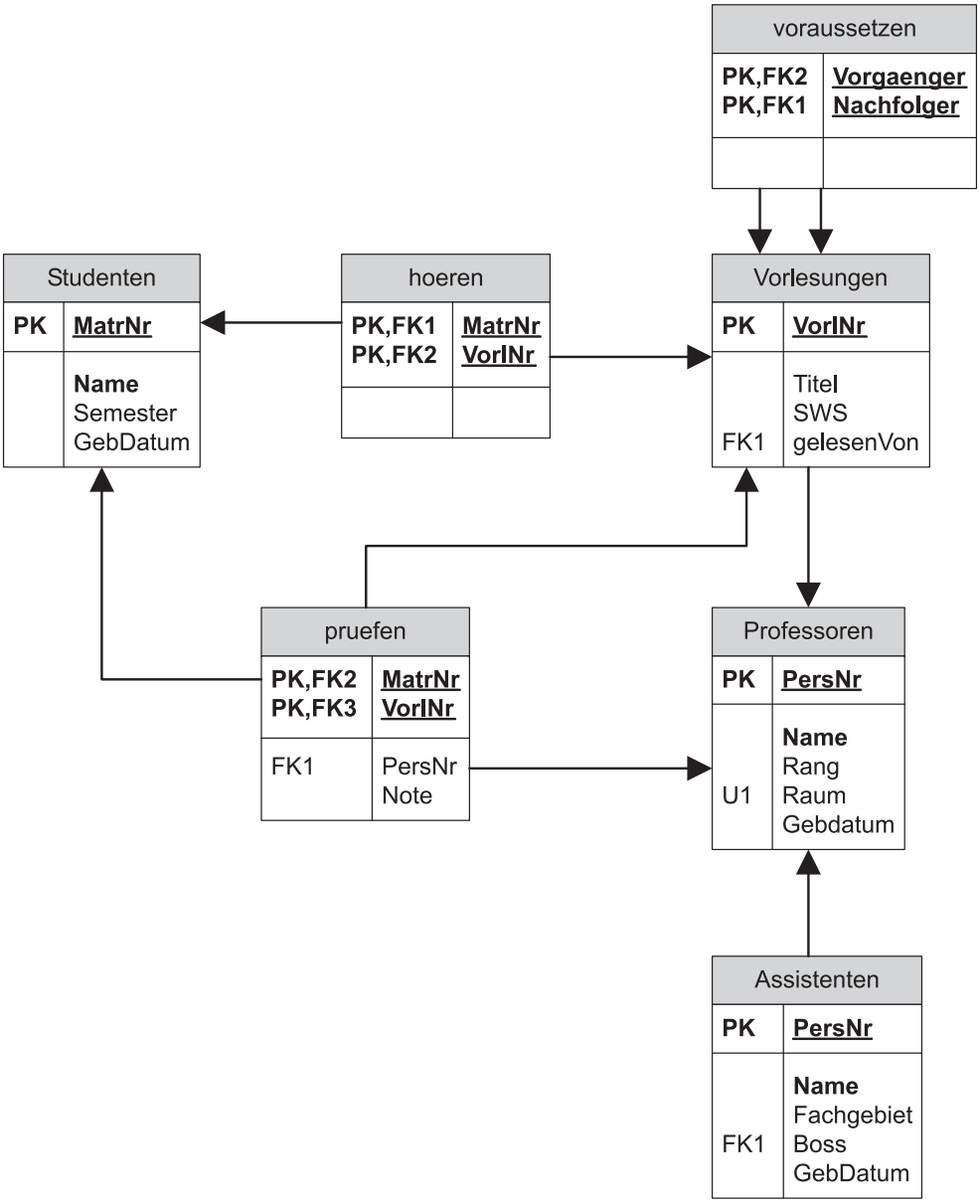


Abb. 74: Universitätsschema, erzeugt von MS Visio

10.3 Microsoft Access

```
CODE CODE
```

Access ist ein relationales Datenbanksystem der Firma *Microsoft*, welches als Einzel- und Mehrplatzsystem unter dem Betriebssystem Windows läuft. Seine wichtigsten Eigenschaften:

- Als **Frontend** eignet es sich für relationale Datenbanksysteme, die per ODBC-Schnittstelle angesprochen werden können. Hierzu wird in der Windows-Systemsteuerung der zum Microsoft-SQL-Server mitgelieferte ODBC (Open Data Base Connectivity) Treiber eingerichtet und eine User Data Source hinzugefügt, z.B. mit dem Namen `dbs`. Nun können die auf dem SQL-Server liegenden Tabellen verknüpft und manipuliert werden.
- Der **Schemaentwurf** geschieht menugesteuert (Abbildung 75)
- Referenzen zwischen den Tabellen werden in Form von **Beziehungen** visualisiert.
- **Queries** können per SQL oder menugesteuert abgesetzt werden (Abbildung 76).

- **Berichte** fassen Tabelleninhalte und Query-Antworten in formatierter Form zusammen und können als Rich-Text-Format exportiert werden (Abbildung 77).
- **Balken- und Tortendiagramme** lassen sich automatisch aus den Trefferlisten der Abfragen erzeugen (Abbildung 78)
- **Formulare** definieren Eingabemasken, die das Erfassen und Updaten von Tabellendaten vereinfachen (Abbildung 79).
- **Visual-Basic-for-Application-Skripte** [?] ermöglichen die Gestaltung individueller Abfrageformulare (Abbildung 80).

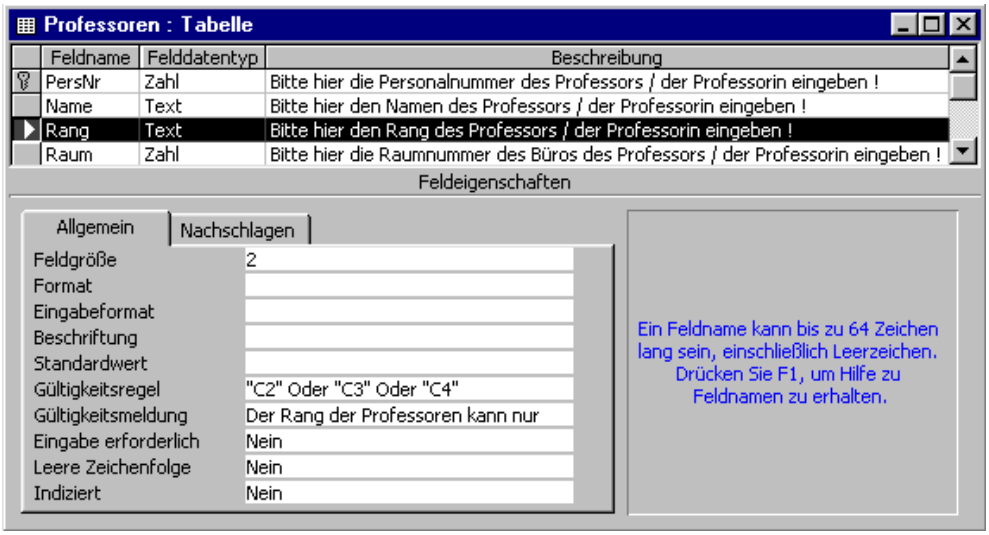


Abb. 75: Schemadefinition in Microsoft Access

Abbildung 76 zeigt die interaktiv formulierte Abfrage, die zu jedem Professor seine Studenten ermittelt. Daraus wird die SQL-Syntax erzeugt, gelistet in [?]. Aus den Treffern dieser Query wird der Bericht in Abbildung 77 generiert.

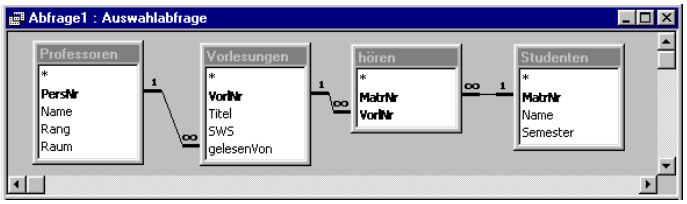


Abb. 76: in Microsoft Access formulierte Abfrage

```
SELECT DISTINCT p.name AS Professor, s.name AS Student
FROM  professoren p, vorlesungen v, hoeren h, studenten s
WHERE v.gelesenvon = p.persnr
and   h.vorlnr     = v.vorlnr
and   h.matrnr     = s.matrnr
ORDER BY p.name, s.name
```

Abb. 77: Word-Ausgabe eines Microsoft Access-Berichts

```
SELECT p.Name, sum(v.SWS) AS Lehrbelastung
FROM   Vorlesungen AS v, Professoren AS p
WHERE  v.gelesenVon=p.PersNr
GROUP  BY p.name;
```

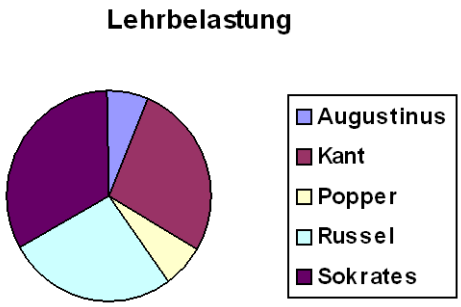


Abb. 78: Tortendiagramm, erstellt aus den Treffern zur SQL-Abfrage

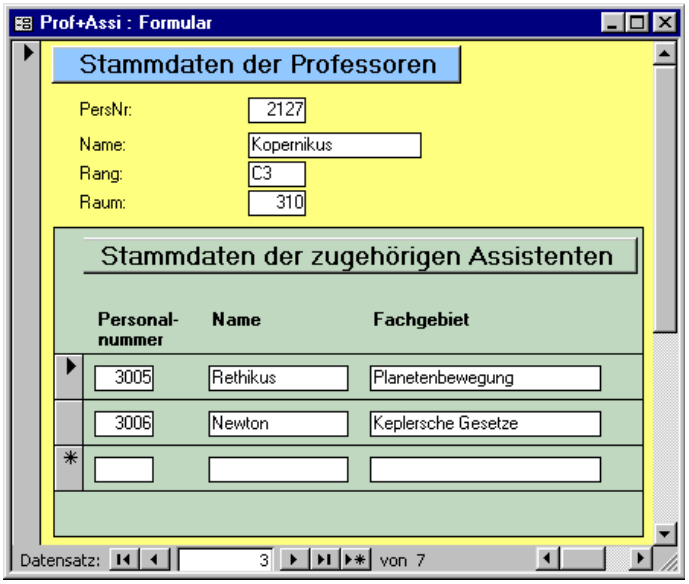


Abb. 79: interaktiv zusammengestelltes Eingabeformular

```
Private Sub Befehl8_Click()

    Dim rang As String
```

```
    Select Case gehaltsgruppe.Value
    Case 1
        rang = "C2"
    Case 2
        rang = "C3"
    Case 3
        rang = "C4"
    Case Else
        rang = " "
    End Select

    If rang = " " Then
        MsgBox ("Sie haben keinen Rang angegeben")
    Else
        Dim rs As Recordset
        Set rs = CurrentDb.OpenRecordset("Select name, gebdatum " & _
        "from professoren where gebdatum = " & _
        "(select min(gebdatum) from professoren where rang = '" & rang & "')")
        ausgabe.Value = rs.Fields("name").Value & _
        ", geboren am " & rs.Fields("gebdatum")
    End If

End Sub
```

Abb. 80: Formular mit VBA-Skript-Funktionalität

10.4 Embedded SQL

Unter *Embedded SQL* versteht man die Einbettung von SQL-Befehlen innerhalb eines Anwendungsprogramms. Das Anwendungsprogramm heist *Host Programm*, die in ihm verwendete Sprache heist *Host-Sprache*.

Der Microsoft SQL-Server unterstützt Embedded SQL im Zusammenspiel mit den Programmiersprachen C und C++ durch einen Pre-Compiler. Abbildung 81 zeigt den prinzipiellen Ablauf: Das mit eingebetteten SQL-Statements formulierte Host-Programm beispiel.sqc wird zunächst durch den Pre-Compiler unter Verwendung von SQL-spezifischen Include-Dateien in ein ANSI-C-Programm beispiel.c überführt. Diese Datei übersetzt ein konventioneller C-Compiler unter Verwendung der üblichen C-Include-Files in ein Objekt-File beispiel.o. Unter Verwendung der Runtime Library wird daraus das ausführbare Programm beispiel.exe gebunden.

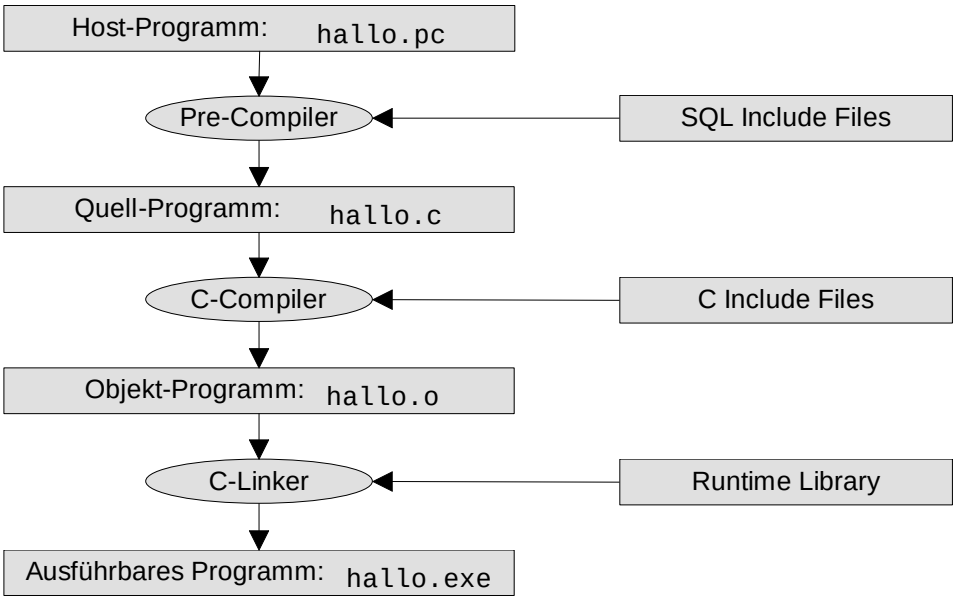


Abb. 81: Datenfluss für Embedded SQL beim Microsoft SQL-Server

Eingebettete SQL-Statements werden durch ein vorangestelltes EXEC SQL gekennzeichnet und ähneln ansonsten ihrem interaktiven Gegenstück. Die Kommunikation zwischen dem Host-Programm und der Datenbank geschieht über sogenannte Host-Variablen, die im C-Programm deklariert werden. Eine Input-Host-Variable überträgt Daten des Hostprogramms an die Datenbank, eine Output-Host-Variable überträgt Datenbankwerte und Statusinformationen an das Host-Programm. Hostvariablen werden innerhalb von SQL-Statements durch einen vorangestellten Doppelpunkt (:) gekennzeichnet.

Für Hostvariablen, die Datenbankattributen vom Typ VARCHAR entsprechen, empfiehlt sich eine Definition nach folgendem Beispiel:

```
char fachgebiet[20];
```

Mit einer Hostvariable kann eine optionale Indikator-Variable assoziiert sein, welche Null-Werte überträgt oder entdeckt. Folgende Zeilen definieren alle Hostvariablen zum Aufnehmen eines Datensatzes der Tabelle Professoren sowie eine Indikator-Variable raum_ind zum Aufnehmen von Status-Information zur Raumangabe.

```
int   persnr;
char  name [20];
char  rang [3];
int   raum;
short raum_ind;
```

Folgende Zeilen transferieren einen einzelnen Professoren-Datensatz in die Hostvariablen persnr, name, rang, raum und überprüfen mit Hilfe der Indikator-Variable raum_ind, ob eine Raumangabe vorhanden war.

```
EXEC SQL SELECT PersNr, Name, Rang, Raum
         INTO   :persnr, :name, :rang, :raum INDICATOR :raum_ind
         FROM   Professoren
         WHERE  PersNr = 2125;
if (raum_ind == -1) printf("PersNr %d ohne Raum \n", persnr);
```

Oft liefert eine SQL-Query kein skalares Objekt zurück, sondern eine Menge von Zeilen. Diese Treffer werden in einer sogenannten private SQL area verwaltet und mit Hilfe eines Cursors sequentiell verarbeitet.

```
EXEC SQL DECLARE C1 CURSOR FOR
```

```
  SELECT PersNr, Name, Rang, Raum
  FROM   Professoren
  WHERE  Rang='C4';
EXEC SQL OPEN C1;
EXEC SQL FETCH C1 into :persnr, :name, :rang, :raum
while (SQLCODE ==0){
  printf("Verarbeite Personalnummer %d\n", persnr);
  EXEC SQL FETCH C1 into :persnr, :name, :rang, :raum
}
EXEC SQL CLOSE C1;


#include <stddef.h>                                    // Standardheader
#include <stdio.h>                                     // Standardheader

void ErrorHandler (void) {                             // im Fehlerfalle
  printf("%li %li %li\n", SQLCODE, SQLERRD1, SQLERRMC); // Ausgabe von
}                                                      // Fehlermeldungen

int main ( int argc, char** argv, char** envp) {

    EXEC  SQL BEGIN DECLARE SECTION;                   // Deklarationen-Start
    char  server[] = "arnold.uni";                     // Server + DB
    char  loginPasswort[] = "erika.mustermann";        // User + Passwort
    int   persnr;                                      // Personalnummer
    char  name[20];                                    // Name
    char  rang[3];                                     // Rang
    int   raum;                                        // Raum
    char  gebdatum[17];                                // Geburtsdatum
    short raum_ind;                                    // Raum-Indikator
    char eingaberang[3];                               // Eingabe vom User
    EXEC SQL END DECLARE SECTION;                      // Deklarationen-Ende

    EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();    // Fehlermarke
    EXEC SQL CONNECT TO :server USER :loginPasswort;   // Verbindung aufbauen

    if (SQLCODE == 0) printf("Verbindung aufgebaut!\n");// Erfolg
    else { printf("Keine Verbindung\n"); return (1); } // Misserfolg
    printf("Bitte Rang eingeben: ");                   // gewuenschten Rang
    scanf("%s", eingaberang);                          // vom user holen
    printf("Mit Rang %s gespeichert:\n", eingaberang);

    EXEC SQL DECLARE C1 CURSOR FOR                     // Cursor vereinbaren
      SELECT PersNr, Name, Rang, Raum, Gebdatum        // SQL-Statement
      FROM Professoren
      WHERE Rang = :eingaberang;
    EXEC SQL OPEN C1;                                  // Cursor oeffnen
    EXEC SQL FETCH C1 INTO :persnr, :name, :rang,      // Versuche eine Zeile
             :raum INDICATOR :raum_ind, :gebdatum;     // zu lesen

    while (SQLCODE == 0){                              // SOLANGE erfolgreich
      printf("%d %s %s", persnr, name, rang);          // Tupel ausgeben
      if(raum_ind == -1) printf(" ???");               // Platzhalter drucken
      else printf("%4d", raum);                       // SONST Raumnr
      printf(" %s\n", gebdatum);                       // letztes Attribut
      EXEC SQL FETCH C1 INTO :persnr, :name, :rang,    // naechste Zeile lesen
                  :raum INDICATOR :raum_ind, :gebdatum;
    }
    EXEC SQL CLOSE C1;                                 // Cursor schliessen
    EXEC SQL DISCONNECT ALL;                           // Verbindung beenden
    return (0);
}
```

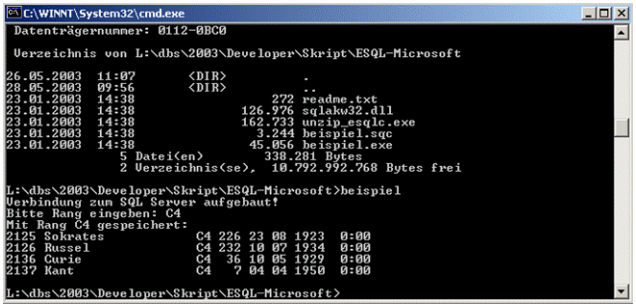


Abb. 82: Ausgabe des Embedded-SQL-Programms für Microsoft SQL-Server

Ein etwas anderer Ansatz wird beim Zugriff auf eine MySQL-Datenbank verfolgt. Die MySQL-Datenbank unterstützt Embedded SQL ebenfalls im Zusammenspiel mit den Programmiersprachen C und C++. Allerdings entfällt der Precompiler. Unter Verwendung von include-Dateien übersetzt ein C-Compiler das C-Programm `hallo.c` in die Objekt-Datei `hallo.o`. Unter Verwendung der Runtime Library wird daraus das ausführbare Programm `hallo.exe` gebunden.

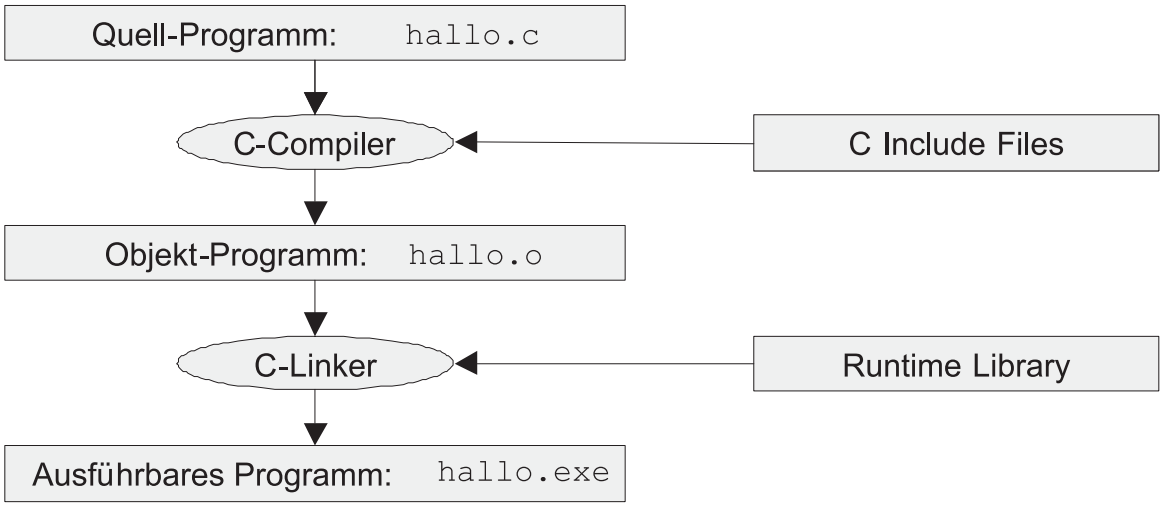


Abb. 83: Datenfluss für Embedded SQL bei MySQL-Datenbank

```c
#include <mysql.h>
#include <stdio.h>

main() {
  MYSQL       *conn;
  MYSQL_RES   *res;
  MYSQL_ROW   row;

  char *server   = "dbs.informatik.uni-osnabrueck.de";         /* Server     */
  char *user     = "erika";                                    /* User       */
  char *password = "mustermann";                               /* Passwort   */
  char *database = "UniWeb";                                   /* Datenbank  */
  char query[80] = "select * from Professoren where rang =";   /* Query      */
  char rang[3];                                                /* Rang       */

  conn = mysql_init(NULL);                          /* Verbindung herstellen   */

                                                    /* Verbindung testen       */
  if (!mysql_real_connect(conn,server,user,password,database,0,NULL,0)) {
    fprintf(stderr, "%s\n", mysql_error(conn));     /* Fehlermeldung           */
  }

  printf("Bitte Rang eingeben: ");                  /* Eingabeaufforderung     */
  scanf("%s", rang);                                /* vom User holen          */
  printf("Mit Rang %s gespeichert:\n", rang);       /* Beginn der Ausgabe      */
```

```
  strcat(query,"'");                              /* Apostroph anhaengen    */
  strcat(query, rang);                            /* Rang anhaengen         */
  strcat(query,"'");                              /* Apostroph anhaengen    */

  if (mysql_query(conn, query)) {                 /* schicke SQL-Query      */
      fprintf(stderr, "%s\n", mysql_error(conn)); /* drucke Fehlermeldung   */
  }

  res = mysql_use_result(conn);                   /* bilde Resultset        */

  while ((row = mysql_fetch_row(res)) != NULL)    /* solange Treffer        */
    printf("%s %s \n", row[0],row[1]);            /* drucke einen Treffer   */

  mysql_free_result(res);                         /* schliesse Resultset    */
  mysql_close(conn);                              /* schliesse Verbindung   */
}
```

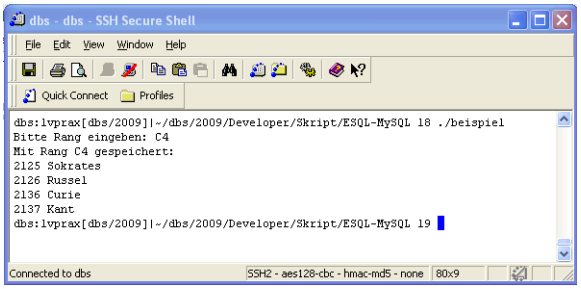


Abb. 84: Ausgabe des Embedded-SQL-Programms für MySQL-Datenbank

10.5 JDBC

JDBC (Java Database Connectivity) ist ein Java-API (Application Programming Interface) zur Ausführung von SQL-Anweisungen innerhalb von Java-Applikationen und Java-Applets. Es besteht aus einer Menge von Klassen und Schnittstellen, die in der Programmiersprache Java geschrieben sind: Java-Dokumentation zu java.sql[10].

Ein JDBC-Programm läuft in drei Phasen ab:

1. Treiber laden und Verbindung zur Datenbank aufbauen,
2. SQL-Anweisungen absenden,
3. Ergebnisse verarbeiten.

Zur Verbindung mit der Datenbank muss zuerst der externe JDBC-Treiber geladen werden. Er liegt als jar-File[11] vor. Anschließend wird vom Treiber-Manager die Verbindung aufgebaut. Dazu wird eine URL, bestehend aus dem Protokoll (`jdbc:mysql:`) gefolgt von `//`, der Host-Adresse (hier `dbs.informatik.uos.de`) und nach einem weiteren `/` dem Namen der Database (`UniWeb`), benötigt.

```
Class.forName("com.mysql.jdbc.Driver");

String url    = "jdbc:mysql://dbs.informatik.uos.de/UniWeb";
String user   = "erika";
String passwd = "mustermann";

Connection con = DriverManager.getConnection(url,user,passwd);
```

Alternativ können Nutzername und Passwort auch direkt in der URL übergeben werden:

10 http://java.sun.com/j2se/1.5.0/docs/api/java/sql/package-summary.html
11 http://media2mult.uni-osnabrueck.de/pmwiki/fields/dbs/uploads/Main/Java/mysql-connector-java-5.1.7-bin.jar

```
String url   = "jdbc:mysql://dbs.informatik.uos.de/UniWeb"
             + "?user=erika&password=mustermann";
Connection con = DriverManager.getConnection(url);
```

Während für die meisten Datenbanken ein externer Treiber geladen werden muss (bei MySQL zum Beispiel der *MySQL Connector/J*), wird der Treiber für eine ODBC-Datenquelle von Sun mitgeliefert und kann sofort eingebunden werden:

```
String url = "jdbc:odbc:dbs";                    // URL der ODBC-Datenquelle
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");   // Treiber
```

Um eine Anfrage an die Datenbank zu stellen, wird ein neues `Statement`[12]-Objekt erzeugt, dem in der Methode `executeQuery(String query)` das SQL-Statement übergeben wird. Die Methode liefert einen `ResultSet`[13] zurück, durch dessen Ergebnismenge iteriert werden kann:

```
Statement stmt = con.createStatement();
ResultSet rs   = stmt.executeQuery("select * from Professoren");

while(rs.nxt()) {
  System.out.print("Der Professor ");
  System.out.print(rs.getString("Name"));
  System.out.print(" \t geboren ");
  System.out.println(rs.getDate("gebdatum"));
}
```

Abbildung 85 zeigt die von ShowJDBC.java[14] erzeugte Ausgabe einer Java-Applikation auf der Konsole.

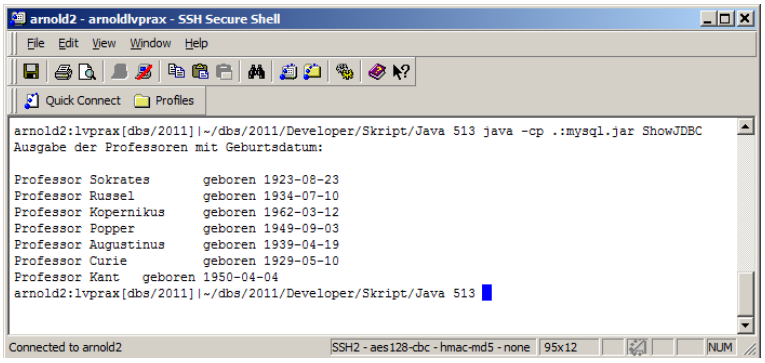


Abb. 85: Ausgabe einer Java-Applikation

Um ein Update oder Delete durchzuführen, wird statt der Methode `executeQuery` die Methode `executeUpdate` bemüht. Sie liefert die Zahl der geänderten Tupel zurück:

```
Statement stmt = con.createStatement();              // Statement
String query   = "UPDATE professoren " +             // Update-String
                 "SET rang='W1' WHERE rang IS NULL"; // vorbereiten
int x = stmt.executeUpdate(query);                   // Update durchfuehren
System.out.println("Es wurden " + x + " befoerdert.");
```

Bei mehrfacher Benutzung desselben SQL-Befehls ist es sinnvoll, diesen vorzuübersetzen als sogenanntes `PreparedStatement`[15]. Hierbei kann durch Fragezeichen ein Platzhalter definiert werden, der zur Laufzeit mittels set-Methode das noch fehlende Argument übergeben bekommt. Dabei gibt es je nach einzusetzendem

12 `http://java.sun.com/j2se/1.5.0/docs/api/java/sql/Statement.html`
13 http://java.sun.com/j2se/1.5.0/docs/api/java/sql/ResultSet.html
14 http://www-lehre.inf.uos.de/~dbs/2011/Java/ShowJDBC.java
15 http://java.sun.com/j2se/1.5.0/docs/api/java/sql/PreparedStatement.html

Typ verschiedene Methoden, so zum Beispiel `setInt(int index, int value)` für Ganzzahlen, `setString(int index, String value)` für Zeichenketten oder `setDate(int index, java.sql.Date value)` für Daten.

```
String query  = "SELECT * FROM Studenten" +     // User-Query
                " WHERE semester < ? ";         // mit Platzhalter

PreparedStatement pstmt;                        // PreparedStatement
pstmt = con.prepareStatement(query);            // initialisieren

Scanner sc = new Scanner(System.in);
System.out.print("Bitte Semesterobergrenze: "); // Benutzereingabe
int vorgabe = sc.nextInt();                     // erfassen und
pstmt.setInt(1, vorgabe);                       // einsetzen
ResultSet rs  = pstmt.executeQuery();           // ResultSet

while(rs.next()) {                              // Ergebnismenge
  System.out.print(rs.getString("Name"));       // durchlaufen
  System.out.print(" studiert im ");            // dabei Name
  System.out.print(rs.getInt("Semester"));      // und Semester
  System.out.println(". Semester.");            // ausgeben
}
```

10.6 SQLJ

Die bisherigen JDBC-Beipiele konnten zur Übersetzungszeit nicht gegen die Datenbank validiert werden, da erst zur Laufzeit der SQL-Query-String an den SQL-Interpreter übergeben wurde. SQLJ löst dieses Problem durch den Einsatz eines Translators, welcher zunächst das mit #SQL-Kommandos erweiterte Javaprogramm in reguläres Java überführt, welches anschließend mit dem Java-Compiler übersetzt wird. Der Translator kann dabei bereits mit Hilfe der Treiberklassen Kontakt zur Datenbank aufnehmen und die SQL-Befehle auf Korrektheit überprüfen.

Abbildung 86 zeigt das Zusammenspiel von Präprozessor, Java-Compiler und Java-Interpreter, ausgehend von einer *.sqlj-Datei.

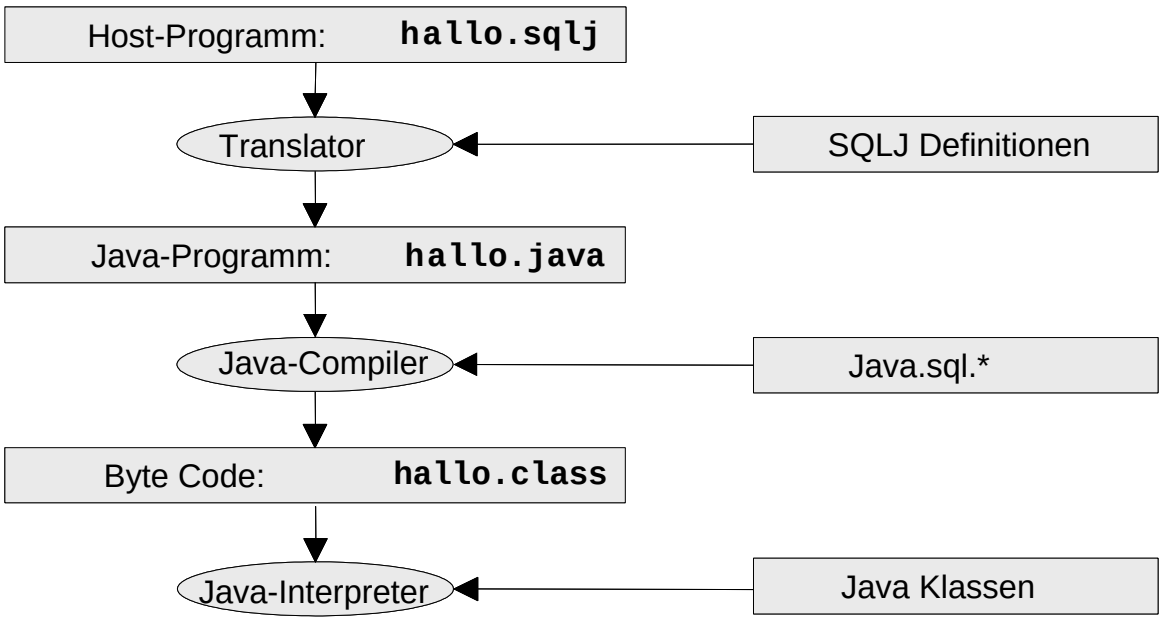


Abb. 86: Datenfluss bei SQLJ

Ausgangsprogramm ShowSqljHost.sqlj mit Treiber für MySQL-Server unter Verwendung von Hostvariablen[16]

16 http://www-lehre.inf.uos.de/~dbs/2011/Java/ShowSqljHost.sqlj

Daraus erzeugtes Java-Programm ShowSqljHost.java[17]

Ausgangsprogramm ShowSqljIter.sqlj mit Treiber für MySQL-Server unter Verwendung eines Iterators[18]

Daraus erzeugtes Java-Programm ShowSqljIter.java[19]

10.7 SQLite und HSQLDB

Oft ist es nicht nötig für eine Datenbankapplikation einen vollständigen Server aufzusetzen. Zum Beispiel wenn stets nur ein Benutzer an demselben Ort auf die Datenbank zugreift. Eine Serververbindung bremst in solchen Fällen die Anwendung und führt zu Performance Einbußen. Besonders im Kontext von weniger leistungsfähigen Embedded Systemen macht eine Standalone Datenbank deswegen Sinn.

SQLite

SQLite ist eine in C geschriebene Programmbibliothek die SQL Abfragen unterstützt. Die Datenbank wird direkt aus einer einzigen Datei gelesen und geschrieben, dadurch ist Mehrbenutzerzugriff nicht möglich, es vereinfacht aber den Austausch ganzer Datenbanken. Die Bibliotheken sind nur wenige hundert Kilobyte groß und können so leicht in eine Applikation eingebunden werden. Für die Programmiersprachen Java und C++ existieren Wrapper, zu dem gibt es JDBC und ODBC Treiber. SQLite ist unter anderem in Betriebssystemen für Mobiltelefone, wie Android, Symbian OS oder dem angepassten Mac OS X für das iPhone, integriert.
In einem Javaprogramm kann über eine JDBC-Brücke eine Verbindung zu einer SQLite Datenbank hergestellt werden:

```
Class.forName("org.sqlite.JDBC");
Connection conn = DriverManager.getConnection("jdbc:sqlite:" + path + dbname, user, passw);
```

Wenn die Datenbank `dbname` im Ordner `path` noch nicht existiert, wird Sie automatisch angelegt. Im betreffenden Ordner würde also eine Datei `dbname` erzeugt, mit der alle Operationen durchgeführt werden. Einzige Voraussetzung ist, dass ein jar-File[20] mit den benötigten Treibern und Bibliotheken vorhanden ist.

HSQLDB

Im Gegensatz zu SQLite ist HSQLDB eine vollständig in Java geschriebene SQL-Datenbank, die große Teile der Standards 92, 99 und 2003 unterstützt. Es gibt mehrere Möglichkeiten eine Datenbank zu erzeugen. Zum einen als Server zu dem sich mehrere Clienten verbinden können, zum anderen kann die Datenbank auch nur zur Laufzeit im Arbeitsspeicher generiert werden und parallel zur Applikation exitstieren. Die dritte Möglichkeit ist eine Datenbank im `file`-Modus ähnlich zu SQLite zu erzeugen:

```
Class.forName("org.hsqldb.jdbcDriver");
Connection conn = DriverManager.getConnection("jdbc:hsqldb:file:" + path + dbname, user, passw);
```

Wenn die Datenbank nicht existiert, wird Sie automatisch erzeugt. Einzige Voraussetzung ist, dass das nötige jar-File[21] zur Verfügung steht. Damit die Datenbank persistent die Laufzeit des Programms überdauert, ist es wichtig, sie über das SQL Statement `SHUTDOWN` zu beenden:

17 http://www-lehre.inf.uos.de/~dbs/2011/Java/ShowSqljHost.java
18 http://www-lehre.inf.uos.de/~dbs/2011/Java/ShowSqljIter.sqlj
19 http://www-lehre.inf.uos.de/~dbs/2011/Java/ShowSqljIter.java
20 http://media2mult.uni-osnabrueck.de/pmwiki/fields/dbs/uploads/Main/Java/sqlitejdbc-v054.jar
21 http://media2mult.uni-osnabrueck.de/pmwiki/fields/dbs/uploads/Main/Java/hsqldb.jar

```
Statement st = conn.createStatement();
st.execute("SHUTDOWN");
st.close();
conn.close();
```

Dabei werden die Dateien `dbname.properties` mit den nötigen Metainformationen und `dbname.script` mit dem Inhalt der Datenbank in Form von `CREATE`, und `INSERT` Befehlen und den Zugriffsrechten erzeugt. Die Befehle in `dbname.script` werden beim Öffnen der Verbindung einmal in den Speicher geladen, was Datenbankanfragen sehr performant macht.

Es ist ebenfalls möglich Tabellen zu cachen und nicht vollständig in den Speicher zu lesen, oder in Text-Tables mit Verlinkung zu `*.csv` Dateien abzuspeichern. Durch diesen nur-lesen Zugriff kann SQLite auch auf CD/DVD Datenträgern eingesetzt werden. Zur Zeit existieren spezielle HSQLDB Versionen für Handhelds und PDAs und es wird in OpenOffice verwendet.

Wenn die Verbindung sowohl zu SQLite als auch zu HSQLDB ersteinmal über JDBC erfolgreich aufgebaut wurde, kann die weitere Verwendung wie gewohnt erfolgen. Da der SQL Standard nicht vollständig unterstützt wird, ist ein Blick in Dokumentationen dabei unerlässlich:

- HSQLDB Homepage[22]
- SQLite Homepage[23]

Beispielprogramme unter Verwendung von SQLite:

- SQLiteLeseTest.java[24]
- SQLiteSchreibTest.java[25]
- SQLiteTest.java[26]

Ausgabe des Programms SQLiteTest.java

```
Oeffne Verbindung mit SQLite Datenbank...
Initialisiere SQLite Datenbank...
Teste SQLite Datenbank...
PersNr   | Name          | Rang | Raum
2125     | Sokrates      | C4   | 226
2126     | Russel        | C4   | 232
2127     | Kopernikus    | C3   | 310
Schliesse SQLite Datenbank und oeffne erneut...
PersNr   | Name          | Rang | Raum
2125     | Sokrates      | C4   | 226
2126     | Russel        | C4   | 232
2127     | Kopernikus    | C3   | 310
```

Beipielprogramm zur Verwendung von HSQLDB:

- HSQLDBTest.java[27]

Ausgabe des Programms HSQLDBTest.java:

```
Oeffne Verbindung mit HSQLDB...
Initialisiere HSQLDB...
```

22 http://hsqldb.org/
23 http://www.sqlite.org/
24 http://www-lehre.inf.uos.de/~dbs/2011/Java/SQLiteLeseTest.java
25 http://www-lehre.inf.uos.de/~dbs/2011/Java/SQLiteSchreibTest.java
26 http://www-lehre.inf.uos.de/~dbs/2011/Java/SQLiteTest.java
27 http://www-lehre.inf.uos.de/~dbs/2011/Java/HSQLDBTest.java

```
Teste HSQLDB...
PersNr   | Name          | Rang | Raum
2125     | Sokrates      | C4   | 226
2126     | Russel        | C4   | 232
2127     | Kopernikus    | C3   | 310
Schliesse HSQLDB und oeffne erneut...
PersNr   | Name          | Rang | Raum
2125     | Sokrates      | C4   | 226
2126     | Russel        | C4   | 232
2127     | Kopernikus    | C3   | 310
```

Die von HSQLDB erzeugte *.script Datei lautet:

```
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE MEMORY TABLE PROFESSOREN(PERSNR INTEGER NOT NULL PRIMARY KEY,
NAME VARCHAR(20) NOT NULL,RANG CHAR(2),RAUM INTEGER)
CREATE USER SA PASSWORD ""
GRANT DBA TO SA
SET WRITE_DELAY 10
SET SCHEMA PUBLIC
INSERT INTO PROFESSOREN VALUES(2125,'Sokrates','C4',226)
INSERT INTO PROFESSOREN VALUES(2126,'Russel','C4',232)
INSERT INTO PROFESSOREN VALUES(2127,'Kopernikus','C3',310)
```

10.8 Java Applets

Ein Hauptanwendungsbereich von JDBC liegt in Java-Applets. Hierbei wird eine Datenbank aus dem Applet heraus angesprochen, um diese auszulesen oder zu manipulieren. Aus Sicherheitsgründen darf ein Applet nur eine Socket-Verbindung zu dem Server aufbauen, von dem es geladen wurde. Daher müssen Web-Server und Datenbankserver auf dem selben Rechner liegen oder dieser Sachverhalt muss durch ein Port-Forwarding simuliert werden.

Der folgende Ausschnitt zeigt den notwendigen Code zum Aufruf einer HTML-Seite mit Java-Applet[28]. Das dort ausgeführte Applet ermöglicht den lesenden Zugriff auf die Datenbank UniWeb.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
       "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>JDBC-Applet UniWeb</title>
</head>
<body>
  <h1>Demo-Applet f&uuml;r JDBC-Datenbankzugriff</H1>
  <div>
    <object classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
      codetype="application/java-vm" width="700" height="400"
      data="applet.JDBCApplet.class">
      <param name="archive"
        value="mysql-connector-java-5.1.7-bin.jar" >
      <param name="code" value="applet.JDBCApplet.class">
      <param name="type" value="application/java-vm">
      <comment>
        <object classid="java:applet.JDBCApplet.class"
          codetype="application/java-vm" width="700" height="400">
          <param name="archive"
            value="mysql-connector-java-5.1.7-bin.jar" >
        </object>
      </comment>
    </object>
  </div>
</body>
```

28 http://dbs.informatik.uos.de/media2mult/applet/

```
</html>
```

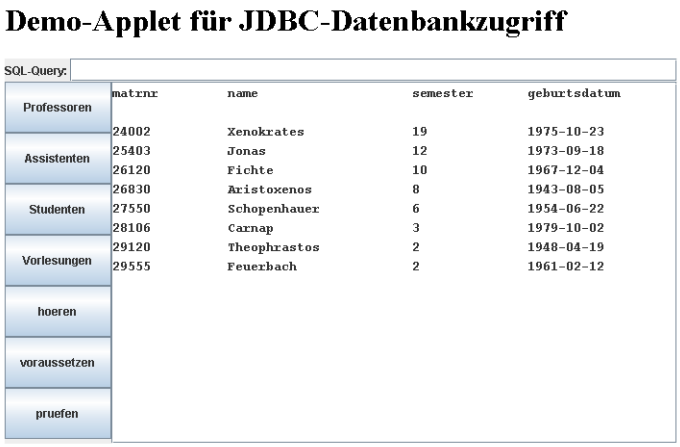


Abb. 87: Java-Applet mit JDBC-Zugriff auf MySQL-Datenbank

```
package applet;

import java.awt.BorderLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.ScrollPaneConstants;

public final class JDBCApplet extends JApplet {
  JTextArea outputArea = new JTextArea();
  JScrollPane pane;
  JTextField inputField = new JTextField();
  Connection con;            /* Verbindungsobjekt zur Verbindung mit dem DBMS */
  Statement stmt;              /* Statement-Objekt zur Kommunikation mit DBMS */



  /* Innere Klasse fuer Buttons mit festen Querys */
  private class QueryButton extends JButton {
    private final String query;
    QueryButton(String titel, String query) {
      super(titel);
      this.query = query;
      addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery(QueryButton.this.query);            /* Query durchfuehren */
        }
      });
    }
  }
```

```
/* Konstruktor fuer Applet */
public JDBCApplet() { }

/* Das Applet initialisieren */
public void init() {

  try {
    setLayout(new BorderLayout());                    /* Aussehen anpassen */
    setSize(700, 400);

                                /* SQL-Eingabefeld mit Listener einrichten */
    inputField.setFont(new Font(Font.MONOSPACED,Font.BOLD,14));
    inputField.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        execQuery(inputField.getText());
      }
    });
    JPanel top = new JPanel();
    top.setLayout(new BorderLayout());
    top.add(new JLabel("SQL-Query: "), BorderLayout.WEST);
    top.add(inputField, BorderLayout.CENTER);
    add(top, BorderLayout.NORTH);

    outputArea.setEditable(false);                /* Ausgabefeld einrichten */
    outputArea.setFont(new Font(Font.MONOSPACED,Font.BOLD,14));
    pane = new JScrollPane(outputArea,                  /* scrollbar machen */
      ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED,
      ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    this.add(pane, BorderLayout.CENTER);

                   /* Buttons fuer spezielle Querys zur Verfuegung stellen */
    List<QueryButton> qu = new ArrayList<QueryButton>(8);

    qu.add(new QueryButton("Professoren",        /* Button fuer Professoren */
        "select persnr,name,rang,raum,"
        +"gebdatum as geburtsdatum from Professoren"));

    qu.add(new QueryButton("Assistenten",        /* Button fuer Assistenten */
        "select persnr, name, fachgebiet, boss,"
        + "gebdatum as geburtsdatum from Assistenten"));

    qu.add(new QueryButton("Studenten",            /* Button fuer Studenten */
        "select matrnr, name, semester,"
        + "gebdatum as geburtsdatum from Studenten"));

                                                 /* Button fuer Vorlesungen */
    qu.add(new QueryButton("Vorlesungen", "select * from Vorlesungen"));

                                                    /* Button fuer hoeren */
    qu.add(new QueryButton("hoeren", "select * from hoeren"));

    qu.add(new QueryButton("voraussetzen",      /* Button fuer voraussetzen */
        "select * from voraussetzen"));

                                                   /* Button fuer pruefen */
    qu.add(new QueryButton("pruefen", "select * from pruefen"));

                                                 /* Buttons links einhaengen */
    JPanel p = new JPanel(new GridLayout(qu.size(), 1));
    for (QueryButton qb : qu)
      p.add(qb);
    add(p, BorderLayout.WEST);
  } catch (Exception e) {
    e.printStackTrace();
  }

                                               /* Verbindungsdaten des DBMS */
  String url = "jdbc:mysql://dbs.informatik.uos.de/UniWeb";
  String user = "erika";
  String passwd = "mustermann";

  try {                                           /* Treiberklassen laden */
    Class.forName("com.mysql.jdbc.Driver");
  } catch (java.lang.ClassNotFoundException ex) {
```

```
      System.err.print("ClassNotFoundException: ");
      System.err.println(ex.getMessage());
    }

    try {                                          /* Verbindung herstellen */
      con = DriverManager.getConnection(url, user, passwd);
    } catch (SQLException ex) {
      outputArea.setText(formatMessage(ex));
    }
  }

  /* Exception-Ausgabe formatieren */
  private String formatMessage(SQLException e) {
    StringBuffer b = new StringBuffer();
    b.append("SQLException: ");
    b.append(e.getErrorCode() + "\n");
    b.append(e.getMessage()+"\n");
    return b.toString();
  }

  /* Abfrage ausfuehren und Ergebnisse ausgeben */
  void execQuery(String query) {
    try {
      stmt = con.createStatement();         /* Statement-Objekt instantiieren */

      ResultSet rs = stmt.executeQuery(query);          /* Query durchfuehren */

                                                /* Zahl der Ergebnisspalten */
      int spalten = rs.getMetaData().getColumnCount();


                                      /* alle Spaltennamen formatiert ausgeben */
      StringBuilder b = new StringBuilder();
      for (int i = 1; i <= spalten; i++) {
        String lab = rs.getMetaData().getColumnLabel(i);
        int max = rs.getMetaData().getColumnDisplaySize(i) + 4;
        int cur = b.length();
        b.setLength(cur+max);
        b.replace(cur, cur+lab.length(),lab);
      }
      b.append("\n");


      String arg;                                   /* Ergebnisse ausgeben */
      while (rs.next()) {                          /* alle Zeilen durchlaufen */
        b.append("\n");
        for (int i = 1; i <= spalten; i++) {      /* alle Spalten durchlaufen */
          arg = rs.getString(i);
          int max = rs.getMetaData().getColumnDisplaySize(i) + 4;
          int cur = b.length();
          b.setLength(cur+max);
          if (arg != null)
            b.replace(cur, cur+arg.length(),arg);
          else
            b.replace(cur, cur+4,"null");
        }
      }
      b.append("\n");
      outputArea.setText(b.toString().replaceAll("\u0000", " "));

      rs.close();                                   /* Ressourcen freigeben */
      stmt.close();
    } catch (SQLException ex) {           /* Abfangen von SQL-Fehlermeldungen */
      outputArea.setText(formatMessage(ex));
    }
  }

}
```

10.9 Java Servlets

Der Einsatz von JDBC auf dem Rechner des Clienten als Applet wird von den Sicherheitsbeschränkungen beeinträchtigt, unter denen Applets ausgeführt werden können. Als Alternative bieten sich Java-Servlets an, die auf dem Server des Datenbankanbieters laufen. Mit einem Formular auf einer HTML-Seite auf dem Web-Server des Anbieters werden die Argumente vom Anwender ermittelt und dann an das in Java formulierte Servlet übergeben. Per JDBC wird die Query an den Datenbankserver gerichtet und das Ergebnis mit Java-Befehlen zu einer dynamischen HTML-Seite aufbereitet.

Listing 9.6 zeigt eine HTML-Seite zum Erfassen der Anwendereingabe, Listing 9.7 zeigt den Quelltext des zugehörigen Java-Servlet, welches vom Rechner `dbs` aus mit dem MySQL-JDBC-Treiber den MySQL-Server kontaktiert. Die erzeugte Ausgabe findet sich in Abbildung 88.

```
<html>
  <head><title>Vorlesungsverzeichnis mittels Java-Servlet</title></head>
  <body>
    <form
     method="GET"
     action="VrlVrz">
     Bitte geben Sie den Namen eines Professors ein:
     <p><input name="professor_name" size="40"><p>
     <input type="submit" value="Vorlesungen ermitteln">
    </form>
  </body>
</html>
```

Aufruf einer HTML-Seite zur Versorgung des Servlets VrlVrz.java[29]

```
import javax.servlet.*; import javax.servlet.http.*; import java.io.*;
import java.sql.*; import java.text.*;

public class VrlVrz extends HttpServlet {

  public void doGet (HttpServletRequest request,HttpServletResponse response)
                    throws ServletException, IOException {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs   = null;
    response.setContentType("Text/html");
    PrintWriter out = response.getWriter();
    try {
      Class.forName("com.mysql.jdbc.Driver");
      con = DriverManager.getConnection(
      "jdbc:mysql://dbs.informatik.uni-osnabrueck.de/UniWeb",
                                        "erika","mustermann");
      stmt = con.createStatement();
      String query = "select v.VorlNr, v.Titel, v.SWS " +
                     "from Vorlesungen v, Professoren p " +
                     "where v.gelesenVon = p.PersNr and p.Name ='" +
                      request.getParameter("professor_name") + "'";
      rs = stmt.executeQuery(query);
      out.println("<HTML>");
      out.println("<HEAD><TITLE>Java Servlet</TITLE></HEAD>");
      out.println("<BODY>");
      out.println("<H1>Vorlesungen von Prof. " +
                  request.getParameter("professor_name") +": </H1>");
      out.println("<UL>");
      while (rs.next())
        out.println("<LI>" +
          rs.getInt("VorlNr") + ": " + rs.getString("Titel") + " (mit " +
          rs.getInt("SWS") + " SWS)" + "</LI>");
      out.println("</UL>");
      out.println("<BODY></HTML>");
    }
```

29 http://dbs.informatik.uos.de:8180/VrlVrz/

```
    catch(ClassNotFoundException e) {
      out.println("Datenbanktreiber nicht gefunden: " + e.getMessage());
    }
    catch(SQLException e) {
      out.println("SQLException: " + e.getMessage());
    }
    finally {
      try { if (con != null ) con.close();
      } catch (SQLException ignorieren) {}
    }
  }
}
```

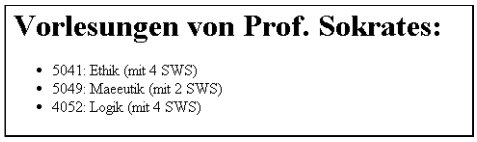


Abb. 88: vom Java-Servlet erzeugte Ausgabe

10.10 Java Server Pages

Java-Servlets vermischen oft in verwirrender Weise HTML mit Java, d.h. Layout-Informationen und algorithmische Bestandteile. Abhilfe wird geschaffen durch sogenannte Java Server Pages, in denen eine bessere Trennung zwischen statischem Layout und den benötigten algorithmischen Bestandteilen ermöglicht wird.

Die wesentliche Inhalt einer Java-Server-Page besteht aus HTML-Vokabeln, die nun Methoden einer Java-Klasse aufrufen dürfen (gekennzeichnet durch spitze Klammern und Prozentzeichen).

Listing 9.8 zeigt die Java-Server-Page `vorlesungen.jsp`, welche zuständig ist für die Erfassung der Benutzereingabe und die Ausgabe der Datenbankantwort. Die Eingabe wird über ein HTML-Formular ermittelt und über den Feldnamen `profname` an die Java-Bean `VorlesungenBean` übermittelt. Die Ausgabe wird im wesentlichen bestritten durch einen String, der von der Methode `generiereVorlListe()` geliefert wird.

Verwendet wird hierbei der Tomcat-Server von Apache, der die Java-Server-Page in ein Servlet übersetzt und mit der zugehörigen Java-Bean verbindet. Dies geschieht automatisch immer dann, sobald sich der Quelltext der JSP-Seite geändert hat. Die generierten Webseiten werden in Abbildungen 89 bzw. 90 gezeigt.

```
<%@ page import="dbs.VorlesungenBean" %>
<jsp:useBean id="prg" class="dbs.VorlesungenBean" scope="request" />
<jsp:setProperty name="prg" property="*" />

<html>
  <% if (prg.getProfname() == null) { %>
    <head>
      <title>Professoren-Namen erfassen</title>
    </head>
    <body>
      <form method="get">
        <p>Bitte geben Sie den Namen eines Professors ein:</p>
        <p><input type="text" name="profname" /></p>
        <p><input type="submit" value="Vorlesungen ermitteln!" /></p>
      </form>
    </body>
  <% } else { %>
    <head>
      <title>Vorlesungen ausgeben</title>
    </head>
    <body>
      <p>Die Vorlesungen von <%= prg.getProfname() %> lauten: </p>
      <%= prg.generiereVorlListe() %>
    </body>
```

```
  <% } %>
</html>
```

Aufruf der Java-Server-Page vorlesungen.jsp auf dem TomCat-Server dbs[30]

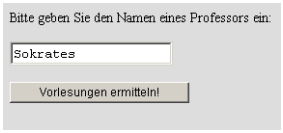


Abb. 89: von der Java-Server-Page erzeugtes Eingabeformular

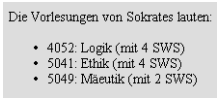


Abb. 90: von der Java-Server-Page erzeugte Ausgabe

```
package dbs;

import java.sql.*;

public class VorlesungenBean {

  Connection con;
  String error;
  String profname;

  public VorlesungenBean() {

    String url    = "jdbc:mysql://dbs.informatik.uos.de/UniWeb";
    String user   = "erika";
    String passwd = "mustermann";

    try {
      Class.forName("com.mysql.jdbc.Driver");
      con = DriverManager.getConnection(url, user, passwd);
    }
    catch(Exception e) {
      error = e.toString();
    }
  }

  public void setProfname(String name) {
    profname = name;
  }

  public String getProfname() {
    return profname;
  }

  public String generiereVorlListe() {
    Statement stmt = null;
    ResultSet rs = null;

    if (con == null)
      return "Probleme mit der Datenbank: " + error + "<br />";

    StringBuffer result = new StringBuffer();

    try {
      stmt = con.createStatement();
      String query =
        "select v.VorlNr, v.Titel, v.SWS " +
```

30 http://dbs.informatik.uos.de:8180/vorlesungen/vorlesungen.jsp

```
        "  from Vorlesungen v, Professoren p " +
        " where v.gelesenVon = p.PersNr " +
        "   and p.name ='" + profname + "'";
      rs = stmt.executeQuery(query);
      result.append("<ul>");
      while (rs.next())
        result.append("<li>" + rs.getInt("VorlNr") + ": " + rs.getString("Titel") +
                      " (mit " + rs.getInt("SWS") + " SWS)" + "</li>");
      result.append("</ul>");
    }
    catch(SQLException e) {
      result.append("Bei der Abfrage fuer " + profname +
               " trat ein Fehler auf: " + e.getMessage() + "<br />");
    }
    return result.toString();
  }

  public void finalize () {
    try {
      if (con != null)
        con.close();
    } catch (SQLException ignorieren) {}
  }
}
```

10.11 PHP

PHP (rekursives Akronym für PHP: Hypertext Preprocessor) ist eine weit verbreitete und für den allgemeinen Gebrauch bestimmte Skriptsprache, welche speziell für die Webprogrammierung geeignet ist und in HTML eingebettet werden kann. Anstatt ein Programm mit vielen Anweisungen zur Ausgabe von HTML zu schreiben, schreibt man etwas HTML und bettet einige Anweisungen ein, die irgendetwas tun. Der PHP-Code steht zwischen speziellen Anfangs- und Abschlusstags <?php und ?>, mit denen man in den "PHP-Modus" und zurück wechseln kann.

PHP unterscheidet sich von clientseitigen Sprachen wie Javascript dadurch, dass der Code auf dem Server ausgeführt wird und dort HTML-Ausgaben generiert, die an den Client gesendet werden. Der Client erhält also nur das Ergebnis der Skriptausführung, ohne dass es möglich ist herauszufinden, wie der eigentliche Code aussieht. Neben dem Generieren von dynamischen HTML-Seiten, ist es aber auch durchaus möglich Grafiken oder andere Dateitypen mit PHP-Skripten zu erzeugen. Wir beschränken uns hier aber auf die Angabe von einigen wenigen Beispielen, in denen gegen einen *MySQL Server* eine SQL-Query abgesetzt und das Ergebnis tabellarisch bzw. graphisch angezeigt wird. Zur Erleichterung der Entwicklung verwenden wir verschiedene Module aus dem PEAR - PHP Extension and Application Repository[31]. So zum Beispiel MDB2 für die Datenbankanbindung und PHPLIB zur Arbeit mit Templates.

Die einfachste Möglichkeit zur Übergabe der Query an die PHP-Seite besteht darin, die Query beim Aufruf der PHP-Seite an die URL der PHP-Seite zu hängen:

http://dbs.informatik.uos.de/media2mult/php/antwort.php?frage=select+name+,+rang+from+Professoren

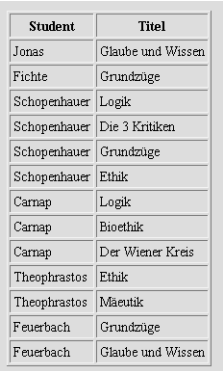
Eine verbesserte Version dieses Beispiel besteht darin, die Query durch ein Formular in einer HTML-Seite zu ermitteln und von dort aus die PHP-Seite aufzurufen. PHP legt dann automatisch Variablen mit den Namen der in dem Formular verwendeten Feldern an.

Im Folgenden ist eine HTML-Seite mit einem Formular zur Erfassung einer SQL-Query, sowie das PHP-Skript zur Verarbeitung dieser Anfrage gezeigt. Die vom Benutzer eingegebene Anfrage wird beim Absenden des Formulare an das PHP-Skript übergeben und ausgeführt. Das Ergebnis einer möglichen Anfrage ist in Abbildung 91 zu sehen.

```
<html>
```

31 http://pear.php.net

```
  <head>
    <title>Frage</title>
  </head>
  <body>
    <form method="post" action="./antwort.php">
      Bitte geben Sie eine SQL-Query ein:
      <p><input name="frage" size="70"></p>
      <input type="submit" value="Query absetzen">
    </form>
  </body>
</html>


<html>
  <head>
    <title>Antwort auf DB-Query</title>
  </head>
  <body bgcolor="#dddddd">
<?php
require_once 'MDB2.php';

// Datenbankverbindung aufbauen
$dsn = array(
  'phptype'  => 'mysql',
  'username' => 'erika',
  'password' => 'mustermann',
  'hostspec' => 'dbs',
  'database' => 'UniWeb',
);

$con =& MDB2::connect($dsn);
if (PEAR::isError($con))
  die($con->getMessage());

// uebergebene Abfrage ausfuehren
$result = $con->query($_REQUEST['frage']);
if (PEAR::isError($result))
  die($result->getMessage());

// Spaltenanzahl und Beschriftung ermitteln
$s = $result->numCols();
$header = $result->getColumnNames();

// Header der Tabelle ausgeben
echo "<table border=\"2\" cellpadding=\"3\">\n";
echo "<tr>";
for ($i = 0; $i < $s; $i++) {
  echo "<td>".ucfirst(key($header))."</td>";
  next($header);
}
echo "</tr>\n";

// Datensaetze ausgeben
while($row = $result->fetchRow()) {
  echo "<tr>";
  for($i = 0; $i < $s; $i++) {
    echo "<td>$row[$i]</td>";
  }
  echo "</tr>\n";
}

echo "</table>\n";

// DB-Verbindung beenden
$result->free();
$con->disconnect();

?>
  </body>
</html>
```



Student	Titel
Jonas	Glaube und Wissen
Fichte	Grundzüge
Schopenhauer	Logik
Schopenhauer	Die 3 Kritiken
Schopenhauer	Grundzüge
Schopenhauer	Ethik
Carnap	Logik
Carnap	Bioethik
Carnap	Der Wiener Kreis
Theophrastos	Ethik
Theophrastos	Mäeutik
Feuerbach	Grundzüge
Feuerbach	Glaube und Wissen

Abb. 91: Ergebnis einer SQL-Query, ermittelt durch antwort.php

Aufruf des Formulares frage.html[32]

Formulare in denen der Benutzer direkt ein SQL-Statement eingeben kann, wird man so sicherlich nie finden, da hier eine enorm große Gefahr besteht, dass ein Benutzer Datensätze, Tabellen oder schlimmstenfalls die gesamte Datenbank löscht, da das SQL-Statement ungeprüft an die Datenbank weitergereicht wird.

Um eine möglichst effektive Trennung von PHP-Skripten und reinem HTML zu erhalten, haben sich Templates bewährt. Diese Template-Dateien, welche das Formatierungsgerüst die Ausgabe enthalten, werden aus dem PHP-Skript dynamisch mit Daten gefüllt und ausgegeben. Vorteil dieser Zweiteilung von Form und Funktionalität ist die Tatsache, dass die Template-Datei in einem beliebigen HTML-Editor gestaltet werden kann.

Im folgenden Beispiel haben wir eine Template-Datei mit einem Blockbereich, der aus dem PHP-Skript heraus Werte zugewiesen bekommt und im Bedarfsfall wiederholt wird. Das generierte Ergebnis ist in Abbildung 92 zu sehen.

```
<html>
  <head>
    <title>Auflistung der Assistenten</title>
  </head>
  <body bgcolor="#abcdef">
    <table border="1" cellspacing="4">
      <tr>
        <th>Name</th><th>Fachgebiet</th><th>Betreuer</th>
      </tr>
      <!-- BEGIN row -->
      <tr>
        <td>{name}</td><td>{fachgebiet}</td><td>{betreuer}</td>
      </tr>
      <!-- END row -->
    </table>
  </body>
</html>


<?php
require_once 'MDB2.php';
require_once 'HTML/Template/PHPLIB.php';

// Verbindungsinformationen in einem Array sammeln
$dsn = array(
  "phptype"  => "mysql",
  "username" => "erika",
  "password" => "mustermann",
  "hostspec" => "dbs",
  "database" => "UniWeb",
);

// Herstellen der Verbindung
// und ggf. auf einen Fehler reagieren
```

32 http://dbs.informatik.uos.de/media2mult/php/frage.html

```php
$con =& MDB2::connect($dsn);
if (PEAR::isError($con))
  die($con->getMessage());

// Erstellen des SQL-Statements und ausfuehren
// gegen die Datenbank
$sql = "SELECT a.name AS Name,
        a.fachgebiet AS Fachgebiet,
        p.name AS Betreuer
        FROM Assistenten a, Professoren p
        WHERE a.boss = p.persnr";
$result = $con->query($sql);
if (PEAR::isError($result))
  die($result->getMessage());

// Laden des HTML-Templates
$template = new HTML_Template_PHPLIB();
$template->setFile("assistenten", "assistenten.ihtml");
$template->setBlock("assistenten", "row", "rows");

// alle Datensaetze der Query durchlaufen
while($row = $result->fetchRow(MDB2_FETCHMODE_ASSOC)) {

  // alle Felder eines Tupels dem Template uebergeben
  foreach($row as $field => $value) {
    $template->setVar($field, $value);
  }

  // Einfuegen der einzelnen Entries in eine Zeile
  $template->parse("rows", "row", true);
}

// Template ausgeben
$template->pParse("output", "assistenten");

// DB-Verbindung beenden
$result->free();
$con->disconnect();
?>
```

Name	Fachgebiet	Betreuer
Platon	Ideenlehre	Sokrates
Aristoteles	Syllogistik	Sokrates
Wittgenstein	Sprachtheorie	Russel
Rhetikus	Planetenbewegung	Kopernikus
Newton	Keplersche Gesetze	Kopernikus
Spinoza	Gott und Natur	Augustinus

Abb. 92: Ergebnis einer SQL-Query, ermittelt durch assistenten.php

Aufruf des PHP-Scripts assistenten.php[33]

Mit Hilfe der Image-Funktionen[34] von PHP lassen sich bereits einfache Grafiken erzeugen. Das Beispiel zeigt ein HTML-Template, welches mittels zweier PHP-Skripte mit den Namen und der Semesterzahlen der Studenten versehen wird. Die Dauer des Studiums wird dabei grafisch durch einen Balken visualisiert. Hierzu wird in dem PHP-Script balken.php mithilfe der GD-Library ein blaues Rechteck gezeichnet und mit einer Breite gemäß dem übergebenen Parameter $zahl versehen. Das Ergebnis ist in Abbildung 93 zu sehen.

```php
<html>
  <head>
    <title>Berechnung von dynamischen Grafiken</title>
  </head>
  <body>
    <table border="1">
```

33 http://dbs.informatik.uos.de/media2mult/php/assistenten.php
34 http://www.selfphp.de/funktionsuebersicht/image_funktionen.php

```
      <tr>
        <th>Student</th><th>Studiendauer</th>
      </tr>
      <!-- BEGIN row -->
      <tr>
        <td>{name}</td><td>{grafik}</td>
      </tr>
      <!-- END row -->
    </table>
  </body>
</html>
```

```
<?php
require_once 'MDB2.php';
require_once 'HTML/Template/PHPLIB.php';

// Datenbankverbindung aufbauen
$dsn = array(
  'phptype'  => 'mysql',
  'username' => 'erika',
  'password' => 'mustermann',
  'hostspec' => 'dbs',
  'database' => 'UniWeb',
);

$con =& MDB2::connect($dsn);
if (PEAR::isError($con))
  die($con->getMessage());

// Hole Name und Semester aller Studenten
$sql = "SELECT name, semester FROM Studenten ORDER BY name";
$result = $con->query($sql);
if (PEAR::isError($result))
  die($result->getMessage());

// Laden des HTML-Templates
$template = new HTML_Template_PHPLIB();
$template->setFile("semester", "semester.ihtml");
$template->setBlock("semester", "row", "rows");

// Balkengrafik pro Eintrag erzeugen
while($row = $result->fetchRow()) {
  $template->setVar("name", $row[0]);
  $template->setVar("grafik", "<img src=\"./balken.php?zahl=$row[1]\">");

  // Zeile ins Template einfuegen
  $template->parse("rows", "row", true);
}
// Template ausgeben
$template->pParse("output", "semester");

// DB-Verbindung beenden
$result->free();
$con->disconnect();
?>
```

```
<?php
  $breite = $_GET['zahl'] * 10;
  $hoehe = 30;
  $bild = imagecreate($breite, $hoehe);
  $farbe_balken = imagecolorallocate($bild, 0, 0, 255);
  $farbe_schrift = imagecolorallocate($bild, 255, 255, 255);
  ImageString($bild, 3, $breite - 16, 8, $_GET['zahl'], $farbe_schrift);
  header("Content-Type: image/png");
  imagepng($bild);
?>
```

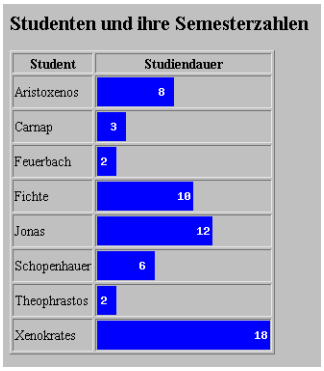


Abb. 93:
Dynamische Grafiken erzeugt von balken.php, aufgerufen von semester.php

Aufruf des PHP-Scripts semester.php zur Berechnung von dynamischen Grafiken[35]

Unter Verwendung der GD-Library erzeugen die Routinen der JpGraph-Library komplexere Bilder, wie z.B. Tortengrafiken (http://www.aditus.nu/jpgraph/index.php). Unser Beispiel zeigt ein PHP-Script, welches die Lehrbelastung der Professoren ermittelt und die relativen Anteile durch eine Tortengraphik visualisiert. Hierzu werden die Namen und Zahlenwerte im PHP-Script torte.php ermittelt und dort mithilfe der jpgraph-Library visualisiert. Das Ergebnis ist in Abbildung 94 zu sehen.

```
<html>
  <head>
    <title>Lehrbelastung der Professoren</title>
  </head>
  <body bgcolor="silver">
    <img src="./torte.php">
  </body>
</html>
```

```
<?php
require_once './jpgraph-2.3.4/src/jpgraph.php';
require_once './jpgraph-2.3.4/src/jpgraph_pie.php';
require_once 'MDB2.php';


// Verbindungsinformationen in einem Array sammeln
$dsn = array(
  "phptype"  => "mysql",
  "username" => "erika",
  "password" => "mustermann",
  "hostspec" => "dbs",
  "database" => "UniWeb",
);

// Herstellen der Verbindung
// und ggf. auf einen Fehler reagieren
$con =& MDB2::connect($dsn);
if (PEAR::isError($con))
  die($con->getMessage());


// Erstellen des SQL-Statements und ausfuehren
// gegen die Datenbank
$sql = "SELECT name, SUM(sws)
        FROM Vorlesungen, Professoren
        WHERE persnr = gelesenvon
        GROUP BY name";
$result = $con->query($sql);
if (PEAR::isError($result))
```

35 http://dbs.informatik.uos.de/media2mult/php/semester.php

```
  die($result->getMessage());

// alle Datensaetze in Arrays speichern
$i = 0;
while($row = $result->fetchRow()) {
  $namen[$i] = $row[0];
  $daten[$i] = $row[1];
  $i++;
}

// DB-Verbindung beenden
$result->free();
$con->disconnect();

// Graph aus Daten erzeugen
$graph = new PieGraph(600, 400, "auto");
$graph->SetShadow();

$graph->title->Set("Lehrbelastung der Professoren");
$graph->title->SetFont(FF_FONT1, FS_BOLD);

$pl = new PiePlot($daten);
$pl->SetLegends($namen);
$pl->SetCenter(0.4);

$graph->Add($pl);
$graph->Stroke();
?>
```

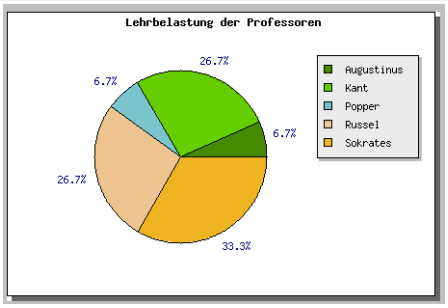


Abb. 94:
Dynamische Grafik, erzeugt von torte.php, aufgerufen von lehre.html

Aufruf des PHP-Scripts lehre.html[36]

Ausführliche Hinweise zu den Möglichkeiten von PHP und weitere Beispielprogramme sind unter http://www.php.net und http://www.selfphp.info zu finden.

36 http://dbs.informatik.uos.de/media2mult/php/lehre.html

11. Ruby on Rails

Hinweis: Dieses Kapitel wird derzeit überarbeitet. Die Inhalte sind weitestgehend fertiggestellt, es werden jedoch noch einige kosmetische Änderungen für die Integration in Media2Mult durchgeführt. Mit den angegebenen Beispielen lässt sich jedoch bereits arbeiten.

Ruby on Rails ist ein im Jahr 2004 erschienenes Framework, das speziell auf die Webentwicklung von Datenbankapplikationen zugeschnitten ist. Es basiert auf der objektorientierten Sprache Ruby und wurde mit dem Ziel entwickelt, häufig auftretende Muster bei der Entwicklung von datenbankgestützten Webapplikationen derart zu vereinfachen, dass ein schnelleres, agilers Vorgehen möglich wird. Dem folgenden Text liegt Rails in der Version 2.3. zugrunde.

Ruby

Ruby ist die Basis des Frameworks und die eigentliche Programmiersprache. Da die Syntax von Ruby beispielsweise für einen klassischen Java-Entwickler, etwas gewöhnungsbedürftig ist, soll an dieser Stelle ein kleiner Überblick über die Ruby Syntax gegeben werden.

Die "klassischen" Datentypen

- Zahlen: z.B. 42 sind Objekte vom Typ Fixnum
- Strings: z.B. "Strings" werden mit einfachen oder doppelten Hochkommata gekennzeichnet
- Symbole: z.B. :symbol sind Konstanten, die in Hashes häufig als Schlüssel zum Einsatz kommen

Collections

- Arrays: z.B. [] können so oder mit ihrem "Konstruktor" Array.new initalisiert werden, eine explizite Zuweisung von Inhalte ist möglich: [42,23]
- Hashes: z.B. {} können genau wie Arrays initalisiert werden und modellieren Schlüssel-Wert Beziehungen ähnlich der HashMap in Java, auch hier sind explizite Wertzuweisungen möglich: {:key => "value"}

Zugriff auf Elemente kann mit dem []-Operator genommen werden. Beispiel: mein_array[0] oder mein_hash[:key]

Methoden und Variablen

An Objekten können Methoden mit der aus Java bekannten Punktsyntax aufgerufen werden. Beispiel: [42,23].include?(42) Es folgt ein Beispiel für eine Methodendeklaration mit verwendeten Variablen. In Ruby sind Variablen dynamisch typisiert und müssen daher nicht mit ihrem Typ angekündigt werden.

```
def hallo_welt

  lokale_variable1 = 23
  lokale_variable2 = "Hallo Welt!"

  @instanz_variable = "In der Instanz sichtbar!"
  @@klassen_variable = "vgl. static in Java"
  $globale_variable = "ueberall sichtbar"

end
```

Besonderheiten

In Ruby werden meist Klammern um die Parameter eines Methodenaufrufs weggelassen. Hier sollte darauf geachtet werden, dass das Statement eindeutig bleibt. Beispiel: [42,23].include?(42) ist äquivalent zu [42,23].include? 42 Bei Hashes können weiterhin die geschweiften Klammern entfallen: hash_paramater_methode :key => "value" ist äquivalent zu hash_paramater_methode({:key => "value"})

Weiterhin ermöglicht es Ruby, ähnlich wie Javascript, Codeblöcke an Methoden zu übergeben. Dazu zwei Beispiele:

```
10.times do |i|
  puts i
end

[42,13].each do |i|
  puts i
end
```

Das erste Beispiel führt dabei den in dem mit do ... end Block beschriebenen Code 10 mal aus wobei i in jedem Durchlauf mit der derzeit betrachteten Zahl - also 1,2,3,... belegt ist. Im zweiten Beispiel wird mit der each-Methode über das Array iteriert - hier wird i in jedem Durchlauf das aktuell betrachtete Element sein.

Rails-Architektur

Das Rails Framework setzt verschiedene Konzepte um. Drei der wichtigsten sollen an dieser Stelle erläutert werden.

Model-View-Controller Design Pattern

Das Model-View-Controller Design Pattern, oder kurz MVC, ist ein bekanntes und häufig genutztes Programmierparadigma um Programmcode besser strukturieren zu können. In Rails steht dahinter die Idee, Code zur Anzeige von HTML-Seiten von Code zur Datenbankkommunikation zu trennen und beides mit Hilfe einer Kontrollstruktur miteinander zu verknüpfen.

Rails Models

Alle Models in Rails basieren auf der Klasse ActiveRecord. Damit stellt Rails eine vollständige Abstraktion von der eigentlichen Datenhaltung zur Verfügung, sodass es meist vermieden werden kann SQL-Queries explizit formulieren zu müssen. Die grundlegenden Datenbankoperationen für das Erstellen, Lesen, Ändern und Löschen (englisch Create, Read, Update, Delete - kurz CRUD) werden dabei automatisch generiert.

Rails Views

Die Views in Rails sind mit den Templates aus dem vorherigen Kapitel PHP zu vergleichen. Hier wird die eigentliche Webseiten-Struktur erstellt, in der die Rails Template-Tags verwendet werden können, um später beim Aufruf durch einen Controller durch Klartext ersetzt zu werden.

Rails Controller

Controller in Rails enthalten alles Kontrollstrukturen für die Applikation und sind damit die erste Anlaufstelle beim Aufruf einer Rails-generierten Seite. Sie sind dafür verantwortlich mit den Models zu kommunizieren und

die entsprechenden Daten an die Views weiterzuleiten. Die genaue Architektur wird im folgenden anhand eines Beispiels erläutert.

Convention Over Configuration

Dieses Konzept besagt, dass es sinnvoller ist sich gewissen Konventionen zu beugen, um bestimmte Funktionalität zu erlagen, als viel Zeit damit zu verbringen ebendiese Funktionalität selbst konfigurieren zu müssen. Rails hat eine Vielzahl von Konventionen, wobei viele dafür mitverantwortlich sind, dass das Entwickeln von Applikationen in Rails schneller geht, als würde man explizites Verhalten konfigurieren müssen. Einige Konventionsbeispiele werden im folgenden Beispiel behandelt.

Don't Repeat Yourself

Auch dieses Konzept soll dafür sorgen, flexibler und schneller entwickeln zu können. Die Idee dahinter ist, dass Code, der eine bestimmte Aufgabe erfüllt oder etwas definiert nicht mehrfach geschrieben werden sollte. Dies schafft insbesondere beim späteren Hinzufügen von Funktionalität, dem Ändern oder dem Beseitigen von Fehlern, den Vorteil dies nur an einer Stelle tun zu müssen.

Eine einfache Rails Applikation

Ist Rails auf dem Entwicklungsrechner installiert, erstellt man eine neue Applikation mit dem Aufruf von

rails -d mysql APPLIKATIONSNAME

Das Hilfsprogramm generiert nun eine komplette Ordnerstruktur für Models, Views und Controller sowie Hilfsscripte um beispielsweise eine Entwicklungskonsole anzuzeigen oder einen Entwicklungsserver zu starten. Im Ordner config sollten zunächst in der Datei database.yml die Zugangsdaten für den zu verwendenden Datenbankserver angegeben werden. Dazu wird die Rubrik development verwendet, die die Rails Standardumgebung repräsentiert.

Danach kann damit begonnen werden, nach dem jeweiligen Konzept, Models zu synthetisieren die die Entitätstypen der Datenbank repräsentieren. Ein Model kann mit einem Hilfsscript im Unterordner script, also dem Befehl

script/generate model ENTITÄTSNAME

generiert werden. Sind die Attribute eines Entitätstypen bereits bekannt, kann mit Hilfe des selben Generator-Skripts eine komplette Grundstruktur für den Entitätstypen generiert werden, ein sogenanntes Scaffold.

script/generate scaffold ENTITÄTSNAME attribut1:typ attribut2:typ ...

Ein Scaffold besteht dabei nicht nur aus dem entsprechenden Model, das den Entitätstypen repräsentiert sondern auch aus einem Controller, der die Grundlegenden CRUD-Operationen auch auf Basis einer Weboberfläche verfügbar macht. Dazu werden auch entsprechende Views generiert. Weiterhin wird eine sogenannte Migration erzeugt, in der in Ruby bzw. Rails-Syntax die Schemadefinition für die Datenbank formuliert ist. Dort sind auch die entsprechend angegebenen Attribute spezifiziert, sodass mit dem Befehl

rake db:migrate

in der zuvor festgelegten Datenbank die zugehörige Tabelle für den jeweiligen Entitätstypen mit den entsprechenden Attributen angelegt wird. Rails übersetzt dabei die Befehle in der Migration, die im Unterordner db/migrate zu finden sind, in einen für die spezifizierte Datenbank verträglichen SQL-Code.

Mit Hilfe des integierten Entwicklungsservers kann nun auf die bestehende Rails Applikation zugegriffen werden.

script/server

startet diesen auf dem Standard-Port 3000. Die Rails Konventionen geben dabei vor, dass das Schema eines URL-Aufrufs wie folgt definiert ist:

http://localhost:3000/controller/action

Dabei ist "controller" der Name eines z.B. durch den Scaffold-Befehl generierten Controller und Action eine Methode in eben diesem.

Beispiel

Es sei eine Rails Applikation bereits angelegt worden. Nun sollen die Entitätstypen Raum und Professor modelliert werden. Zwischen beiden soll eine 1:n Relation bestehen, wobei jedem Professor genau ein Raum zugeordnet sein soll. Das Generieren der Scaffolds sieht dann so aus:

script/generate scaffold Room room_no:string script/generate scaffold Professor name:string birthday:date room:references

Beide Befehle generieren für ihre Entitätstypen wie oben beschrieben Models, Views, Controller und jeweils eine Migration, die mit rake db:migrate in die Datenbank übernommen werden kann. Der Aufruf von

localhost:3000/rooms

führt nach Start des Servers auf eine View, die es ermöglichst webgestützt Räume anzulegen, zu löschen, zu bearbeiten und anzuzeigen.

localhost:3000/rooms/new

Ruft dabei die Methode new der Klasse RoomsController auf, die wiederum die View new.html.erb im Ordner app/views/rooms zur Darstellung benutzt. Die Namensgleichheit ist dabei Teil der Rails-Konventionen.

Datenbankabstraktion mit Rails

Ein wesentlicher Teil der Rails Architektur ist die Abstraktion von der eigentlichen Datenhaltung. Wie oben schon erwähnt müssen für die einfache Verwendung einige Konventionen in Bezug auf die Models eingehalten werden.

Model-Klassen sind im Singular definiert und im Mixed-Case geschrieben. Beispiele:

```
class Professor << ActiveRecord::Base

end


class GoodProfessor << ActiveRecord::Base

end
```

Die zugehörigen Tabellen sollten klein geschrieben, im (englischen) Plural und mit Unterstrichen benannt sein. Analog zu oben also professors und good_professors. Die Konvention für Schlüssel und Fremdschlüssel sieht dabei vor, dass jeder Entitätstyp mit dem synthetischen Schlüssel id versehen wird, und jeder Fremdschlüssel den Namen des verknüpften Entitätstypen Unterstrich id erhält. Für obiges Beispiel würde also der Entitätstyp Professor im relationalen Schema einen Fremdschlüssel mit dem Namen room_id erhalten.

Auf diese Weise kann Rails ein automatisches Mapping zwischen den Models und den Tabellen in der Datenbank durchführen. Dies ermöglicht einen komfortablen Umgang mit den Daten. Getestet werden kann dies z.B. mit der Rails-Entwicklungskonsole, die mit dem Befehl

script/console

gestartet wird. Hier können zeilenweise Ruby bzw. Rails Statements abgesetzt werden. Ein Beispiel:

```
Loading development environment (Rails 2.3.2)
>> all_profs = Professor.all
=> []
>> my_professor = Professor.new
=> #<Professor id: nil, name: nil, birth: nil, room_id: nil, created_at: nil, updated_at: nil>
>> my_professor.name = "Sokrates"
=> "Sokrates"
>> my_professor.birth = 2500.year.ago
=> Thu, 22 Jun -0491 13:15:34 UTC +00:00
>> my_professor.save
=> true
>> all_profs = Professor.all
=> [#<Professor id: 1, name: "Sokrates", birth: nil, room_id: nil, created_at: "2009-06-17
13:15:48", updated_at: "2009-06-17 13:15:48">]
>> my_professor = Professor.find(1)
=> #<Professor id: 1, name: "Sokrates", birth: nil, room_id: nil, created_at: "2009-06-17
13:15:48", updated_at: "2009-06-17 13:15:48">
>> my_room = Room.new
=> #<Room id: nil, room_nr: nil, name: nil, created_at: nil, updated_at: nil>
>> my_room.room_nr = "31/449a"
=> "31/449a"
>> my_room.save
=> true
>> my_professor.room = my_room
=> #<Room id: 1, room_nr: 31, name: nil, created_at: "2009-06-17 13:16:35", updated_at: "2009-06-17
 13:16:35">
>> my_professor.room
=> #<Room id: 1, room_nr: 31, name: nil, created_at: "2009-06-17 13:16:35", updated_at: "2009-06-17
 13:16:35">
>> my_room = Room.first
=> #<Room id: 1, room_nr: 31, name: nil, created_at: "2009-06-17 13:16:35", updated_at: "2009-06-17
 13:16:35">
>> my_room.professors
=> [#<Professor id: 1, name: "Sokrates", birth: nil, room_id: 1, created_at: "2009-06-17 13:15:48",
 updated_at: "2009-06-17 13:17:07">]
```

Beim Zugriff auf die speziellen Methoden der Model-Klasse, wie all, new oder find erzeugt Rails automatisch eine entsprechende Query und setzt diese an die Datenbank ab, die Ergebnismenge wird geparsed und in entsprechende Rails Models übersetzt. Damit die Befehle my_room.professors oder my_professor.room funktionieren, müssen in den Models die Typen der Beziehungen definiert sein. Dazu stehen verschiedene Direkten zur Verfügung, die die entsprechenden Relationen modellieren. Für obiges Beispiel müssten die Klassen so aussehen:

```
class Professor << ActiveRecord::Base
  belongs_to :room
end


class Room << ActiveRecord::Base
  has_many :professors
end
```

Für die Modellierung von n:m Beziehungen und solche mit Attributen stehen weitere Direkten zur Verfügung. Sind diese korrekt angegeben, synthetisiert Rails entsprechende Getter- und Setter-Methoden für den Zugriff auf die Verknüpften Entitäten, die als Array von Rails-Objekten zurückgeliefert werden.

Eigene Views

Selbstverständlich sollte die Applikationsentwicklung nicht nach der Generierung der Scaffolds beendet sein. Für die Darstellung eigener Views können entweder die vorhandenen Controller modifiziert oder neue erstellt werden.

script/generate controller CONTROLLERNAME

erzeugt einen solchen. Soll beispielsweise eine Liste von Räumen mit den jeweils darin befindlichen Professoren in einem neuen Controller umgesetzt werden, könnte der Aufruf

script/generate controller ProfRooms

lauten. Im Controller sollte nun eine Methode (in Rails auch Action genannt) spezifiert werden:

```
class ProfRoomsController < ApplicationController

  def list
    @all_rooms = Room.all
  end

end
```

Für die Methode muss nun noch ein geeignetes Template im Ordner views/profrooms mit dem Namen der Methode definiert werden:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <title>list.html</title>
</head>
<body>

<% @all_rooms.each do |room| %>
<h1><%= room.name %></h1>
  <ul>
  <% room.professors.each do |prof| %>
    <li><%= prof.name %></li>
  <% end %>
  </ul>
<% end %>

</body>
</html>
```

In der View werden nun in Ruby geschriebene Befehle innerhalb der Template Tags ausgeführt und in Klartext ausgewertet, sodass beim Aufruf von

localhost:3000/profrooms/list

eben genau dieses Template, mit den entsprechenden Werten gefüllt, ausgeliefert wird. Zu beachten ist dabei, dass die Variablen, die aus dem Controller an das Template übergeben werden sollen, Instanzvariablen des Controllers, also mit @ gekennzeichnet, sein müssen.

Weitere Informationen

Weitere Hinweise zur Verwendung von Rails und Ruby liefern die jeweiligen API-Dokumentationen und die Webseite von Ruby on Rails, die Einstiegspunkt zu vielen Anleitungen und Ressourcen ist.

- Ruby on Rails Webseite[37]
- Rails Framework Dokumentation[38]
- Ruby Dokumentation[39]

37 http://www.rubyonrails.org/
38 http://api.rubyonrails.org/
39 http://corelib.rubyonrails.org/

12. Relationale Entwurfstheorie

12.1 Funktionale Abhängigkeiten

Gegeben sei ein Relationenschema $\mathcal{R}$ mit einer Ausprägung *R*. Eine *funktionale Abhängigkeit* (engl. *functional dependency*) stellt eine Bedingung an die möglichen gültigen Ausprägungen des Datenbankschemas dar. Eine funktionale Abhängigkeit, oft abgekürzt als FD, wird dargestellt als

$$\alpha \rightarrow \beta$$

Die griechischen Buchstaben α und β repräsentieren Mengen von Attributen. Es sind nur solche Ausprägungen zulässig, für die gilt:

$$\vee\ r, t \in R:\ r.\alpha = t.\alpha \Rightarrow r.\beta = t.\beta$$

D. h., wenn zwei Tupel gleiche Werte für alle Attribute in α haben, dann müssen auch ihre β-Werte übereinstimmen. Anders ausgedrückt: Die α-Werte bestimmen eindeutig die β-Werte; die β-Werte sind funktional abhängig von den α-Werten.

Die nächste Tabelle zeigt ein Relationenschema $\mathcal{R}$ über der Attributmenge {*A, B, C, D*}.

R			
A	*B*	*C*	*D*
a_4	b_2	c_4	d_3
a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_2	c_3	d_2
a_3	b_2	c_4	d_3

Aus der momentanen Ausprägung lassen sich z. B. die funktionalen Abhängigkeiten $\{A\} \rightarrow \{B\}, \{A\} \rightarrow \{C\}, \{C, D\} \rightarrow \{B\}$ erkennen, hingegen gilt nicht $\{B\} \rightarrow \{C\}$.

Ob diese Abhängigkeiten vom Designer der Relation als semantische Konsistenzbedingung verlangt wurden, läst sich durch Inspektion der Tabelle allerdings nicht feststellen.

Statt $\{C, D\} \rightarrow \{B\}$ schreiben wir auch $CD \rightarrow B$. Statt $\alpha \cup \beta$ schreiben wir auch $\alpha\beta$.

Ein einfacher Algorithmus zum Überprüfen einer (vermuteten) funktionalen Abhängigkeit $\alpha \rightarrow \beta$ in der Relation R lautet:

1. sortiere *R* nach α-Werten
2. falls alle Gruppen bestehend aus Tupeln mit gleichen α-Werten auch gleiche β-Werte aufweisen, dann gilt $\alpha \rightarrow \beta$, sonst nicht.

12.2 Schlüssel

In dem Relationenschema $\mathcal{R}$ ist $\alpha \subseteq \mathcal{R}$ ein *Superschlüssel*, falls gilt

$$\alpha \rightarrow \mathcal{R}$$

Der Begriff Superschlüssel besagt, das alle Attribute von α abhängen aber noch nichts darüber bekannt ist, ob α eine minimale Menge von Attributen enthält.

Wir sagen: β ist *voll funktional abhängig* von α, in Zeichen $\alpha \dot{\rightarrow} \beta$, falls gilt

1. $\alpha \rightarrow \beta$
2. $\forall A \in \alpha : \alpha - \{A\} \not\rightarrow \beta$

In diesem Falle heißt α*Schlüsselkandidat*. Einer der Schlüsselkandidaten wird als *Primärschlüssel* ausgezeichnet.

Folgende Tabelle zeigt die Relation *Städte*:

Städte			
Name	BLand	Vorwahl	EW
Frankfurt	Hessen	069	650000
Frankfurt	Brandenburg	0335	84000
München	Bayern	089	1200000
Passau	Bayern	0851	50000
...	...	...	...

Offenbar gibt es zwei Schlüsselkandidaten:

1. {Name, BLand}
2. {Name, Vorwahl}

12.3 Bestimmung funktionaler Abhängigkeiten

Wir betrachten folgendes Relationenschema:

ProfessorenAdr :{[PersNr, Name, Rang, Raum,
Ort, Strase, PLZ, Vorwahl, BLand, Landesregierung]}

Hierbei sei *Ort* der eindeutige Erstwohnsitz des Professors, die *Landesregierung* sei die eindeutige Partei des Ministerpräsidenten, *BLand* sei der Name des Bundeslandes, eine Postleitzahl (*PLZ*) ändere sich nicht innerhalb einer Strase, Städte und Strasen gehen nicht über Bundesgrenzen hinweg.

Folgende Abhängigkeiten gelten:

1. {PersNr} $\rightarrow$ {PersNr, Name, Rang, Raum,
   Ort, Strase, PLZ, Vorwahl, BLand, EW, Landesregierung}
2. {Ort, BLand} $\rightarrow$ {Vorwahl}
3. {PLZ} $\rightarrow$ {BLand, Ort}
4. {Ort, BLand, Strase} $\rightarrow$ {PLZ}
5. {BLand} $\rightarrow$ {Landesregierung}
6. {Raum} $\rightarrow$ {PersNr}

Hieraus können weitere Abhängigkeiten abgeleitet werden:

7. {Raum} $\rightarrow$ {PersNr, Name, Rang, Raum,
   Ort, Strase, PLZ, Vorwahl, BLand, Landesregierung}
8. {PLZ} $\rightarrow$ {Landesregierung}

Bei einer gegebenen Menge *F* von funktionalen Abhängigkeiten über der Attributmenge U interessiert uns die Menge F^+ aller aus F ableitbaren funktionalen Abhängigkeiten, auch genannt die *Hülle* (engl. *closure*) von F.

Zur Bestimmung der Hülle reichen folgende *Inferenzregeln*, genannt *Armstrong Axiome*, aus:

- Reflexivität: Aus $\beta \subseteq \alpha$ folgt: $\alpha \rightarrow \beta$
- Verstärkung: Aus $\alpha \rightarrow \beta$ folgt: $\alpha\gamma \rightarrow \beta\gamma$ für $\gamma \subseteq U$
- Transitivität: Aus $\alpha \rightarrow \beta$ und $\beta \rightarrow \gamma$ folgt: $\alpha \rightarrow \gamma$

Die Armstrong-Axiome sind *sound* (korrekt) und *complete* (vollständig). Korrekt bedeutet, das nur solche FDs abgeleitet werden, die von jeder Ausprägung erfüllt sind, für die F erfüllt ist. Vollständig bedeutet, das sich alle Abhängigkeiten ableiten lassen, die durch F logisch impliziert werden. Weitere Axiome lassen sich ableiten:

- Vereinigung: Aus $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$ folgt: $\alpha \rightarrow \beta\gamma$
- Dekomposition: Aus $\alpha \rightarrow \beta\gamma$ folgt: $\alpha \rightarrow \beta$ und $\alpha \rightarrow \gamma$
- Pseudotransitivität: Aus $\alpha \rightarrow \beta$ und $\gamma\beta \rightarrow \delta$ folgt $\alpha\gamma \rightarrow \delta$

Wir wollen zeigen: {*PLZ*} $\rightarrow$ {*Landesregierung*} läst sich aus den FDs 1-6 für das Relationenschema *ProfessorenAdr* herleiten:

- {PLZ} $\rightarrow$ {BLand} (Dekomposition von FD Nr.3)
- {BLand} $\rightarrow$ {Landesregierung} (FD Nr.5)
- {PLZ} $\rightarrow$ {Landesregierung} (Transitivität)

Oft ist man an der Menge von Attributen α^+ interessiert, die von α gemäs der Menge *F* von FDs funktional bestimmt werden:

$$\alpha^+ := \{\beta \subseteq U \mid \alpha \rightarrow \beta \in F^+\}$$

Es gilt der Satz: $\alpha \rightarrow \beta$ folgt aus Armstrongaxiomen genau dann wenn $\beta \in \alpha^+$.

Die Menge α^+ kann aus einer Menge *F* von FDs und einer Menge von Attributen α wie folgt bestimmt werden:

$$X^0 := \alpha$$
$$X^{i+1} := X^i \cup \gamma \quad falls\ \beta \rightarrow \gamma \in F \wedge \beta \subseteq X^i$$

D. h. von einer Abhängigkeit $\beta \rightarrow \gamma$, deren linke Seite schon in der Lösungsmenge enthalten ist, wird die rechte Seite hinzugefügt. Der Algorithmus wird beendet, wenn keine Veränderung mehr zu erzielen ist, d. h. wenn gilt: $X^{i+1} = X^i$.

Beispiel :

Sei *U* = $\{A, B, C, D, E, G\}$

Sei *F* = $\{AB \rightarrow C, C \rightarrow A, BC \rightarrow D, ACD \rightarrow B,$
$D \rightarrow EG, BE \rightarrow C, CG \rightarrow BD, CE \rightarrow AG\}$

Sei X = $\{B, D\}$

X^0 = BD

X^1 = $BDEG$

X^2 = $BCDEG$

X^3 = $ABCDEG$ = X^4, Abbruch.

Also: $(BD)^+$ = $ABCDEG$

Zwei Mengen *F* und *G* von funktionalen Abhängigkeiten heißen genau dann *äquivalent* (in Zeichen $F \equiv G$), wenn ihre Hüllen gleich sind:

$$F \equiv G \Leftrightarrow F^+ = G^+$$

Zum Testen, ob $F^+ = G^+$, muss für jede Abhängigkeit $\alpha \rightarrow \beta \in F$ überprüft werden, ob gilt: $\alpha \rightarrow \beta \in G^+$, d. h. $\beta \subseteq \alpha^+$. Analog mus für die Abhängigkeiten $\gamma \rightarrow \delta \in G$ verfahren werden.

Zu einer gegebenen Menge F von FDs interessiert oft eine kleinstmögliche äquivalente Menge von FDs.

Eine Menge von funktionalen Abhängigkeiten heißt minimal $\Leftrightarrow$

1. Jede rechte Seite hat nur ein Attribut.
2. Weglassen einer Abhängigkeit aus F verändert F^+.
3. Weglassen eines Attributs in der linken Seite verändert F^+.

Konstruktion der minimalen Abhängigkeitsmenge geschieht durch Aufsplitten der rechten Seiten und durch probeweises Entfernen von Regeln bzw. von Attributen auf der linken Seite.

Beispiel :

Sei U = $\{ A, B, C, D, E, G\}$

Sei F = $\{$ $AB \rightarrow C$, $D \rightarrow EG$

$C \rightarrow A$, $BE \rightarrow C$,

$BC \rightarrow D$, $CG \rightarrow BD$,

$ACD \rightarrow B$, $CE \rightarrow AG$ $\}$

Aufspalten der rechten Seiten liefert

$AB \rightarrow C$

$C \rightarrow A$

$BC \rightarrow D$

$ACD \rightarrow B$

$D \rightarrow E$

$D \rightarrow G$

$BE \rightarrow C$

$CG \rightarrow B$

$CG \rightarrow D$

$CE \rightarrow A$

$CE \rightarrow G$

Regel	$CE \rightarrow A$	ist redundant wegen	$C \rightarrow A$
	$CG \rightarrow B$	ist redundant wegen	$CG \rightarrow D$
			$C \rightarrow A$
			$ACD \rightarrow B$
Regel	$ACD \rightarrow B$	kann gekürzt werden zu	$CD \rightarrow B$, wegen $C \rightarrow A$

12.4 Schlechte Relationenschemata

Als Beispiel für einen schlechten Entwurf zeigen wir die Relation *ProfVorl*:

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	C4	226	5041	Ethik	4
2125	Sokrates	C4	226	5049	Mäutik	2
2125	Sokrates	C4	226	4052	Logik	4
...	...	...	...	...	...	...
2132	Popper	C3	52	5259	Der Wiener Kreis	2
2137	Kant	C4	7	4630	Die 3 Kritiken	4

Folgende Anomalien treten auf:

- Update-Anomalie :
  Angaben zu den Räumen eines Professors müssen mehrfach gehalten werden.
- Insert-Anomalie :
  Ein Professor kann nur mit Vorlesung eingetragen werden (oder es entstehen NULL-Werte).
- Delete-Anomalie :
  Das Entfernen der letzten Vorlesung eines Professors entfernt auch den Professor (oder es müssen NULL-Werte gesetzt werden).

12.5 Zerlegung von Relationen

Unter *Normalisierung* verstehen wir die Zerlegung eines Relationenschemas $\mathcal{R}$ in die Relationenschemata $\mathcal{R}_1, \mathcal{R}_2, \ldots R_n$, die jeweils nur eine Teilmenge der Attribute von $\mathcal{R}$ aufweisen, d. h. $\mathcal{R}_i \subseteq \mathcal{R}$. Verlangt werden hierbei

- Verlustlosigkeit:

Die in der ursprünglichen Ausprägung *R* des Schemas $\mathcal{R}$ enthaltenen Informationen müssen aus den Ausprägungen $R_1, \ldots, R_n$ der neuen Schemata $\mathcal{R}_1, \mathcal{R}_2, \ldots \mathcal{R}_n$ rekonstruierbar sein.

- Abhängigkeitserhaltung:
  Die für $\mathcal{R}$ geltenden funktionalen Abhängigkeiten müssen auf die Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ übertragbar sein.

Wir betrachten die Zerlegung in zwei Relationenschemata. Dafür muss gelten $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$. Für eine Ausprägung *R* von $\mathcal{R}$ definieren wir die Ausprägung R_1 von $\mathcal{R}_1$ und R_2 von $\mathcal{R}_2$ wie folgt:

$$R_1 := \Pi_{\mathcal{R}_1}(R)$$
$$R_2 := \Pi_{\mathcal{R}_2}(R)$$

Eine Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1$ und $\mathcal{R}_2$ heist *verlustlos*, falls für jede gültige Ausprägung *R* von $\mathcal{R}$ gilt:

$$R = R_1 \bowtie R_2$$

Es folgt eine Relation *Biertrinker*, die in zwei Tabellen zerlegt wurde. Der aus den Zerlegungen gebildete natürliche Verbund weicht vom Original ab. Die zusätzlichen Tupel (kursiv gesetzt) verursachen einen Informationsverlust.

Biertrinker		
Kneipe	Gast	Bier
Stiefel	Wacker	Pils
Stiefel	Sorglos	Hefeweizen
Zwiebel	Wacker	Hefeweizen

Besucht	
Kneipe	Gast
Stiefel	Wacker
Stiefel	Sorglos
Zwiebel	Wacker

Trinkt	
Gast	Bier
Wacker	Pils
Sorglos	Hefeweizen
Wacker	Hefeweizen

Besucht ⋈ Trinkt		
Kneipe	Gast	Pils
Stiefel	Wacker	Pils
Stiefel	*Wacker*	*Hefeweizen*
Stiefel	Sorglos	Hefeweizen
Zwiebel	*Wacker*	*Pils*
Zwiebel	Wacker	Hefeweizen

Eine Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1$, ..., $\mathcal{R}_n$ heist $abhängigkeitsbewahrend$ (auch genannt $hüllentreu$) falls die Menge der ursprünglichen funktionalen Abhängigkeiten äquivalent ist zur Vereinigung der funktionalen Abhängigkeiten jeweils eingeschränkt auf eine Zerlegungsrelation, d. h.

- $F_{\mathcal{R}} \equiv (F_{\mathcal{R}_1} \cup \ldots \cup F_{\mathcal{R}_n})$ bzw.
- $F_{\mathcal{R}}^{+} = (F_{\mathcal{R}_1} \cup \ldots \cup F_{\mathcal{R}_n})^{+}$

Es folgt eine Relation *PLZverzeichnis*, die in zwei Tabellen zerlegt wurde. Fettgedruckt sind die jeweiligen Schlüssel.

PLZverzeichnis			
Ort	BLand	Strase	PLZ
Frankfurt	Hessen	Goethestrase	60313
Frankfurt	Hessen	Galgenstrase	60437
Frankfurt	Brandenburg	Goethestrase	15234

Strasen	
PLZ	Strase
15234	Goethestrase
60313	Goethestrase
60437	Galgenstrase

Orte		
Ort	BLand	PLZ
Frankfurt	Hessen	60313
Frankfurt	Hessen	60437
Frankfurt	Brandenburg	15234

Es sollen die folgenden funktionalen Abhängigkeiten gelten:

- {PLZ} $\rightarrow$ {Ort, BLand}
- {Strase, Ort, BLand} $\rightarrow$ {PLZ}

Die Zerlegung ist verlustlos, da PLZ das einzige gemeinsame Attribut ist und {PLZ} $\rightarrow$ {Ort, BLand} gilt.

Die funktionale Abhängigkeit {Strase, Ort, BLand} $\rightarrow$ {PLZ} ist jedoch keiner der beiden Relationen zuzuordnen, so das diese Zerlegung nicht abhängigkeitserhaltend ist.

Folgende Auswirkung ergibt sich: Der Schlüssel von *Straßen* ist {PLZ, Strase} und erlaubt das Hinzufügen des Tupels [15235, Goethestrase].

Der Schlüssel von $Orte$ ist {PLZ} und erlaubt das Hinzufügen des Tupels [Frankfurt, Brandenburg, 15235]. Beide Relationen sind lokal konsistent, aber nach einem Join wird die Verletzung der Bedingung {Straße, Ort, BLand} $\rightarrow$ {PLZ} entdeckt.

12.6 Erste Normalform

Ein Relationenschema $\mathcal{R}$ ist in erster Normalform, wenn alle Attribute atomare Wertebereiche haben. Verboten sind daher zusammengesetzte oder mengenwertige Domänen.

Zum Beispiel müste die Relation

Eltern		
Vater	Mutter	Kinder
Johann	Martha	{Else, Lucia}
Johann	Maria	{Theo, Josef}
Heinz	Martha	{Cleo}

"flachgeklopft" werden zur Relation

Eltern		
Vater	Mutter	Kind
Johann	Martha	Else
Johann	Martha	Lucia
Johann	Maria	Theo
Johann	Maria	Josef
Heinz	Martha	Cleo

12.7 Zweite Normalform

Ein Attribut heist $Primärattribut$, wenn es in mindestens einem Schlüsselkandidaten vorkommt, andernfalls heißt es Nichtprimärattribut.

Ein Relationenschema $\mathcal{R}$ ist in zweiter Normalform falls gilt:

- $\mathcal{R}$ ist in der ersten Normalform
- Jedes Nichtprimär-Attribut $A \in \mathcal{R}$ ist voll funktional abhängig von jedem Schlüsselkandidaten.

Seien also $\kappa_1, \ldots, \kappa_n$ die Schlüsselkandidaten in einer Menge F von FDs. Sei $A \in \mathcal{R} - (\kappa_1 \cup \ldots \cup \kappa_n)$ ein $Nichtprimärattribut$. Dann mus für $1 \leq j \leq n$ gelten:

$$\kappa_j \stackrel{\cdot}{\rightarrow} A \in F^+$$

Folgende Tabelle verletzt offenbar diese Bedingung:

StudentenBelegung			
MatrNr	**VorlNr**	Name	Semester
26120	5001	Fichte	10
27550	5001	Schopenhauer	6
27550	4052	Schopenhauer	6
28106	5041	Carnap	3
28106	5052	Carnap	3
28106	5216	Carnap	3
28106	5259	Carnap	3
...	...	...	...

Abbildung 95 zeigt die funktionalen Abhängigkeiten der Relation *StudentenBelegung*. Offenbar ist diese Relation nicht in der zweiten Normalform, denn *Name* ist nicht voll funktional abhängig vom Schlüsselkandidaten {*MatrNr, VorlNr*}, weil der Name alleine von der Matrikelnummer abhängt.

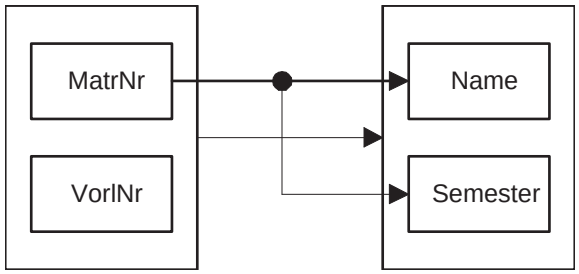


Abb. 95: Graphische Darstellung der funktionalen Abhängigkeiten von StudentenBelegung

Als weiteres Beispiel betrachten wir die Relation

Hörsaal : { [Vorlesung, Dozent, Termin, Raum] }

Eine mögliche Ausprägung könnte sein:

Vorlesung	Dozent	Termin	Raum
Backen ohne Fett	Kant	Mo, 10:15	32/102
Selber Atmen	Sokrates	Mo, 14:15	31/449
Selber Atmen	Sokrates	Di, 14:15	31/449
Schneller Beten	Sokrates	Fr, 10:15	31/449

Die Schlüsselkandidaten lauten:

- {Vorlesung, Termin}
- {Dozent, Termin}
- {Raum, Termin}

Alle Attribute kommen in mindestens einem Schlüsselkandidaten vor. Also gibt es keine Nichtprimärattribute, also ist die Relation in zweiter Normalform.

12.8 Dritte Normalform

Wir betrachten die Relation

Student : {[MatrNr, Name, Fachbereich, Dekan]}

Eine mögliche Ausprägung könnte sein:

MatrNr	Name	Fachbereich	Dekan
29555	Feuerbach	6	Matthies
27550	Schopenhauer	6	Matthies
26120	Fichte	4	Kapphan
25403	Jonas	6	Matthies
28106	Carnap	7	Weingarten

Offenbar ist *Student* in der zweiten Normalform, denn die Nichtprimärattribute *Name*, *Fachbereich* und *Dekan* hängen voll funktional vom einzigen Schlüsselkandidat *MatrNr* ab.

Allerdings bestehen unschöne Abhängigkeiten zwischen den Nichtprimärattributen, z. B. hängt *Dekan* vom *Fachbereich* ab. Dies bedeutet, das bei einem Dekanswechsel mehrere Tupel geändert werden müssen.

Seien X, Y, Z Mengen von Attributen eines Relationenschemas $\mathcal{R}$ mit Attributmenge U. Z heist *transitiv abhängig* von X, falls gilt

$X \cap Z = \emptyset$
$\exists\, Y \subset U : X \cap Y = \emptyset, Y \cap Z = \emptyset$
$X \rightarrow Y \rightarrow Z, Y \not\rightarrow X$

Zum Beispiel ist in der Relation *Student* das Attribut *Dekan* transitiv abhängig von *MatrNr*:

$$\mathrm{MatrNr} \xrightarrow{\not\subset} \mathit{Fachbereich} \rightarrow \mathit{Dekan}$$

Ein Relationenschema $\mathcal{R}$ ist in dritter Normalform falls gilt

- $\mathcal{R}$ ist in zweiter Normalform
- Jedes Nichtprimärattribut ist nicht-transitiv abhängig von jedem Schlüsselkandidaten.

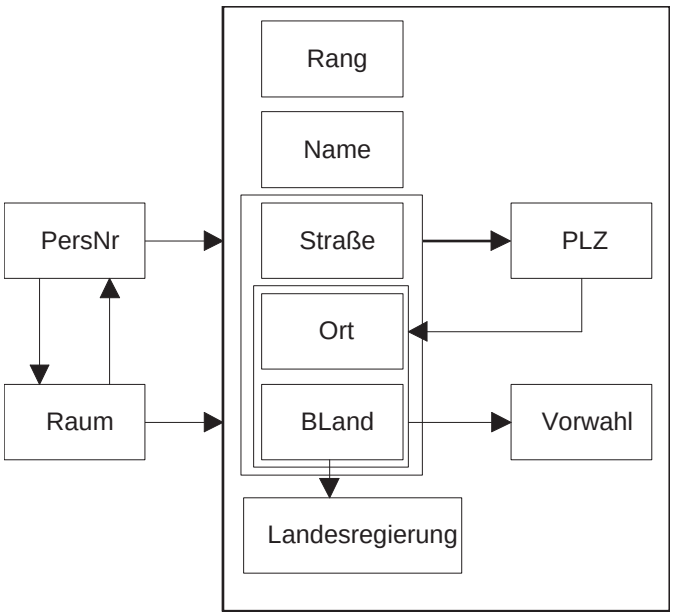


Abb. 96:
Graphische Darstellung der funktionalen Abhängigkeiten von ProfessorenAdr

Als Beispiel betrachten wir die bereits bekannte Relation

ProfessorenAdr : {[PersNr, Name, Rang, Raum, Ort, StrasePLZ, Vorwahl, BLand, Landesregierung]}

Abbildung 96 zeigt die funktionalen Abhängigkeiten in der graphischen Darstellung. Offenbar ist die Relation nicht in der dritten Normalform, da das Nichtprimärattribut *Vorwahl* transitiv abhängig vom Schlüsselkandidaten *PersNr* ist:

$$\mathit{PersNr} \xrightarrow{\not\subset} \{\mathit{Ort},\ \mathit{BLand}\} \rightarrow \mathit{Vorwahl}$$

12.9 Boyce-Codd Normalform

Die Boyce-Codd Normalform (BCNF) stellt nochmals eine Verschärfung dar. Ein Relationenschema $\mathcal{R}$ mit funktionalen Abhängigkeiten F ist in BCNF, falls für jede funktionale Abhängigkeit $\alpha \rightarrow \beta$ mindestens eine der folgenden beiden Bedingungen gilt:

- $\beta \subseteq \alpha$, d.h. die Abhängigkeit ist trivial oder
- α ist ein Superschlüssel von $\mathcal{R}$

Betrachten wir die folgende Relation *Städte*:

Städte: {[Ort, BLand, Ministerpräsident, EW]}

Städte			
Ort	BLand	Ministerpräsident	EW
Frankfurt	Hessen	Koch	660.000
Frankfurt	Brandenburg	Platzek	70.000
Bonn	NRW	Steinbrück	300.000
Lotte	NRW	Steinbrück	14.000
...	...	...	...

Offenbar gibt es die folgenden funktionalen Abhängigkeiten

fd_1 : {Ort, Bland} $\rightarrow$ {EW}
fd_2 : {BLand} $\rightarrow$ {Ministerpräsident}
fd_3 : {Ministerpräsident} $\rightarrow$ {Bland}

Daraus ergeben sich die folgenden beiden Schlüsselkandidaten

- κ_1 = {Ort, Bland}
- κ_2 = {Ort, Ministerpräsident}

Städte ist in dritter Normalform, denn das einzige Nichtprimärattribut *EW* ist nicht-transitiv abhängig von beiden Schlüsselkandidaten.

Städte ist jedoch nicht in Boyce-Codd Normalform, da die linken Seiten der funktionalen Abhängigkeiten fd_2 und fd_3 keine Superschlüssel sind.

Obacht: Um Relationen in dritter Normalform oder Boyce-Codd Normalform zu erhalten, ist häufig eine starke Aufsplittung erforderlich. Dies führt natürlich zu erhöhtem Aufwand bei Queries, da ggf. mehrere Verbundoperationen erforderlich werden.

13. Transaktionsverwaltung

13.1 Begriffe

Unter einer $Transaktion$ versteht man die Bündelung mehrerer Datenbankoperationen zu einer Einheit. Verwendet werden Transaktionen im Zusammenhang mit

- **Mehrbenutzersynchronisation** (Koordinierung von mehreren Benutzerprozessen),
- **Recovery** (Behebung von Fehlersituationen).

Die Folge der Operationen (lesen, ändern, einfügen, löschen) soll die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführen. Als Beispiel betrachten wir die Überweisung von 50,- Euro von Konto A nach Konto B:

```
read(A, a);
a := a - 50;
write(A, a);
read(B, b);
b := b + 50;
write(B, b);
```

Offenbar sollen entweder alle oder keine Befehle der Transaktion ausgeführt werden.

13.2 Operationen auf Transaktionsebene

Zur Steuerung der Transaktionsverwaltung sind folgende Operationen notwendig:

- **begin of transaction (BOT):** Markiert den Anfang einer Transaktion.
- **commit:** Markiert das Ende einer Transaktion. Alle Änderungen seit dem letzten BOT werden festgeschrieben.
- **abort:** Markiert den Abbruch einer Transaktion. Die Datenbasis wird in den Zustand vor Beginn der Transaktion zurückgeführt.
- **define savepoint:** Markiert einen zusätzlichen Sicherungspunkt.
- **backup transaction:** Setzt die Datenbasis auf den jüngsten Sicherungspunkt zurück.

13.3 Abschlus einer Transaktion

Der erfolgreiche Abschluss einer Transaktion erfolgt durch eine Sequenz der Form

$$BOT \;\; op_1;\; op_2;\; \ldots\; ; op_n;\;\; commit$$

Der erfolglose Abschluss einer Transaktion erfolgt entweder durch eine Sequenz der Form

$$BOT \;\; op_1;\; op_2;\; \ldots\; ; op_j;\;\; abort$$

oder durch das Auftreten eines Fehlers

$$BOT \;\; op_1;\; op_2;\; \ldots\; ; op_k;\;\; < Fehler >$$

In diesen Fällen muss der Transaktionsverwalter auf den Anfang der Transaktion zurücksetzen.

13.4 Eigenschaften von Transaktionen

Die Eigenschaften des Transaktionskonzepts werden unter der Abkürzung $ACID$ zusammengefast:

- **Atomicity :** Eine Transaktion stellt eine nicht weiter zerlegbare Einheit dar mit dem Prinzip *alles-oder-nichts*.
- **Consistency :** Nach Abschlus der Transaktion liegt wieder ein konsistenter Zustand vor, während der Transaktion sind Inkonsistenzen erlaubt.
- **Isolation :** Nebenläufig ausgeführte Transaktionen dürfen sich nicht beeinflussen, d. h. jede Transaktion hat den Effekt, den sie verursacht hätte, als wäre sie allein im System.
- **Durability :** Die Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank (auch nach einem späteren Systemfehler).

13.5 Transaktionsverwaltung in SQL

In SQL-92 werden Transaktionen implizit begonnen mit Ausführung der ersten Anweisung. Eine Transaktion wird abgeschlossen durch

- **commit work:** Alle Änderungen sollen festgeschrieben werden (ggf. nicht möglich wegen Konsistenzverletzungen).
- **rollback work:** Alle Änderungen sollen zurückgesetzt werden (ist immer möglich).

Innerhalb einer Transaktion sind Inkonsistenzen erlaubt. Im folgenden Beispiel fehlt vorübergehend der Professoreneintrag zur Vorlesung:

```
insert into Vorlesungen
values (5275, `Kernphysik', 3, 2141);
insert into Professoren
values (2141, `Meitner', `C4', 205);
commit work;
```

13.6 Zustandsübergänge einer Transaktion

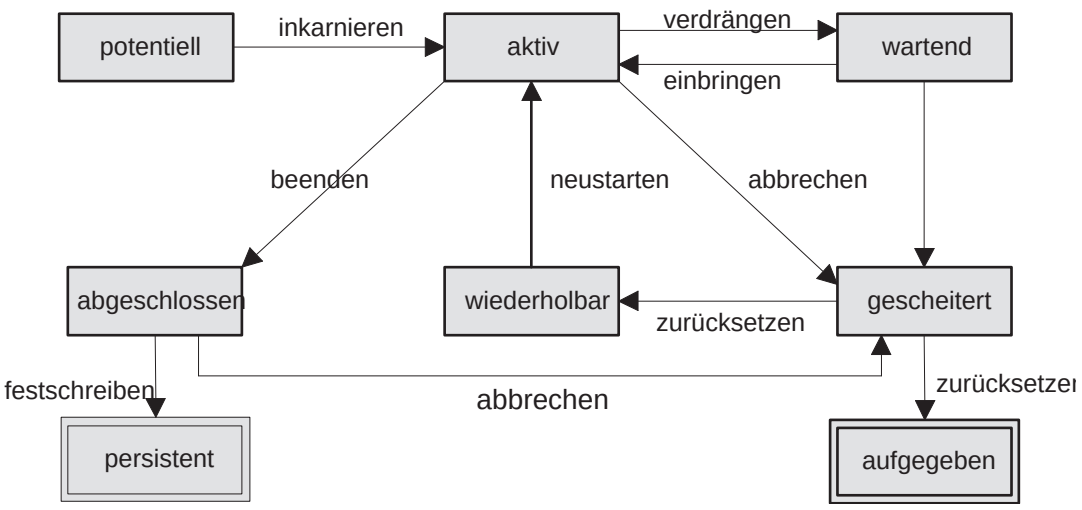


Abb. 97: Zustandsübergangsdiagramm für Transaktionen

Abbildung 97 zeigt die möglichen Übergänge zwischen den Zuständen:

- **potentiell:** Die Transaktion ist codiert und wartet auf ihren Einsatz.
- **aktiv:** Die Transaktion arbeitet.
- **wartend:** Die Transaktion wurde vorübergehend angehalten
- **abgeschlossen:** Die Transaktion wurde durch einen commit-Befehl beendet.
- **persistent:** Die Wirkung einer abgeschlossenen Transaktion wird dauerhaft gemacht.
- **gescheitert:** Die Transaktion ist wegen eines Systemfehlers oder durch einen abort-Befehl abgebrochen worden.
- **wiederholbar:** Die Transaktion wird zur erneuten Ausführung vorgesehen.
- **aufgegeben:** Die Transaktion wird als nicht durchführbar eingestuft.

13.7 Transaktionsverwaltung beim SQL-Server 2000

Listing 12.1 zeigt ein Beispiel für den Einsatz einer Transaktion. Durch das explizite Kommando `begin transaction` sind nach dem `insert` solange andere Transaktionen blockiert, bis durch das explizite `commit work` die Transaktion abgeschlossen wird.

```
begin transaction
insert into professoren
values(55555,'Erika','C4',333,1950-12-24)
select * from professoren where name='Erika'
commit work
```

Listing 12.1 Beispiel für Commit

Listing 12.2 zeigt ein Beispiel für die Möglichkeit, die Auswirkungen einer Transaktion zurückzunehmen. Der zweite `select`-Befehl wird den Studenten mit Namen Fichte nicht auflisten. Andere Transaktionen sind blockiert. Nach dem `Rollback`-Befehl taucht der Student Fichte wieder auf.

```
begin transaction
select * from studenten
delete   from studenten where name='Fichte'
select * from studenten
rollback transaction
select * from studenten
```

Listing 12.2 Beispiel für Rollback

14. Recovery

Aufgabe der Recovery-Komponente des Datenbanksystems ist es, nach einem Fehler den jüngsten konsistenten Datenbankzustand wiederherzustellen.

14.1 Fehlerklassen

Wir unterscheiden drei Fehlerklassen:

1. lokaler Fehler in einer noch nicht festgeschriebenen Transaktion,
2. Fehler mit Hauptspeicherverlust,
3. Fehler mit Hintergrundspeicherverlust.

Lokaler Fehler einer Transaktion

Typische Fehler in dieser Fehlerklasse sind

- Fehler im Anwendungsprogramm,
- expliziter Abbruch (**abort**) der Transaktion durch den Benutzer,
- systemgesteuerter Abbruch einer Transaktion, um beispielsweise eine Verklemmung (Deadlock) zu beheben.

Diese Fehler werden behoben, indem alle Änderungen an der Datenbasis, die von dieser noch aktiven Transaktion verursacht wurden, rückgängig gemacht werden (*lokales Undo*). Dieser Vorgang tritt recht häufig auf und sollte in wenigen Millisekunden abgewickelt sein.

Fehler mit Hauptspeicherverlust

Ein Datenbankverwaltungssystem manipuliert Daten innerhalb eines *Datenbankpuffers*, dessen Seiten zuvor aus dem Hintergrundspeicher *eingelagert* worden sind und nach gewisser Zeit (durch Verdrängung) wieder *ausgelagert* werden müssen. Dies bedeutet, das die im Puffer durchgeführten Änderungen erst mit dem Zurückschreiben in die materialisierte Datenbasis permanent werden. Abbildung 98 zeigt eine Seite P_A, in die das von A nach A' geänderte Item bereits zurückgeschrieben wurde, während die Seite P_C noch das alte, jetzt nicht mehr aktuelle Datum C enthält.

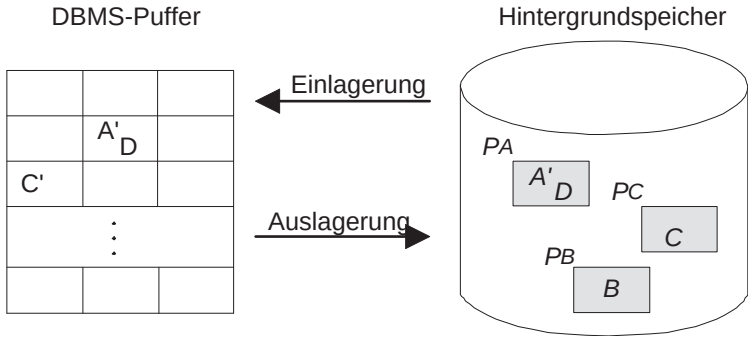


Abb. 98: Schematische Darstellung der zweistufigen Speicherhierarchie

Bei einem Verlust des Hauptspeicherinhalts verlangt das Transaktionsparadigma, das

- alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen rückgängig gemacht werden (*globales undo*) und
- alle noch nicht in die materialisierte Datenbasis eingebrachten Änderungen durch abgeschlossene Transaktionen nachvollzogen werden (*globales redo*).

Fehler dieser Art treten im Intervall von Tagen auf und sollten mit Hilfe einer Log-Datei in wenigen Minuten behoben sein.

Fehler mit Hintergrundspeicherverlust

Fehler mit Hintergrundspeicherverlust treten z.B in folgenden Situationen auf:

- *head crash*, der die Platte mit der materialisierten Datenbank zerstört,
- Feuer/Erdbeben, wodurch die Platte zerstört wird,
- Fehler im Systemprogramm (z. B. im Plattentreiber).

Solche Situationen treten sehr selten auf (etwa im Zeitraum von Monaten oder Jahren). Die Restaurierung der Datenbasis geschieht dann mit Hilfe einer (hoffentlich unversehrten) Archiv-Kopie der materialisierten Datenbasis und mit einem Log-Archiv mit allen seit Anlegen der Datenbasis-Archivkopie vollzogenen Änderungen.

14.2 Die Speicherhierarchie

Ersetzen von Pufferseiten

Eine Transaktion referiert Daten, die über mehrere Seiten verteilt sind. Für die Dauer eines Zugriffs wird die jeweilige Seite im Puffer *fixiert*, wodurch ein Auslagern verhindert wird. Werden Daten auf einer fixierten Seite geändert, so wird die Seite als *dirty* markiert. Nach Abschlus der Operation wird der *FIX*-Vermerk wieder gelöscht und die Seite ist wieder für eine Ersetzung freigegeben.

Es gibt zwei Strategien in Bezug auf das Ersetzen von Seiten:

- ¬*steal* : Die Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ist ausgeschlossen.
- *steal* : Jede nicht fixierte Seite darf ausgelagert werden.

Bei der ¬*steal*-Strategie werden niemals Änderungen einer noch nicht abgeschlossenen Transaktion in die materialisierte Datenbasis übertragen. Bei einem *rollback* einer noch aktiven Transaktion braucht man sich also um den Zustand des Hintergrundspeichers nicht zu kümmern, da die Transaktion vor dem **commit** keine Spuren hinterlassen hat. Bei der *steal*-Strategie müssen nach einem *rollback* die bereits in die materialisierte Datenbasis eingebrachten Änderungen durch ein *Undo* rückgängig gemacht werden.

Zurückschreiben von Pufferseiten

Es gibt zwei Strategien in Bezug auf die Wahl des Zeitpunkts zum Zurückschreiben von modifizierten Seiten:

- *force*: Beim **commit** einer Transaktion werden alle von ihr modifizierten Seiten in die materialisierte Datenbasis zurückkopiert.
- ¬*force*: Modifizierte Seiten werden nicht unmittelbar nach einem **commit**, sondern ggf. auch später, in die materialisierte Datenbasis zurückkopiert.

Bei der ¬*force*-Strategie müssen daher weitere Protokoll-Einträge in der Log-Datei notiert werden, um im Falle eines Fehlers die noch nicht in die materialisierte Datenbasis propagierten Änderungen nachvollziehen zu können. Tabelle 13.1 zeigt die vier Kombinationsmöglichkeiten.

	force	¬force
	• kein Redo	• Redo
¬steal	• kein Undo	• kein Undo
	• kein Redo	• Redo
steal	• Undo	• Undo

Tabelle 13.1: Kombinationsmöglichkeiten beim Einbringen von Änderungen

Auf den ersten Blick scheint die Kombination *force* und ¬*steal* verlockend. Allerdings ist das sofortige Ersetzen von Seiten nach einem **commit** sehr unwirtschaftlich, wenn solche Seiten sehr intensiv auch von anderen, noch aktiven Transaktionen benutzt werden (*hot spots*).

Einbringstrategie

Es gibt zwei Strategien zur Organisation des Zurückschreibens:

- *update-in-place*: Jeder eingelagerten Seite im Datenbankpuffer entspricht eine Seite im Hintergrundspeicher, auf die sie kopiert wird im Falle einer Modifikation.
- Twin-Block-Verfahren: Jeder eingelagerten Seite P im Datenbankpuffer werden zwei Seiten P^0 und P^1 im Hintergrundspeicher zugeordnet, die den letzten bzw. vorletzten Zustand dieser Seite in der materialisierten Datenbasis darstellen. Das Zurückschreiben erfolgt jeweils auf den vorletzten Stand, sodas bei einem Fehler während des Zurückschreibens der letzte Stand noch verfügbar ist.

14.3 Protokollierung der Änderungsoperationen

Wir gehen im weiteren von folgender Systemkonfiguration aus:

- *steal* : Nicht fixierte Seiten können jederzeit ersetzt werden.
- ¬*force* : Geänderte Seiten werden kontinuierlich zurückgeschrieben.
- *update-in-place* : Jede Seite hat genau einen Heimatplatz auf der Platte.
- *Kleine Sperrgranulate* : Verschiedene Transaktionen manipulieren verschiedene Records auf derselben Seite. Also kann eine Seite im Datenbankpuffer sowohl Änderungen einer abgeschlossenen Transaktion als auch Änderungen einer noch nicht abgeschlossenen Transaktion enthalten.

Rücksetzbare Historien

Wie im vorigen Kapitel geschildert wurde, überwacht der Scheduler die Serialisierbarkeit von Transaktionen. Um auch Recovery-Masnahmen durchführen zu können, verlangen wir jetzt zusätzlich die Verwendung von *rücksetzbaren Historien*, die auf den Schreib- und Leseabhängigkeiten basieren.

Wir sagen, das in einer Historie H die Transaktion T_i von der Transaktion T_j liest, wenn folgendes gilt:

- T_j schreibt ein Datum A, das T_i nachfolgend liest.

- T_j wird nicht vor dem Lesevorgang von T_i zurückgesetzt.
- Alle anderen zwischenzeitlichen Schreibvorgänge auf A durch andere Transaktionen werden vor dem Lesen durch T_i zurückgesetzt.

Eine Historie heist *rücksetzbar*, falls immer die schreibende Transaktion T_j vor der lesenden Transaktion T_i ihr **commit** ausführt. Anders gesagt: Eine Transaktion darf erst dann ihr **commit** ausführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind. Wäre diese Bedingung nicht erfüllt, könnte man die schreibende Transaktion nicht zurücksetzen, da die lesende Transaktion dann mit einem offiziell nie existenten Wert für A ihre Berechnung **commited** hätte.

Struktur der Log-Einträge

Für jede Änderungsoperation einer Transaktion wird folgende Protokollinformationen benötigt:

- Die *Redo*-Information gibt an, wie die Änderung nachvollzogen werden kann.
- Die *Undo*-Information gibt an, wie die Änderung rückgängig gemacht werden kann.
- Die *LSN (Log Sequence Number)* ist eine eindeutige Kennung des Log-Eintrags und wird monoton aufsteigend vergeben.
- Die *Transaktionskennung TA* der ausführenden Transaktion.
- Die *PageID* liefert die Kennung der Seite, auf der die Änderung vollzogen wurde.
- Die *PrevLSN* liefert einen Verweis auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion (wird nur aus Effizienzgründen benötigt).

Beispiel einer Log-Datei

Tabelle 13.2 zeigt die verzahnte Ausführung zweier Transaktionen und das zugehörige Log-File. Zum Beispiel besagt der Eintrag mit der *LSN* #3 folgendes:

- Der Log-Eintrag bezieht sich auf Transaktion T_1 und Seite P_A.
- Für ein *Redo* mus A um 50 erniedrigt werden.
- Für ein *Undo* mus A um 50 erhöht werden.
- Der vorhergehende Log-Eintrag hat die *LSN* #1.

Schritt	T_1	T_2	Log
			[LSN, TA, PageID, Redo, Undo, PrevLSN]
1.	BOT		[#1, T_1, **BOT**, 0]
2.	$r(A, a_1)$		
3.		BOT	[#2, T_2, **BOT**, 0]
4.		$r(C, c_2)$	
5.	$a_1 := a_1 - 50$		
6.	$w(A, a_1)$		[#3, T_1, P_A, A-=50, A+=50, #1]
7.		$c_2 := c_2 + 100$	
8.		$w(C, c_2)$	[#4, T_2, P_C, C+=100, C-=100, #2]

9.	$r(B, b_1)$		
10.	$b_1 := b_1 + 50$		
11.	$w(B, b_1)$		[#5, T_1, P_B, B+=50, B-=50, #3]
12.	commit		[#6, T_1, **commit**, #5]
13.		$r(A, a_2)$	
14.		$a_2 := a_2 - 100$	
15.		$w(A, a_2)$	[#7, T_2, P_A, A-=100, A+=100, #4]
16.		commit	[#8, T_2,**commit**, #7]

Tabelle 13.2: Verzahnte Ausführung zweier Transaktionen und Log-Datei

Logische versus physische Protokollierung

In dem Beispiel aus Tabelle 13.2 wurden die *Redo*- und die *Undo*-Informationen logisch protokolliert, d.h. durch Angabe der Operation. Eine andere Möglichkeit besteht in der physischen Protokollierung, bei der statt der *Undo*-Operation das sogenannte *Before-Image* und für die *Redo*-Operation das sogenannte *After-Image* gespeichert wird. Bei der logischen Protokollierung wird

- das *Before-Image* durch Ausführung des *Undo*-Codes aus dem *After-Image* generiert,
- das *After-Image* durch Ausführung des *Redo*-Codes aus dem *Before*-Image generiert.

Während der Recovery-Phase ist zunächst unbekannt, bis zu welcher Einzeloperation die Effekte einer Transaktion bereits materialisiert wurden. Also ist es wichtig zu erkennen, ob das *Before-Image* oder *After-Image* in der materialisierten Datenbasis enthalten ist. Hierzu dient die *LSN*. Beim Anlegen eines Log-Eintrages wird die neu generierte LSN in einen reservierten Bereich der Seite geschrieben und dann später mit dieser Seite in die Datenbank zurückkopiert. Daraus läst sich erkennen, ob für einen bestimmten Log-Eintrag das *Before-Image* oder das *After-Image* in der Seite steht (da während der Reparaturphase nicht bekannt ist, bis zu welcher Einzeloperation die Effekte bereits materialisiert wurden):

- Wenn die LSN der Seite einen kleineren Wert als die LSN des Log-Eintrags enthält, handelt es sich um das *Before-Image*.
- Ist die LSN der Seite gröser oder gleich der LSN des Log-Eintrags, dann wurde bereits das *After-Image* auf den Hintergrundspeicher propagiert.

Schreiben der Log-Information

Bevor eine Änderungsoperation ausgeführt wird, mus der zugehörige Log-Eintrag angelegt werden. Die Log-Einträge werden im *Log-Puffer* im Hauptspeicher zwischengelagert. Abbildung 99 zeigt das Wechselspiel zwischen den beteiligten Sicherungskomponenten.

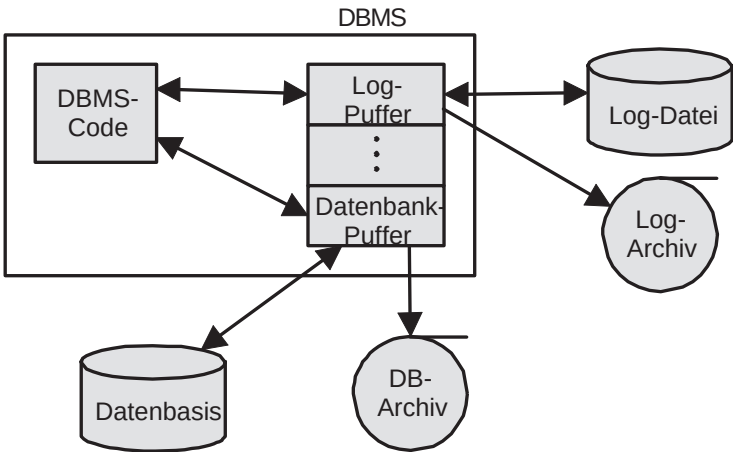


Abb. 99: Speicherhierarchie zur Datensicherung

In modernen Datenbanksystemen ist der Log-Puffer als Ringpuffer organisiert. An einem Ende wird kontinuierlich geschrieben und am anderen Ende kommen laufend neue Einträge hinzu (Abbildung 100). Die Log-Einträge werden gleichzeitig auf das temporäre Log (Platte) und auf das Log-Archiv (Magnetband) geschrieben.

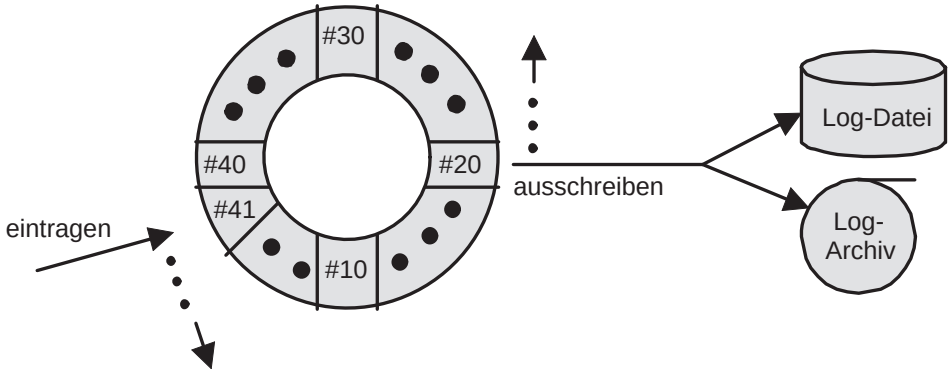


Abb. 100: Log-Ringpuffer

WAL-Prinzip

Beim Schreiben der Log-Information gilt das *WAL*-Prinzip (Write Ahead Log):

- Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle zu ihr gehörenden Log-Einträge geschrieben werden. Dies ist erforderlich, um eine erfolgreich abgeschlossene Transaktion nach einem Fehler nachvollziehen zu können (*redo*).
- Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in die Log-Datei geschrieben werden. Dies ist erforderlich, um im Fehlerfall die Änderungen nicht abgeschlossener Transaktionen aus den modifizierten Seiten der materialisierten Datenbasis entfernen zu können (*undo*).

14.4 Wiederanlauf nach einem Fehler

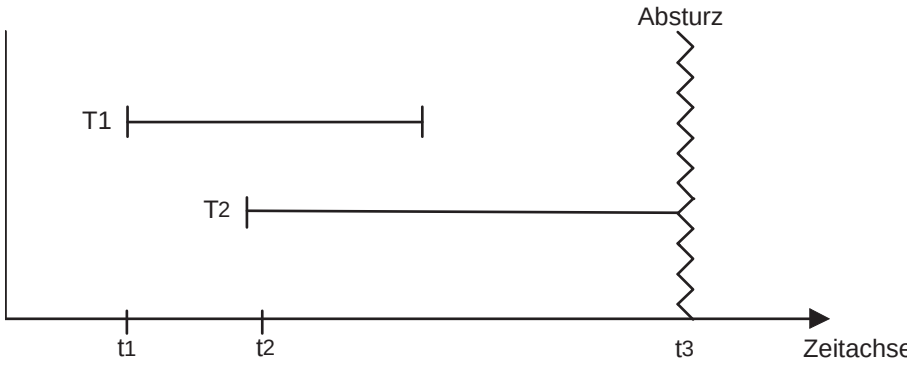


Abb. 101: Zwei Transaktionstypen bei Systemabsturz

Abbildung 101 zeigt die beiden Transaktionstypen, die nach einem Fehler mit Verlust des Hauptspeicherinhalts zu behandeln sind:

- Transaktion T_1 ist ein *Winner* und verlangt ein *Redo*.
- Transaktion T_2 ist ein *Loser* und verlangt ein *Undo*.

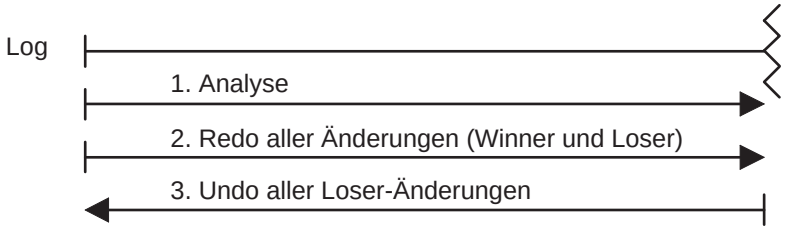


Abb. 102: Wiederanlauf in drei Phasen

Der Wiederanlauf geschieht in drei Phasen (Abbildung 102):

1. *Analyse*: Die Log-Datei wird von Anfang bis Ende analysiert, um die *Winner* (kann **commit** vorweisen) und die *Loser* (kann kein **commit** vorweisen) zu ermitteln.
2. *Redo*: Es werden alle protokollierten Änderungen (von Winner und Loser) in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht, sofern sich nicht bereits das Afterimage des Protokolleintrags in der materialisierten Datenbasis befindet. Dies ist dann der Fall, wenn die *LSN* der betreffenden Seite gleich oder gröser ist als die *LSN* des Protokolleintrags.
3. Undo: Die Log-Datei wird in umgekehrter Richtung, d.h. von hinten nach vorne, durchlaufen. Dabei werden die Einträge von *Winner*-Transaktionen übergangen. Für jeden Eintrag einer *Loser*-Transaktion wird die *Undo*-Operation durchgeführt.

Spezielle Vorkehrungen müssen getroffen werden, um auch Fehler beim Wiederanlauf kompensieren zu können. Es wird nämlich verlangt, das die *Redo*- und *Undo*-Phasen *idempotent* sind, d.h. sie müssen auch nach mehrmaliger Ausführung (hintereinander) immer wieder dasselbe Ergebnis liefern:

undo(undo(...(undo(a))...)) = undo(a)
redo(redo(...(redo(a))...)) = redo(a)

Für die Redo-Phase wird dies erreicht, indem jeweils die zum Log-Eintrag gehörende Log-Sequence-Number in den reservierten Bereich der Seite geschrieben und beim Zurückschreiben persistent wird. Hierdurch kann bei einem erneuten Redo überprüft werden, ob sich auf der Seite bereits das *After Image* befindet oder auf dem *Before Image* noch die Redo-Operation angewendet werden muss.

Während der Redo-Phase wird für jede durchgeführte *Undo*-Operation ein *Compensation Log Record* an die Log-Datei angehängt mit eigener LSN, so dass bei einem erneuten Absturz in der nachfolgenden erneuten Redo-Phase diese *Undo*-Schritte unter Beachtung ihrer LSN-Einträge wiederholt werden können.

14.5 Sicherungspunkte

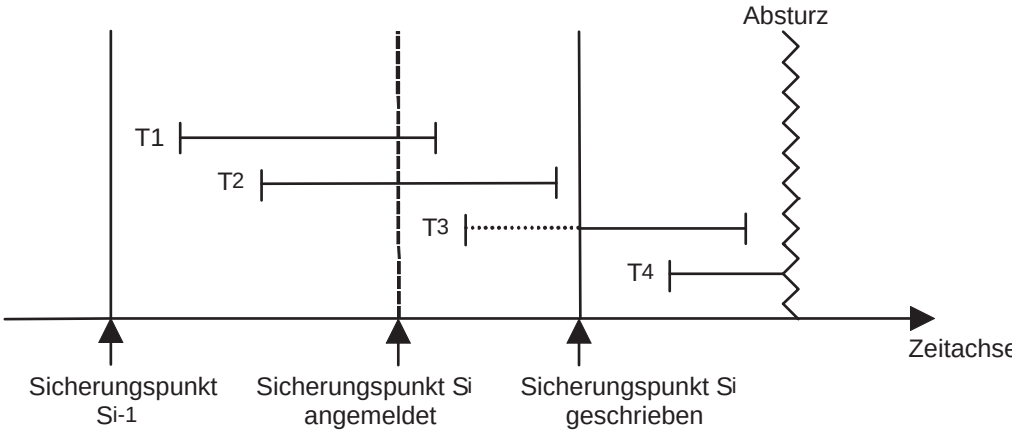


Abb. 103: Transaktionsausführung relativ zu einem Sicherungspunkt

Mit zunehmender Betriebszeit des Datenbanksystems wird die zu verarbeitende Log-Datei immer umfangreicher. Durch einen *Sicherungspunkt* wird eine Position im Log vermerkt, über den man beim Wiederanlauf nicht hinausgehen mus.

Abbildung 103 zeigt den dynamischen Verlauf. Nach Anmeldung des neuen Sicherungspunktes S_i wird die noch aktive Transaktion T_2 zu Ende geführt und der Beginn der Transaktion T_3 verzögert. Nun werden alle modifizierten Seiten auf den Hintergrundspeicher ausgeschrieben und ein transaktionskonsistenter Zustand ist mit dem Sicherungspunkt S_i erreicht. Danach kann man mit der Log-Datei wieder von vorne beginnen.

14.6 Verlust der materialisierten Datenbasis

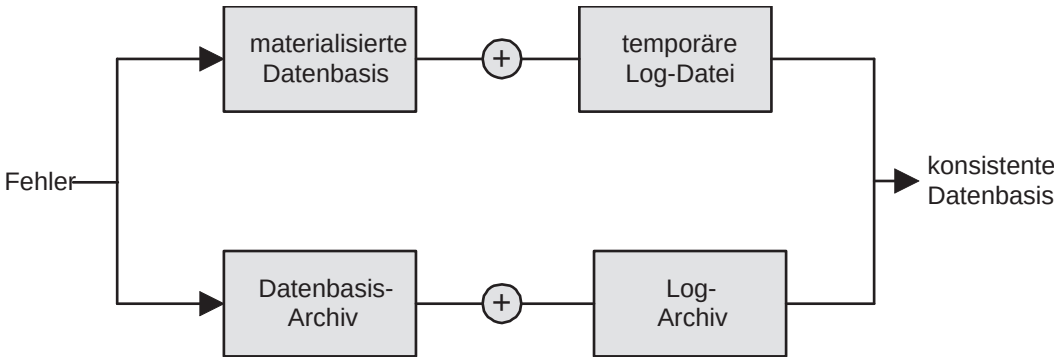


Abb. 104: Zwei Recovery-Arten

Bei Zerstörung der materialisierten Datenbasis oder der Log-Datei kann man aus der Archiv-Kopie und dem Log-Archiv den jüngsten, konsistenten Zustand wiederherstellen.

Abbildung 104 fast die zwei möglichen Recoveryarten nach einem Systemabsturz zusammen:

- Der obere (schnellere) Weg wird bei intaktem Hintergrundspeicher beschritten.

- Der untere (langsamere) Weg wird bei zerstörtem Hintergrundspeicher beschritten.

15. Mehrbenutzersynchronisation

15.1 Multiprogramming

Unter $Multiprogramming$ versteht man die nebenläufige, verzahnte Ausführung mehrerer Programme. Abbildung 105 zeigt exemplarisch die dadurch erreichte bessere CPU-Auslastung.

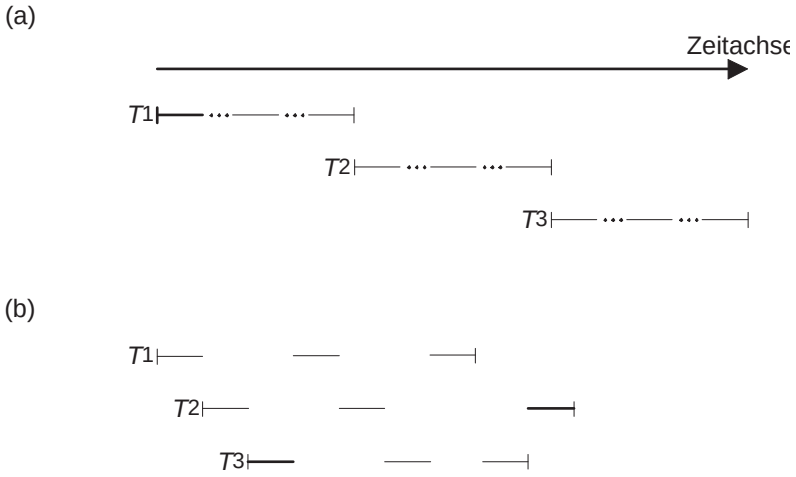


Abb. 105: Einbenutzerbetrieb (a) versus Mehrbenutzerbetrieb (b)

15.2 Fehler bei unkontrolliertem Mehrbenutzerbetrieb

Lost Update

Transaktion T_1 transferiert 300,- Euro von Konto A nach Konto B, Transaktion T_2 schreibt Konto A die 3 % Zinseinkünfte gut.

Den Ablauf zeigt Tabelle 13.1. Die im Schritt 5 von Transaktion T_2 gutgeschriebenen Zinsen gehen verloren, da sie in Schritt 6 von Transaktion T_1 wieder überschrieben werden.

Schritt	T_1	T_2
1.	read(A, a_1)	
2.	$a_1 := a_1$ - 300	
3.		read(A, a_2)
4.		$a_2 := a_2$ * 1.03
5.		write(A, a_2)
6.	write(A, a_1)	
7.	read(B, b_1)	
8.	$b_1 := b_1 + 300$	
9.	write(B, b_1)	

Tabelle 13.1: Beispiel für Lost Update

Dirty Read

Transaktion T_2 schreibt die Zinsen gut anhand eines Betrages, der nicht in einem konsistenten Zustand der Datenbasis vorkommt, da Transaktion T_1 später durch ein **abort** zurückgesetzt wird. Den Ablauf zeigt Tabelle 13.2.

Schritt	T_1	T_2
1.	read(A, a_1)	
2.	$a_1 := a_1 - 300$	
3.	write(A, a_1)	
4.		read(A, a_2)
5.		$a_2 := a_2 * 1.03$
6.		write(A, a_2)
7.	read(B, b_1)	
8.	...	
9.	abort	

Tabelle 13.2: Beispiel für Dirty Read

Phantomproblem

Während der Abarbeitung der Transaktion T_2 fügt Transaktion T_1 ein Datum ein, welches T_2 liest. Dadurch berechnet Transaktion T_2 zwei unterschiedliche Werte. Den Ablauf zeigt Tabelle 13.3.

T_1	T_2
	select sum(KontoStand)
	from Konten;
insert into Konten	
values (C, 1000, ...);	
	select sum(KontoStand)
	from Konten;

Tabelle 13.3: Beispiel für das Phantomproblem

15.3 Serialisierbarkeit

Eine $Historie$, auch genannt $Schedule$, für eine Menge von Transaktionen ist eine Festlegung für die Reihenfolge sämtlicher relevanter Datenbankoperationen. Ein Schedule heist $seriell$, wenn alle Schritte einer Transaktion unmittelbar hintereinander ablaufen. Wir unterscheiden nur noch zwischen $read$- und $write$-Operationen.

Zum Beispiel transferiere T_1 einen bestimmten Betrag von A nach B und T_2 transferiere einen Betrag von C nach A. Eine mögliche Historie zeigt Tabelle 13.4.

Schritt	T_1	T_2
1.	BOT	
2.	read(A)	
3.		BOT
4.		read(C)
5.	write(A)	
6.		write(C)
7.	read(B)	
8.	write(B)	
9.	commit	
10.		read(A)
11.		write(A)
12.		commit

Tabelle 13.4: Serialisierbare Historie

Offenbar wird derselbe Effekt verursacht, als wenn zunächst T_1 und dann T_2 ausgeführt worden wäre, wie Tabelle 13.5 demonstriert.

Schritt	T_1	T_2
1.	BOT	
2.	read(A)	
3.	write(A)	
4.	read(B)	
5.	write(B)	
6.	commit	
7.		BOT
8.		read(C)
9.		write(C)
10.		read(A)
11.		write(A)
12.		commit

Tabelle 13.5: Serielle Historie

Wir nennen deshalb das (verzahnte) Schedule $serialisierbar$.

Tabelle 13.6 zeigt ein Schedule der Transaktionen T_1 und T_3, welches nicht serialisierbar ist.

Schritt	T_1	T_3
1.	BOT	
2.	read(A)	
3.	write(A)	

4.		BOT
5.		read(A)
6.		write(A)
7.		read(B)
8.		write(B)
9.		commit
10.	read(B)	
11.	write(B)	
12.	commit	

Tabelle 13.6: Nicht-serialisierbares Schedule

Der Grund liegt darin, das bzgl. Datenobjekt A die Transaktion T_1 vor T_3 kommt, bzgl. Datenobjekt B die Transaktion T_3 vor T_1 kommt. Dies ist nicht äquivalent zu einer der beiden möglichen seriellen Ausführungen T_1T_3 oder T_3T_1.

Im Einzelfall kann die konkrete Anwendungssemantik zu einem äquivalenten seriellen Schedule führen, wie Tabelle 13.7 zeigt.

Schritt	T_1	T_3
1.	BOT	
2.	read(A, a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A, a_1)	
5.		BOT
6.		read(A, a_2)
7.		$a_2 := a_2 - 100$
8.		write(A, a_2)
9.		read(B, b_2)
10.		$b_2 := b_2 + 100$
11.		write(B, b_2)
12.		commit
13.	read(B, b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B, b_1)	
16.	commit	

Tabelle 13.7: Zwei verzahnte Überweisungen

In beiden Fällen wird Konto A mit 150,- Euro belastet und Konto B werden 150,- Euro gutgeschrieben.

Unter einer anderen Semantik würde T_1 einen Betrag von 50,- Euro von A nach B überweisen und Transaktion T_2 würde beiden Konten jeweils 3 % Zinsen gutschreiben. Tabelle 13.8 zeigt den Ablauf.

Schritt	T_1	T_3

1.	BOT	
2.	read(A, a_1)	
3.	$a_1 := a_1 - 50$	
4.	write(A, a_1)	
5.		BOT
6.		read(A, a_2)
7.		$a_2 := a_2 * 1.03$
8.		write(A, a_2)
9.		read(B, b_2)
10.		$b_2 := b_2 * 1.03$
11.		write(B, b_2)
12.		commit
13.	read(B, b_1)	
14.	$b_1 := b_1 + 50$	
15.	write(B, b_1)	
16.	commit	

Tabelle 13.8: Überweisung verzahnt mit Zinsgutschrift

Offenbar entspricht diese Reihenfolge keiner möglichen seriellen Abarbeitung T_1T_3 oder T_3T_1, denn es fehlen in jedem Falle Zinsen in Höhe von 3 % von 50,- Euro = 1,50 Euro.

15.4 Theorie der Serialisierbarkeit

Eine $TransaktionT_i$ besteht aus folgenden elementaren Operationen:

- r_i(A) zum Lesen von Datenobjekt A,
- w_i(A) zum Schreiben von Datenobjekt A,
- a_i zur Durchführung eines **abort**,
- c_i zur Durchführung eines **commit**.

Eine Transaktion kann nur eine der beiden Operationen **abort** oder **commit** durchführen; diese müssen jeweils am Ende der Transaktion stehen. Implizit wird ein **BOT** vor der ersten Operation angenommen. Wir nehmen für die Transaktion eine feste Reihenfolge der Elementaroperationen an. Eine $Historie$, auch genannt $Schedule$, ist eine Festlegung der Reihenfolge für sämtliche beteiligten Einzeloperationen.

Gegeben Transaktionen T_i und T_j, beide mit Zugriff auf Datum A. Folgende vier Fälle sind möglich:

- r_i(A) und r_j(A) : kein Konflikt, da Reihenfolge unerheblich
- r_i(A) und w_j(A) : Konflikt, da Reihenfolge entscheidend
- w_i(A) und r_j(A) : Konflikt, da Reihenfolge entscheidend
- w_i(A) und w_j(A) : Konflikt, da Reihenfolge entscheidend

Von besonderem Interesse sind die $Konfliktoperationen$.

Zwei Historien H_1 und H_2 über der gleichen Menge von Transaktionen sind äquivalent (in Zeichen $H_1 \equiv H_2$), wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen. D. h., für die durch H_1 und H_2 induzierten Ordnungen auf den Elementaroperationen $<_{H_1}$ bzw. $<_{H_2}$ wird verlangt: Wenn p_i und q_j Konfliktoperationen sind mit $p_i <_{H_1} q_j$, dann mus auch $p_i <_{H_2} q_j$ gelten. Die Anordnung der nicht in Konflikt stehenden Operationen ist irrelevant.

15.5 Algorithmus zum Testen auf Serialisierbarkeit:

- **Input:** Eine Historie H für Transaktionen $T_1, \ldots, T_k$.
- **Output:** entweder: "nein, ist nicht serialisierbar`` oder "ja, ist serialisierbar`` + serielles Schedule
- **Idee:** Bilde gerichteten Graph G, dessen Knoten den Transaktionen entsprechen. Für zwei Konfliktoperationen p_i, q_j aus der Historie H mit $p_i <_H q_j$ fügen wir die Kante $T_i \to T_j$ in den Graph ein.

Es gilt das **Serialisierbarkeitstheorem:** Eine Historie H ist genau dann serialisierbar, wenn der zugehörige Serialisierbarkeitsgraph azyklisch ist. Im Falle der Kreisfreiheit läst sich die äquivalente serielle Historie aus der topologischen Sortierung des Serialisierbarkeitsgraphen bestimmen. Als Beispiel-Input für diesen Algorithmus verwenden wir die in Tabelle 13.9 gezeigte Historie über den Transaktionen T_1, T_2, T_3 mit insgesamt 14 Operationen.

Schritt	T_1	T_2	T_3
1.	r_1(A)		
2.		r_2(B)	
3.		r_2(C)	
4.		w_2(B)	
5.	r_1(B)		
6.	w_1(A)		
7.		r_2(A)	
8.		w_2(C)	
9.		w_2(A)	
10.			r_3(A)
11.			r_3(C)
12.	w_1(B)		
13.			w_3(C)
14.			w_3(A)

Tabelle 13.9: Historie H mit drei Transaktionen

Folgende Konfliktoperationen existieren für Historie H:

$$w_2(B) < r_1(B),$$
$$w_1(A) < r_2(A),$$
$$w_2(C) < r_3(C),$$
$$w_2(A) < r_3(A).$$

Daraus ergeben sich die Kanten

$$T_2 \rightarrow T_1,$$
$$T_1 \rightarrow T_2,$$
$$T_2 \rightarrow T_3,$$
$$T_2 \rightarrow T_3.$$

Den resultierenden Graph zeigt Abbildung 106

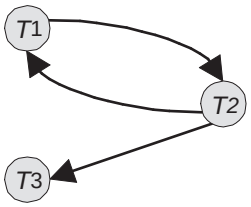


Abb. 106: Der zu Historie H konstruierte Serialisierbarkeitsgraph

Da der konstruierte Graph einen Kreis besitzt, ist die Historie nicht serialisierbar.

15.6 Sperrbasierte Synchronisation

Bei der sperrbasierten Synchronisation wird während des laufenden Betriebs sichergestellt, das die resultierende Historie serialisierbar bleibt. Dies geschieht durch die Vergabe einer $Sperre$ (englisch: $lock$).

Je nach Operation (**read** oder **write**) unterscheiden wir zwei Sperrmodi:

- **S** (shared, read lock, Lesesperre):
  Wenn Transaktion T_i eine S-Sperre für Datum A besitzt, kann T_i **read**(A) ausführen. Mehrere Transaktionen können gleichzeitig eine S-Sperre auf dem selben Objekt A besitzen.
- **X** (exclusive, write lock, Schreibsperre):
  Ein write(A) darf nur die eine Transaktion ausführen, die eine X-Sperre auf A besitzt.

Tabelle 13.10 zeigt die Kompatibilitätsmatrix für die Situationen NL (no lock), S (read lock) und X (write lock).

	NL	S	X
S	$\surd$	$\surd$	-
X	$\surd$	-	-

Tabelle 13.10: Kompatibilitätsmatrix

Folgendes Zwei-Phasen-Sperrprotokoll ($twophaselocking$, $2PL$) garantiert die Serialisierbarkeit:

1. Jedes Objekt mus vor der Benutzung gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. Eine Transaktion respektiert vorhandene Sperren gemäs der Verträglichkeitsmatrix und wird ggf. in eine Warteschlange eingereiht.
4. Jede Transaktion durchläuft eine $Wachstumsphase$ (nur Sperren anfordern) und dann eine $Schrumpfungsphase$ (nur Sperren freigeben).
5. Bei Transaktionsende mus eine Transaktion alle ihre Sperren zurückgeben.

Abbildung 107 visualisiert den Verlauf des 2PL-Protokolls. Tabelle 13.11 zeigt eine Verzahnung zweier Transaktionen nach dem 2PL-Protokoll.

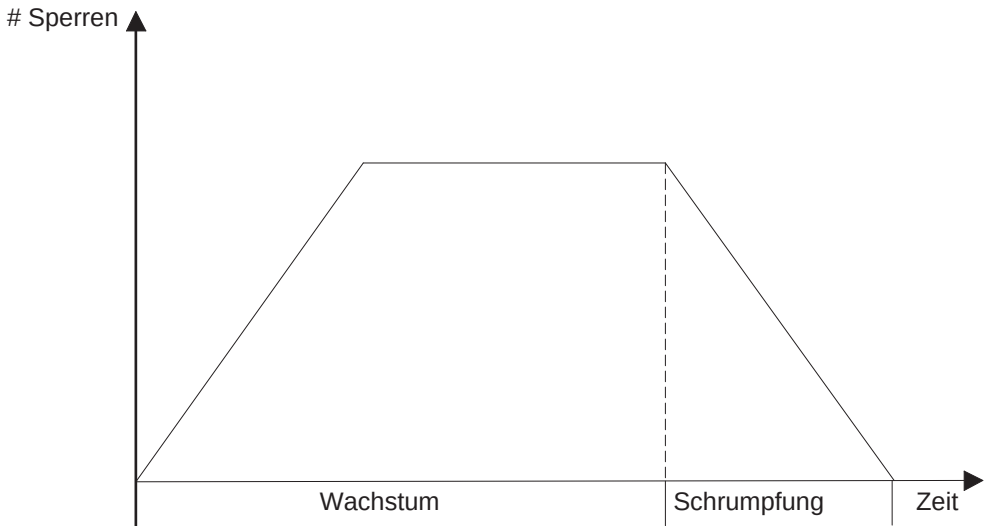


Abb. 107: 2-Phasen-Sperrprotokoll

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		BOT	
6.		**lockS**(A)	T_2 mus warten
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wecken
10.		read(A)	
11.		**lockS**(B)	T_2 mus warten
12.	write(B)		
13.	**unlockX**(B)		T_2 wecken
14.		read(B)	
15.	commit		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		commit	

Tabelle 13.11: Beispiel für 2PL-Protokoll

15.7 Verklemmungen (Deadlocks)

Ein schwerwiegendes Problem bei sperrbasierten Synchronisationsmethoden ist das Auftreten von Verklemmungen (englisch: deadlocks). Tabelle 13.12 zeigt ein Beispiel.

Schritt	T_1	T_2	Bemerkung
1.	BOT		
2.	**lockX**(A)		
3.		BOT	
4.		**lockS**(B)	
5.		read(B)	
6.	read(A)		
7.	write(A)		
8.	**lockX**(B)		T_1 mus warten auf T_2
9.		**lockS**(A)	T_2 mus warten auf T_1
10.	...	...	$\Rightarrow Deadlock$

Tabelle 13.12: Ein verklemmter Schedule

Eine Methode zur Erkennung von Deadlocks ist die $Time-out$–Strategie. Falls eine Transaktion innerhalb eines Zeitmases (z. B. 1 Sekunde) keinerlei Fortschritt erzielt, wird sie zurückgesetzt. Allerdings ist die Wahl des richtigen Zeitmases problematisch. Eine präzise, aber auch teurere - Methode zum Erkennen von Verklemmungen basiert auf dem sogenannten $Wartegraphen$. Seine Knoten entsprechen den Transaktionen. Eine Kante existiert von T_i nach T_j, wenn T_i auf die Freigabe einer Sperre von T_j wartet. Abbildung 108 zeigt ein Beispiel.

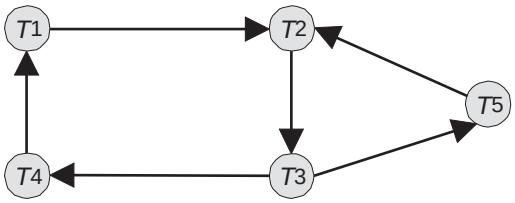


Abb. 108: Wartegraph mit zwei Zyklen

Es gilt der Satz: Die Transaktionen befinden sich in einem Deadlock genau dann, wenn der Wartegraph einen Zyklus aufweist.

Eine Verklemmung wird durch das Zurücksetzen einer Transaktion aufgelöst:

- Minimierung des Rücksetzaufwandes: Wähle jüngste beteiligte Transaktion.
- Maximierung der freigegebenen Resourcen: Wähle Transaktion mit den meisten Sperren.
- Vermeidung von Verhungern (engl. Starvation): Wähle nicht diejenige Transaktion, die schon oft zurückgesetzt wurde.
- Mehrfache Zyklen: Wähle Transaktion, die an mehreren Zyklen beteiligt ist.

15.8 Hierarchische Sperrgranulate

Bisher wurden alle Sperren auf derselben $Granularität$ erworben. Mögliche Sperrgranulate sind:

- Datensatz $\widehat{=}$ Tupel
- Seite $\widehat{=}$ Block im Hintergrundspeicher
- Segment $\widehat{=}$ Zusammenfassung von Seiten

- Datenbasis $\widehat{=}$ gesamter Datenbestand

Abbildung 109 zeigt die hierarchische Anordnung der möglichen Sperrgranulate.

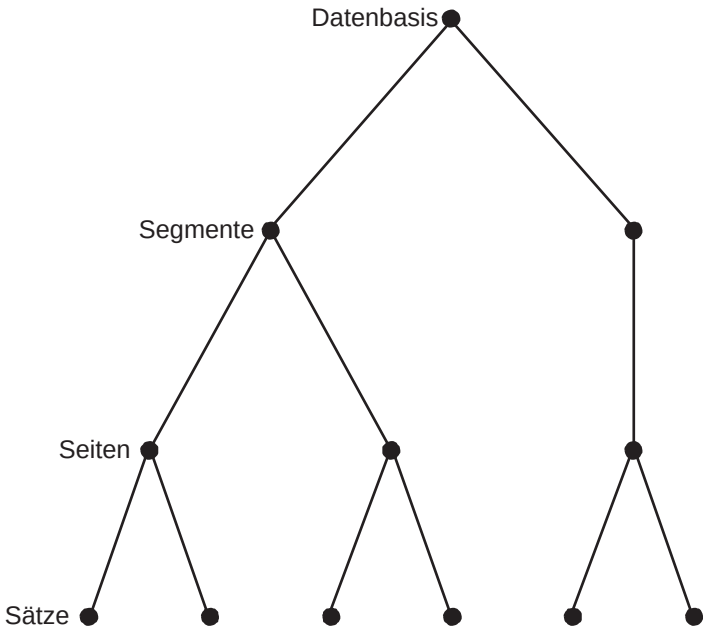


Abb. 109: Hierarchie der Sperrgranulate

Eine Vermischung von Sperrgranulaten hätte folgende Auswirkung. Bei Anforderung einer Sperre für eine Speichereinheit, z.B. ein Segment, müssen alle darunterliegenden Seiten und Sätze auf eventuelle Sperren überprüft werden. Dies bedeutet einen immensen Suchaufwand. Auf der anderen Seite hätte die Beschränkung auf nur eine Sperrgranularität folgende Nachteile:

- Bei zu kleiner Granularität werden Transaktionen mit hohem Datenzugriff stark belastet.
- Bei zu groser Granularität wird der Parallelitätsgrad unnötig eingeschränkt.

Die Lösung des Problems besteht im *multiple granularity locking (MGL)*. Hierbei werden zusätzliche *Intentionssperren* verwendet, welche die Absicht einer weiter unten in der Hierarchie gesetzten Sperre anzeigen. Tabelle 13.13 zeigt die Kompatibilitätsmatrix. Die Sperrmodi sind:

- **NL:** keine Sperrung (no lock);
- **S:** Sperrung durch Leser,
- **X:** Sperrung durch Schreiber,
- **IS:** Lesesperre (S) weiter unten beabsichtigt,
- **IX:** Schreibsperre (X) weiter unten beabsichtigt.

	NL	S	X	IS	IX
S	$\surd$	$\surd$	-	$\surd$	-
X	$\surd$	-	-	-	-
IS	$\surd$	$\surd$	-	$\surd$	$\surd$
IX	$\surd$	-	-	$\surd$	$\surd$

Tabelle 13.13: Kompatibilitätsmatrix beim Multiple-Granularity-Locking

Die Sperrung eines Datenobjekts mus so durchgeführt werden, das erst geeignete Sperren in allen übergeordneten Knoten in der Hierarchie erworben werden:

1. Bevor ein Knoten mit *S* oder *IS* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*- oder *IS*-Modus gehalten werden.
2. Bevor ein Knoten mit *X* oder *IX* gesperrt wird, müssen alle Vorgänger vom Sperrer im *IX*-Modus gehalten werden.
3. Die Sperren werden von unten nach oben freigegeben.

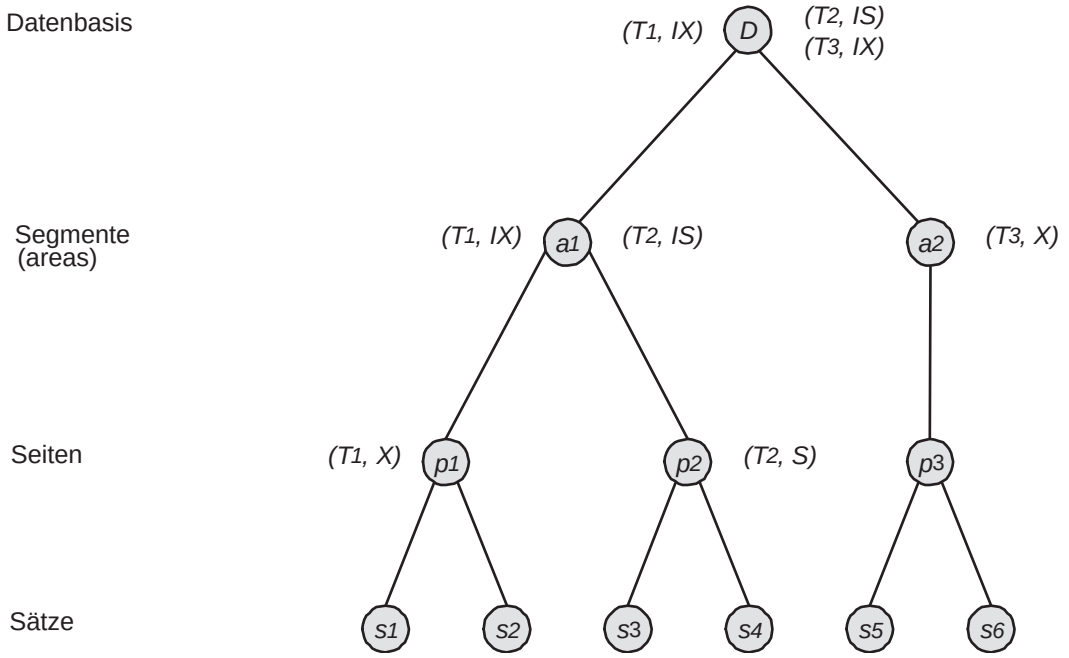


Abb. 110: Datenbasis-Hierarchie mit Sperren

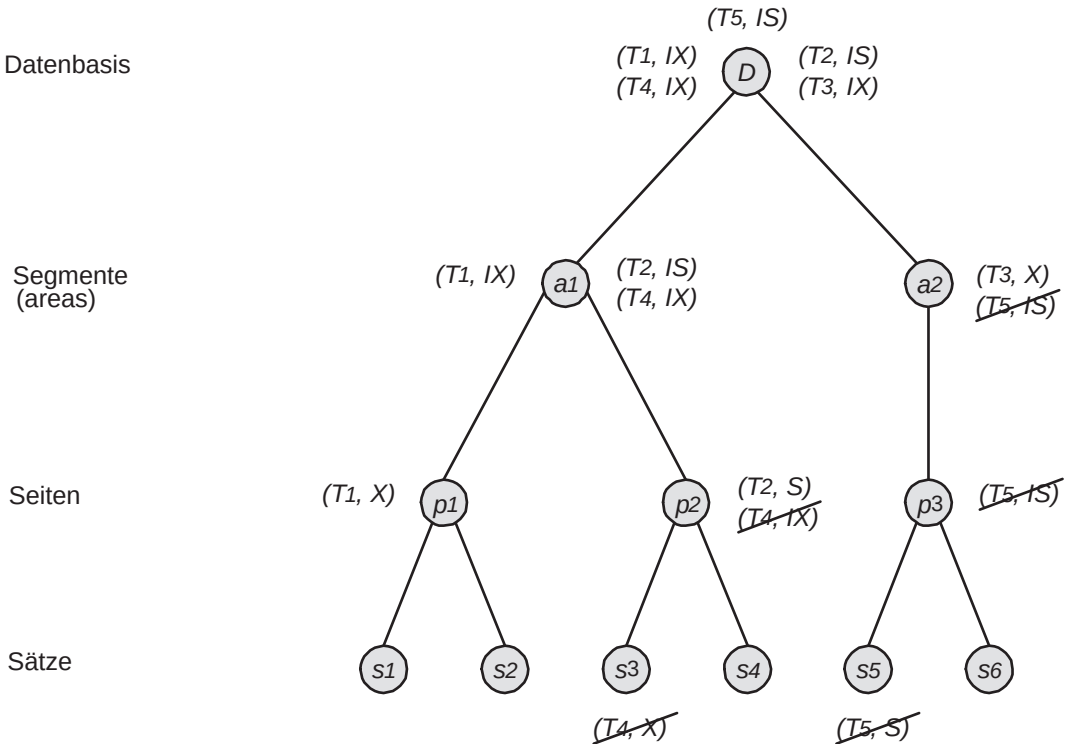


Abb. 111: Datenbasis-Hierarchie mit zwei blockierten Transaktionen

Abbildung 110 zeigt eine Datenbasis-Hierarchie, in der drei Transaktionen erfolgreich Sperren erworben haben:

- T_1 will die Seite p_1 zum Schreiben sperren und erwirbt zunächst *IX*-Sperren auf der Datenbasis *D* und auf Segment a_1.
- T_2 will die Seite p_2 zum Lesen sperren und erwirbt zunächst *IS*-Sperren auf der Datenbasis *D* und auf Segment a_1.
- T_3 will das Segment a_2 zum Schreiben sperren und erwirbt zunächst eine *IX*-Sperre auf der Datenbasis *D*.

Nun fordern zwei weitere Transaktionen T_4 (Schreiber) und T_5 (Leser) Sperren an:

- T_4 will Satz s_3 exklusiv sperren. Auf dem Weg dorthin erhält T_4 die erforderlichen *IX*-Sperren für D und a_1, jedoch kann die *IX*-Sperre für p_2 nicht gewährt werden.
- T_5 will Satz s_5 zum Lesen sperren. Auf dem Weg dorthin erhält T_5 die erforderliche *IS*-Sperren nur für D, jedoch können die *IS*-Sperren für a_2 und p_3 zunächst nicht gewährt werden.

Abbildung 111 zeigt die Situation nach dem gerade beschriebenen Zustand. Die noch ausstehenden Sperren sind durch eine Durchstreichung gekennzeichnet. Die Transaktionen T_4 und T_5 sind blockiert, aber nicht verklemmt und müssen auf die Freigabe der Sperren (T_2, S) und $T_3, X)$ warten.

15.9 Zeitstempelverfahren

Jede Transaktion erhält beim Eintritt ins System einen eindeutigen Zeitstempel durch die System-Uhr (bei 1 tic pro Millisecunde $\Rightarrow$ 32 Bits reichen für 49 Tage). Das entstehende Schedule gilt als korrekt, falls seine Wirkung dem seriellen Schedule gemäs Eintrittszeiten entspricht.

Jede Einzelaktion drückt einem Item seinen Zeitstempel auf. D.h. jedes Item hat einen

Lesestempel $\equiv$höchster Zeitstempel, verabreicht durch eine Leseoperation
Schreibstempel$\equiv$höchster Zeitstempel, verabreicht durch eine Schreiboperation

Die gesetzten Marken sollen Verbotenes verhindern:

1. Transaktion mit Zeitstempel t darf kein Item lesen mit Schreibstempel $t_w > t$ (denn der alte Item-Wert ist weg).
2. Transaktion mit Zeitstempel t darf kein Item schreiben mit Lesestempel $t_r > t$ (denn der neue Wert kommt zu spät).
3. Zwei Transaktionen können dasselbe Item zu beliebigen Zeitpunkten lesen.
4. Wenn Transaktion mit Zeitstempel t ein Item beschreiben will mit Schreibstempel $t_w > t$, so wird der Schreibbefehl ignoriert.

Bei Eintreten von Fall 1 und 2 mus die Transaktion zurückgesetzt werden. Bei den beiden anderen Fällen brauchen die Transaktionen nicht zurückgesetzt zu werden.

Also folgt als Regel für Einzelaktion X mit Zeitstempel t bei Zugriff auf Item mit Lesestempel t_r und Schreibstempel t_w:

```
if (X = read) and (t ≥ t_w)

   führe X aus und setze t_r := max{t_r, t}
if (X = write) and (t ≥ t_r) and (t ≥ t_w) then

   führe X aus und setze t_w := t
```

`if (X = write) and (`$t_r \leq t < t_w$`) then tue nichtselse {(X = read and` $t < t_w$`) or (X = write and` $t < t_r$`)}setze Transaktion zurück`

Tabelle 13.14 und 13.15 zeigen zwei Beispiele für die Synchronisation von Transaktionen mit dem Zeitstempelverfahren.

	T_1	T_2	
	Stempel 150	160	Item a hat $t_r = t_w = 0$
1.)	read(a)		
	t_r := 150		
2.)		read(a)	
		t_r := 160	
3.)	a := a - 1		
4.)		a := a - 1	
5.)		write(a)	ok, da $160 \geq t_r = 160$ und$160 \geq t_w = 0$
		$t_w := 160$	
6.)	write(a)		T_1 wird zurückgesetzt, da
			$150 < t_r = 160$

Tabelle 13.14: Beispiel für Zeitstempelverfahren

In Tabelle 13.14 wird in Schritt 6 die Transaktion T_1 zurückgesetzt, da ihr Zeitstempel kleiner ist als der Lesestempel des zu überschreibenden Items *a*$(150 < t_r = 160)$. In Tabelle 13.15 wird in Schritt 6 die Transaktion T_2 zurückgesetzt, da ihr Zeitstempel kleiner ist als der Lesestempel von Item *c*$(150 < t_r(c) = 175)$. In Schritt 7 wird der Schreibbefehl von Transaktion T_3 ignoriert, da der Zeitstempel von T_3 kleiner ist als der Schreibstempel des zu beschreibenden Items *a*$(175 < t_w(a) = 200)$.

	T_1	T_2	T_3	a	b	c
	200	150	175	t_r = 0	t_r = 0	t_r = 0
				t_w = 0	t_w = 0	t_w = 0
1.)	read(b)				t_r = 200	
2.)		read(a)		t_r = 150		
3.)			read(c)			t_r = 175
4.)	write(b)				t_w = 200	
5.)	write(a)			t_w = 200		
6.)		write(c)				
		Abbruch				
7.)			write(a)			
			ignoriert			

Tabelle 13.15: Beispiel für Zeitstempelverfahren

16. Objektorientierte Datenbanken

Relationale Datenbanksysteme sind derzeit in administrativen Anwendungsbereichen marktbeherrschend, da sich die sehr einfache Strukturierung der Daten in flachen Tabellen als recht benutzerfreundlich erwiesen hat. Unzulänglichkeiten traten jedoch zutage bei komplexeren, ingenieurwissenschaftlichen Anwendungen, z.B. in den Bereichen CAD, Architektur und Multimedia.

Daraus ergaben sich zwei unterschiedliche Ansätze der Weiterentwicklung:

- Der *evolutionäre* Ansatz: Das relationale Modell wird um komplexe Typen erweitert zum sogenannten *geschachtelten* relationalen Modell.
- Der *revolutionäre* Ansatz: In Analogie zur Objektorientierten Programmierung wird in einem Objekttyp die *strukturelle* Information zusammen mit der *verhaltensmäsigen* Information integriert.

16.1 Schwächen relationaler Systeme

Die Relation

Buch:{[ISBN, Verlag, Titel, Autor, Version, Stichwort]}

erfordert bei 2 Autoren, 5 Versionen, 6 Stichworten für jedes Buch $2 \times 5 \times 6 = 60$ Einträge. Eine Aufsplittung auf mehrere Tabellen ergibt

Buch	:	{[ISBN, Titel, Verlag]}
Autor	:	{[ISBN, Name, Vorname]}
Version	:	{[ISBN, Auflage, Jahr]}
Stichwort	:	{[ISBN, Stichwort]}

Nun sind die Informationen zu einem Buch auf vier Tabellen verteilt. Beim Einfügen eines neuen Buches mus mehrmals dieselbe ISBN eingegeben werden. Die referentielle Integrität mus selbst überwacht werden. Eine Query der Form ``*Liste Bücher mit den Autoren Meier und Schmidt*`` ist nur sehr umständlich zu formulieren.

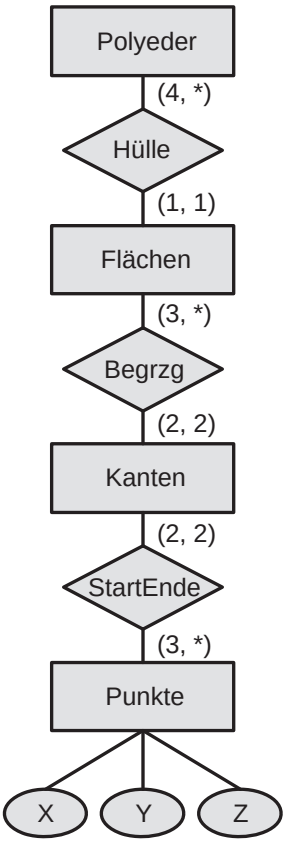


Abb. 112: (a) ER-Diagramm

Polyeder

PolyID	Gewicht	Material	...
cubo#5	25.765	Eisen	...
tetra#7	37.985	Glas	...
...	...	...	...

Flächen

FlächenID	PolyID	Oberflächen
f1	cubo#5	...
f2	cubo#5	...
...	...	...
f6	cubo#5	...
f7	tetra#7	...

Kanten

KantenID	F1	F2	P1	P2
k1	f1	f4	p1	p4
k2	f1	f2	p2	p3
...	...	...	...	...

Punkte

PunktId	X	Y	Z
p1	0.0	0.0	0.0
p2	1.0	0.0	0.0
...	...	...	...
p8	0.0	1.0	1.0
...	...	...	...

Abb. 113: (b) Relationales Schema

Abbildung 112 zeigt die Modellierung von Polyedern nach dem Begrenzungsflächenmodell, d. h. ein Polyeder wird beschrieben durch seine begrenzenden Flächen, diese wiederum durch ihre beteiligten Kanten und diese wiederum durch ihre beiden Eckpunkte. Abbildung 16.1b zeigt eine mögliche Umsetzung in ein relationales Schema, wobei die Beziehungen *Hülle, Begrzg* und *StartEnde* aufgrund der Kardinalitäten in die Entity-Typen integriert wurden.

Die relationale Modellierung hat etliche Schwachpunkte:

- **Segmentierung:** Ein Anwendungsobjekt wird über mehrere Relationen verteilt, die immer wieder durch einen Verbund zusammengefügt werden müssen.
- **Künstliche Schlüsselattribute:** Zur Identifikation von Tupeln müssen vom Benutzer relationenweit eindeutige Schlüssel vergeben werden.
- **Fehlendes Verhalten:** Das anwendungsspezifische Verhalten von Objekten, z.B. die Rotation eines Polyeders, findet im relationalen Schema keine Berücksichtigung.
- **Externe Programmierschnittstelle:** Die Manipulation von Objekten erfordert eine Programmierschnittstelle in Form einer Einbettung der (mengenorientierten) Datenbanksprache in eine (satzorientierte) Programmiersprache.

16.2 Vorteile der objektorientierten Modellierung

In einem objektorientierten Datenbanksystem werden *Verhaltens*- und *Struktur*-Beschreibungen in einem Objekt-Typ integriert. Das anwendungsspezifische Verhalten wird integraler Bestandteil der Datenbank. Dadurch können die umständlichen Transformationen zwischen Datenbank und Programmiersprache vermieden werden. Vielmehr sind die den Objekten zugeordneten Operationen direkt ausführbar, ohne detallierte Kenntnis der strukturellen Repräsentation der Objekte. Dies wird durch das *Geheimnisprinzip* (engl.: *information hiding*) unterstützt, wonach an der Schnittstelle des Objekttyps eine Kollektion von Operatoren angeboten wird, für deren Ausführung man lediglich die *Signatur* (Aufrufstruktur) kennen mus.

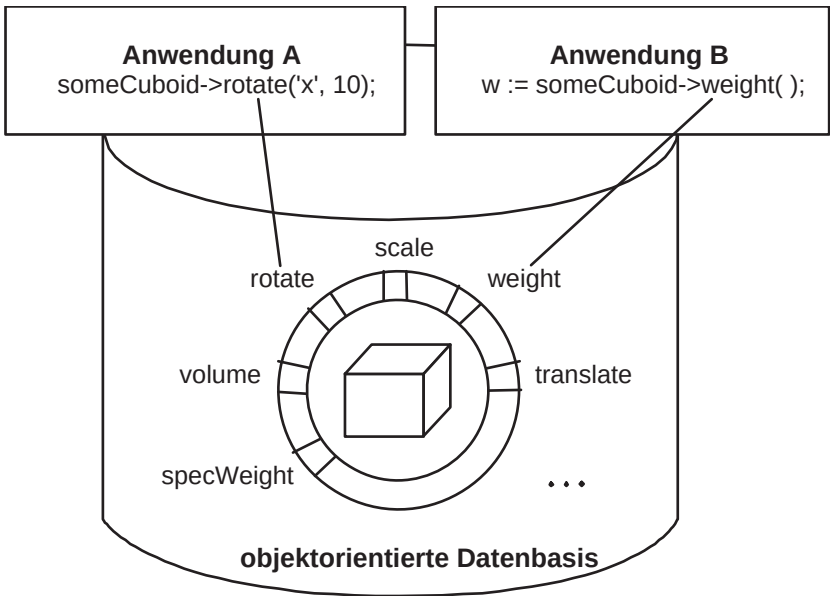


Abb. 114: Visualisierung der Vorteile der objektorientierten Datenmodellierung

114 visualisiert den objektorientierten Ansatz bei der Datenmodellierung. Ein Quader wird zusammen mit einer Reihe von Datenfeldern und Operatoren zur Verfügung gestellt. Unter Verwendung dieser Schnittstelle rotiert Anwendung A einen Quader und bestimmt Anwendung B das Gewicht.

16.3 Der ODMG-Standard

Im Gegensatz zum relationalen Modell ist die Standardisierung bei objektorientierten Datenbanksystemen noch nicht so weit fortgeschritten. Ein (de-facto) Standard wurde von der *Object Database Management Group* entworfen. Das ODMG-Modell umfast das objektorientierte Datenbanksystem und eine einheitliche Anbindung an bestehende Programmiersprachen. Bisher wurden Schnittstellen für C++ und Smalltalk vorgesehen. Auserdem wurde eine an SQL angelehnte deklarative Abfragesprache namens *OQL* (Object Query Language) entworfen.

16.4 Eigenschaften von Objekten

Im relationalen Modell werden Entitäten durch Tupel dargestellt, die aus atomaren *Literalen* bestehen.

Im objektorientierten Modell hat ein Objekt drei Bestandteile:

- **Identität:** Jedes Objekt hat eine systemweit eindeutige Objektidentität, die sich während seiner Lebenszeit nicht verändert.
- **Typ:** Der Objekttyp, auch *Klasse* genannt, legt die Struktur und das Verhalten des Objekts fest. Individuelle Objekte werden durch die *Instanziierung* eines Objekttyps erzeugt und heisen *Instanzen*. Die Menge aller Objekte (Instanzen) eines Typs wird als (Typ-)*Extension* (eng. *extent*) bezeichnet.
- **Wert** bzw. **Zustand:** Ein Objekt hat zu jedem Zeitpunkt seiner Lebenszeit einen bestimmten Zustand, auch Wert genannt, der sich aus der momentanen Ausprägung seiner Attribute ergibt.

115 zeigt einige Objekte aus der Universitätswelt. Dabei wird zum Beispiel der Identifikator id_1 als Wert des Attributs *gelesenVon* in der Vorlesung mit dem Titel *Grundzüge* verwendet, um auf die Person mit dem Namen *Kant* zu verweisen. Wertebereiche bestehen nicht nur aus atomaren Literalen, sondern auch aus Mengen. Zum Beispiel liest *Kant* zwei Vorlesungen, identifiziert durch id_2 und id_3.

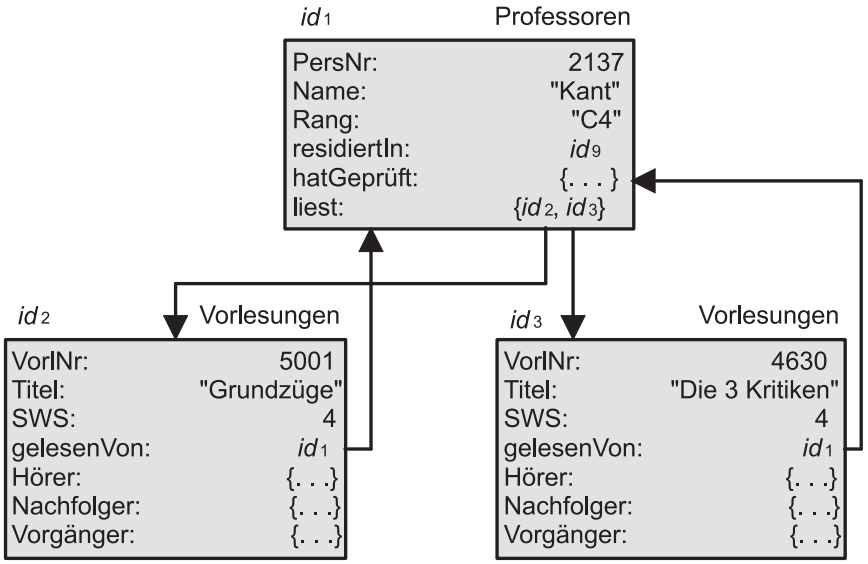


Abb. 115: Einige Objekte aus der Universitätswelt

Im relationalen Modell wurden Tupel anhand der Werte der Schlüsselattribute identifiziert (*identity through content*). Dieser Ansatz hat verschiedene Nachteile:

- Objekte mit gleichem Wert müssen nicht unbedingt identisch sein. Zum Beispiel könnte es zwei Studenten mit Namen ``*Willy Wacker*`` im 3. Semester geben.
- Aus diesem Grund müssen künstliche Schlüsselattribute ohne Anwendungsemantik (siehe Polyedermodellierung) eingeführt werden.

- Schlüssel dürfen während der Lebenszeit eines Objekts nicht verändert werden, da ansonsten alle Bezugnahmen auf das Objekt ungültig werden.

In Programmiersprachen wie Pascal oder C verwendet man Zeiger, um Objekte zu referenzieren. Dies ist für kurzlebige (transiente) Hauptspeicherobjekte akzeptabel, allerdings nicht für persistente Objekte.

Objektorientierte Datenbanksysteme verwenden daher zustands- und speicherungsort-unabhängige *Objektidentifikatoren* (OIDs). Ein OID wird vom Datenbanksystem systemweit eindeutig generiert, sobald ein neues Objekt erzeugt wird. Der OID bleibt dem Anwender verborgen, er ist unveränderlich und unabhängig vom momentanen Objekt-Zustand. Die momentane physikalische Adresse ergibt sich aus dem Inhalt einer Tabelle, die mit dem OID referiert wird.

Die Objekttyp-Definition enthält folgende Bestandteile:

- die Strukturbeschreibung der Instanzen, bestehend aus Attributen und Beziehungen zu anderen Objekten,
- die Verhaltensbeschreibung der Instanzen, bestehend aus einer Menge von Operationen,
- die Typeigenschaften, z.B. Generalisierungs- und Spezialisierungsbeziehungen.

16.5 Definition von Attributen

Die Definition des Objekttyps *Professoren* könnte wie folgt aussehen:

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

Attribute können strukturiert werden mit Hilfe des Tupelkonstruktors **struct{...}**:

```
class Person {
  attribute string Name;
  attribute struct Datum {
     short Tag;
     short Monat;
     short Jahr;
  } GebDatum;
};
```

16.6 Definition von Beziehungen

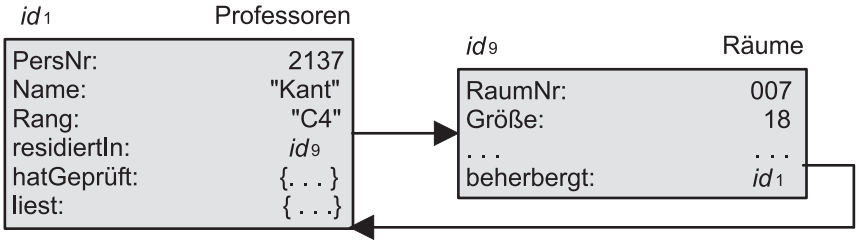


Abb. 116: Ausprägung einer 1:1-Beziehung

Eine $1:1$-Beziehung wird symmetrisch in beiden beteiligten Objekt-Typen modelliert:

```
class Professoren {
   @@attribute long PersNr;
   ...
   relationship Raeume residiertIn;
};


class Raeume {
  attribute long RaumNr;
  attribute short Groesse;
  ...
  relationship Professoren beherbergt;
};
```

116 zeigt eine mögliche Ausprägung der Beziehungen *residiertIn* und *beherbergt*.

Allerdings wird durch die gegebene Klassenspezifikation weder die Symmetrie noch die 1:1-Einschränkung garantiert. 117 zeigt einen inkonsistenten Zustand des Beziehungspaars *residiertIn* und *beherbergt*.

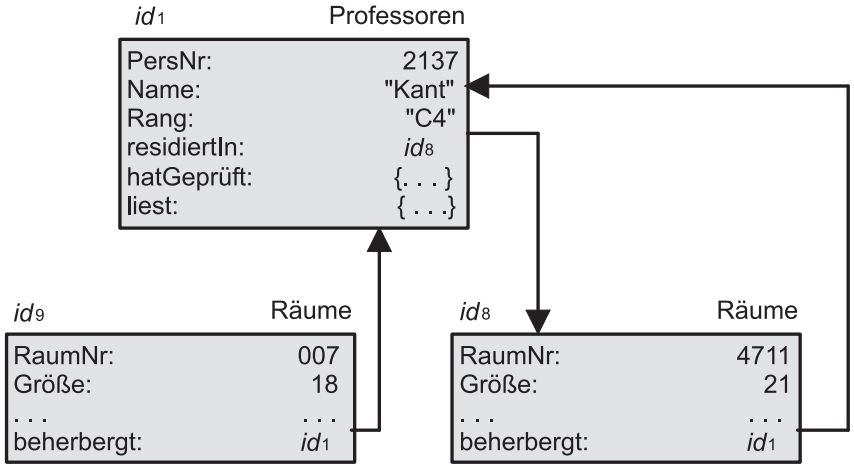


Abb. 117: Inkonsistenter Zustand einer Beziehung

Um Inkonsistenzen dieser Art zu vermeiden, wurde im ODMG-Objektmodell das **inverse**-Konstrukt integriert:

```
class Professoren {
  attribute long PersNr;
  ...
  relationship Raeume residiertIn inverse Raeume::beherbergt;
};


class Raeume {
  attribute long RaumNr;
    attribute short Groesse;
    ..
    relationship Professoren beherbergt inverse Professoren::residiertIn;
};
```

Damit wird sichergestellt, das immer gilt:

$$p = r.beherbergt[[\ ]] \Leftrightarrow [[\ ]]r = p.residiertIn$$

Binäre *1:N* - Beziehungen werden modelliert mit Hilfe des Mengenkonstruktors **set**, der im nächsten Beispiel einem Professor eine Menge von Referenzen auf *Vorlesungen*-Objekte zuordnet:

```
(:class Professoren {
```

```
    ...
    relationship set (Vorlesungen) liest inverse Vorlesungen::gelesenVon;
};

class Vorlesungen {
    ...
    relationship Professoren gelesenVon inverse Professoren::liest;
};
```

Man beachte, das im relationalen Modell die Einführung eines Attributs *liest* im Entity-Typ *Professoren* die Verletzung der 3. Normalform verursacht hätte.

Binäre *N:M* - Beziehungen werden unter Verwendung von zwei **set**-Konstruktoren modelliert:

```
class Studenten {
    ...
    relationship set (Vorlesungen) hoert inverse Vorlesungen::Hoerer;
};

class Vorlesungen {
    ...
    relationship set (Studenten) Hoerer inverse Studenten::hoert;
};
```

Durch die **inverse**-Spezifikation wird sichergestellt, das gilt:

$$s \in v.Hoerer[[\ ]] \Leftrightarrow [[\ ]]v \in s.hoert$$

Analog lassen sich rekursive $N : M$ - Beziehungen beschreiben:

```
class Vorlesungen {
    ...
    relationship set (Vorlesungen) Vorgaenger inverse Vorlesungen::Nachfolger;
    relationship set (Vorlesungen) Nachfolger inverse Vorlesungen::Vorgaenger; };
```

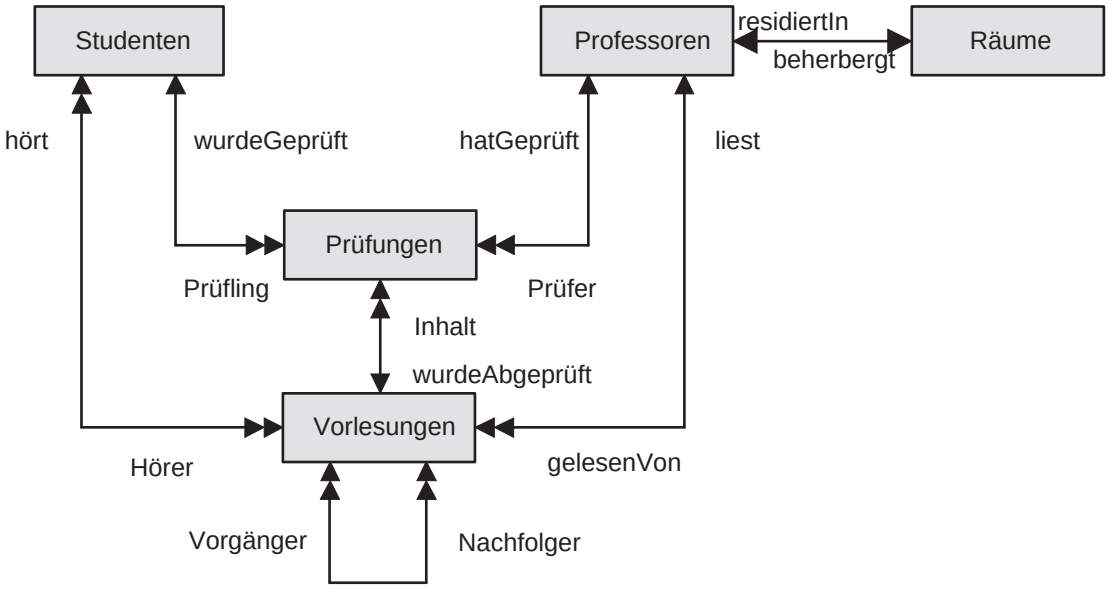


Abb. 118: Modellierungen von Beziehungen im Objektmodell

Ternäre oder $n \geq 3$ stellige Beziehungen benötigen einen eigenständigen Objekttyp, der die Beziehung repräsentiert. Zum Beispiel wird die ternäre Beziehung

pruefen : {[MatrNr, VorlNr, PersNr, Note]}

zwischen den *Studenten*, *Vorlesungen* und *Professoren* wie folgt modelliert

```
class Pruefungen {
  attribute    float       Note;
  relationship Professoren Pruefer   inverse Professoren::hatgeprueft;
  relationship Studenten   Pruefling inverse Studenten::wurdegeprueft;
  relationship Vorlesungen Inhalt    inverse Vorlesungen::wurdeAbgeprueft;
};


class Professoren {
  attribute    long             PersNr;
  attribute    string           Name;
  attribute    string           Rang;
  relationship Raeume           residiertIn inverse Raeume::beherbergt;
  relationship set<Vorlesungen> liest       inverse Vorlesungen::gelesenVon;
  relationship set<Pruefungen>  hatgeprueft inverse Pruefungen::Pruefer;
};


class Vorlesungen {
  attribute    long             VorlNr;
  attribute    string           Titel;
  attribute    short            SWS;
  relationship Professoren      gelesenVon inverse Professoren::liest;
  relationship set<Studenten>   Hoerer     inverse Studenten::hoert;
  relationship set<Vorlesungen> Nachfolger inverse Vorlesungen::Vorgaenger;
  relationship set<Vorlesungen> Vorgaenger inverse Vorlesungen::Nachfolger;
  relationship set<Pruefungen>  wurdeAbgeprueft inverse Pruefungen::Inhalt;
};


class Studenten {
  attribute    long             MatrNr;
  attribute    string           Name;
  attribute    short            Semester;
  relationship set<Pruefungen>  wurdeGeprueft inverse Pruefungen::Pruefling;
  relationship set<Vorlesungen> hoert         inverse Vorlesungen::Hoerer;
};
```

Abbildung 118 visualisiert die bislang eingeführten Beziehungen. Die Anzahl der Pfeilspitzen gibt die Wertigkeit der Beziehung an:

$\leftrightarrow$ bezeichnet eine $1:1$-Beziehung $\leftrightarrow\!\!\!\rightarrow$ bezeichnet eine $1:N$-Beziehung

$\leftarrow\!\!\!\rightarrow$ bezeichnet eine $N:1$-Beziehung $\leftarrow\!\!\!\rightarrow$ bezeichnet eine $N:M$-Beziehung

16.7 Extensionen und Schlüssel

Eine *Extension* ist die Menge aller Instanzen eines Objekt-Typs incl. seiner spezialisierten Untertypen (siehe später). Sie kann verwendet werden für Anfragen der Art *Suche alle Objekte eines Typs, die eine bestimmte Bedingung erfüllen*. Man kann zu einem Objekttyp auch *Schlüssel* definieren, deren Eindeutigkeit innerhalb der Extension gewährleistet wird. Diese Schlüsselinformation wird jedoch nur als Integritätsbedingung verwendet und nicht zur Referenzierung von Objekten:

```
class Studenten (extent AlleStudenten key MatrNr) {
    attribute long MatrNr;
```

```
    attribute string Name;
    attribute short Semester;
    relationship set(Vorlesungen) hoert inverse Vorlesungen::Hoerer;
    relationship set(Pruefungen) wurdeGeprueft inverse Pruefungen::Pruefling;
};
```

16.8 Modellierung des Verhaltens

Der Zugriff auf den Objektzustand und die Manipulation des Zustands geschieht über eine *Schnittstelle*. Die Schnittstellenoperationen können

- ein Objekt erzeugen (instanziieren) und initialisieren mit Hilfe eines *Konstruktors*,
- freigegebene Teile des Zustands erfragen mit Hilfe eines *Observers*,
- konsistenzerhaltende Operationen auf einem Objekt ausführen mit Hilfe eines *Mutators*,
- das Objekt zerstören mit Hilfe eines *Destruktors*.

Die Aufrufstruktur der Operation, genannt *Signatur*, legt folgendes fest:

- Name der Operation,
- Anzahl und Typ der Parameter,
- Typ des Rückgabewerts, falls vorhanden, sonst **void**,
- ggf. die durch die Operation ausgelöste *Ausnahme* (engl. *exception*).

Beispiel:

```
class Professoren {
    exception hatNochNichtGeprueft { };
    exception schonHoechsteStufe { };
    ...
    float wieHartAlsPruefer() raises (hatNochNichtgeprueft);
    void befoerdert() raises (schonHoechsteStufe);
};
```

Hierdurch wird der Objekttyp *Professoren* um zwei Signaturen erweitert:

- Der Observer *wieHartalsPruefer* liefert die Durchschnittsnote und stöst die Ausnahmebehandlung *hatNochNichtGeprueft* an, wenn keine Prüfungen vorliegen.
- Der Mutator *befoerdert* erhöht den Rang um eine Stufe und stöst die Ausnahmebehandlung *schonHoechsteStufe* an, wenn bereits die Gehaltsstufe C4 vorliegt.

Man bezeichnet den Objekttyp, auf dem die Operationen definiert wurden, als *Empfängertyp* (engl *receiver type*) und das Objekt, auf dem die Operation aufgerufen wird, als *Empfängerobjekt*.

Die Aufrufstruktur hängt von der Sprachanbindung ab. Innerhalb von C++ würde befördert aufgerufen als

```
meinLieblingsProf->befoerdert();
```

In der deklarativen Anfragesprache OQL ist der Aufruf wahlweise mit Pfeil (`->`) oder mit einem Punkt (`.`)durchzuführen:

```
select p.wieHartAlsPruefer()
from p in AlleProfessoren
where p.Name ='Kant';
```

16.9 Vererbung

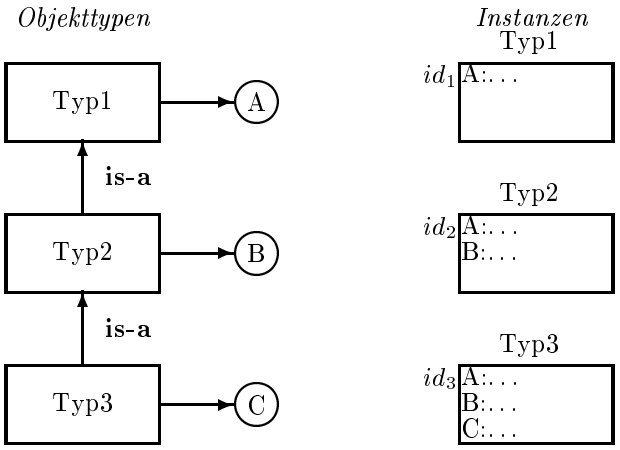


Abb. 119: Schematische Darstellung einer abstrakten Typhierarchie

Das in Kapitel 2 eingeführte Konzept der Generalisierung bzw. Spezialisierung läst sich bei objektorientierten Datenbanksystemen mit Hilfe der *Vererbung* lösen. Hierbei erbt der Untertyp nicht nur die Struktur, sondern auch das Verhalten des Obertyps. Auserdem sind Instanzen des Untertyps überall dort einsetzbar (*substituierbar*), wo Instanzen des Obertyps erforderlich sind.

119 zeigt eine Typhierarchie, bei der die Untertypen *Typ2* und *Typ3* jeweils ein weiteres Attribut, nämlich B bzw. C aufweisen. Operationen sind hier gänzlich auser Acht gelassen. Instanzen des Typs *Typ3* gehören auch zur Extension von Typ *Typ2* und von Typ *Typ1*.

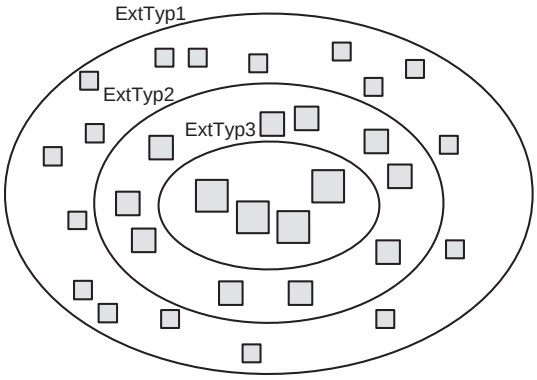


Abb. 120: Darstellung der Subtypisierung

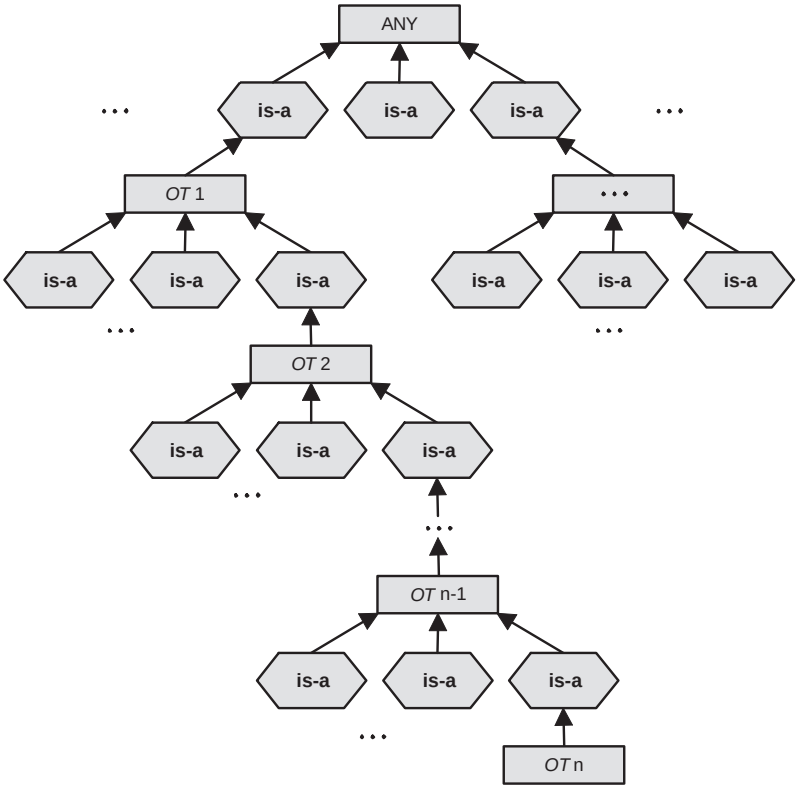


Abb. 121: Abstrakte Typhierachie bei einfacher Vererbung

120 zeigt die geschachtelte Anordung der drei Extensionen der Typen *Typ1*, *Typ2* und *Typ3*. Durch die unterschiedliche Kästchengröse soll angedeutet werden, das Untertyp-Instanzen mehr Eigenschaften haben und daher mehr wissen als die direkten Instanzen eines Obertyps.

Man unterscheidet zwei unterschiedliche Arten der Vererbung:

- *Einfachvererbung* (*single inheritance*): Jeder Objekttyp hat höchstens einen direkten Obertyp.
- *Mehrfachvererbung* (*multiple inheritance*): Jeder Objekttyp kann mehrere direkte Obertypen haben.

121 zeigt eine abstrakte Typhierarchie mit Einfachvererbung. Der Vorteil der Einfachvererbung gegenüber der Mehrfachvererbung besteht darin, das es für jeden Typ einen eindeutigen Pfad zur Wurzel der Typhierarchie (hier: ANY) gibt. Ein derartiger Super-Obertyp findet sich in vielen Objektmodellen, manchmal wird er *object* genannt, in der ODMG C++-Einbindung heist er *d_Object*.

16.10 Beispiel einer Typhierarchie

Wir betrachten eine Typhierarchie aus dem Universitätsbereich. *Angestellte* werden spezialisiert zu *Professoren* und *Assistenten*:

```
class Angestellte (extent AlleAngestellte) {
    attribute long PersNr;
    attribute string Name;
    attribute date GebDatum;
    short Alter();
    long Gehalt();
};


class Assistenten extends Angestellte (extent AlleAssistenten) {
    attribute string Fachgebiet;
};
```

```
class Professoren extends Angestellte (extent AlleProfessoren) {
    attribute string Rang;
    relationship Raeume residiertIn inverse Raeume::beherbergt;
    relationship set(Vorlesungen) liest inverse Vorlesungen::gelesenVon;
    relationship set(Pruefungen) hatgeprueft inverse Pruefungen::Pruefer;
};
```

Abbildung 122 zeigt die drei Objekttypen *Angestellte*, *Professoren* und *Assistenten*, wobei die geerbten Eigenschaften in den gepunkteten Ovalen angegeben ist.

Abbildung 123 zeigt schematisch die aus der Ober-/Untertyp-Beziehung resultierende Inklusion der Extensionen *AlleProfessoren* und *AlleAssistenten* in der Extension *AlleAngestellte*.

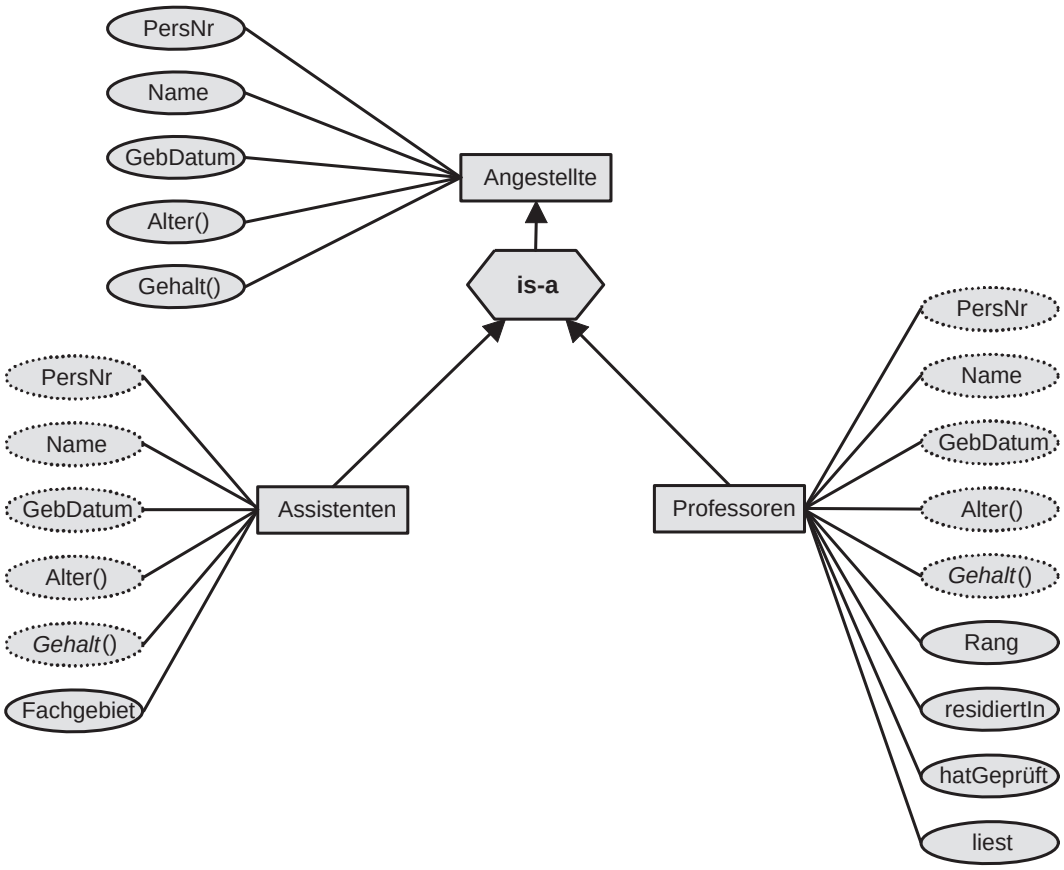


Abb. 122: Vererbung von Eigenschaften

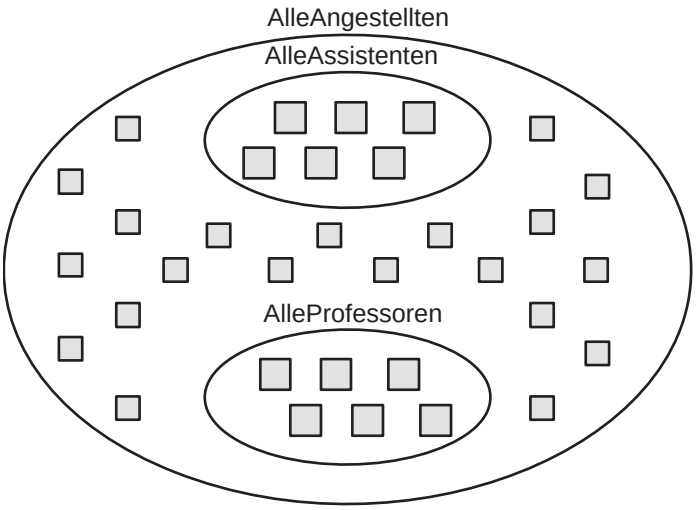


Abb. 123: Visualisierung der Extensionen

16.11 Verfeinerung und spätes Binden

Genau wie Attribute werden auch Operationen vom Obertyp an alle Untertypen vererbt. Zum Beispiel steht der bei *Angestellte* definierte Observer *Gehalt()* auch bei den Objekttypen *Professoren* und *Assistenten* zur Verfügung.

Wir hätten auch eine *Verfeinerung* bzw. *Spezialisierung* (engl. *Refinement*) vornehmen können. Das Gehalt würde danach für jeden Objekttyp unterschiedlich berechnet:

- Angestellte erhalten 2000 + (Alter() - 21) * 100 Euro,
- Assistenten erhalten 2500 + (Alter() - 21) * 125 Euro,
- Professoren erhalten 3000 + (Alter() - 21) * 150 Euro.

In Abbildung 122 ist dies durch den kursiven Schrifttyp der geerbten *Gehalt*-Eigenschaft gekennzeichnet.

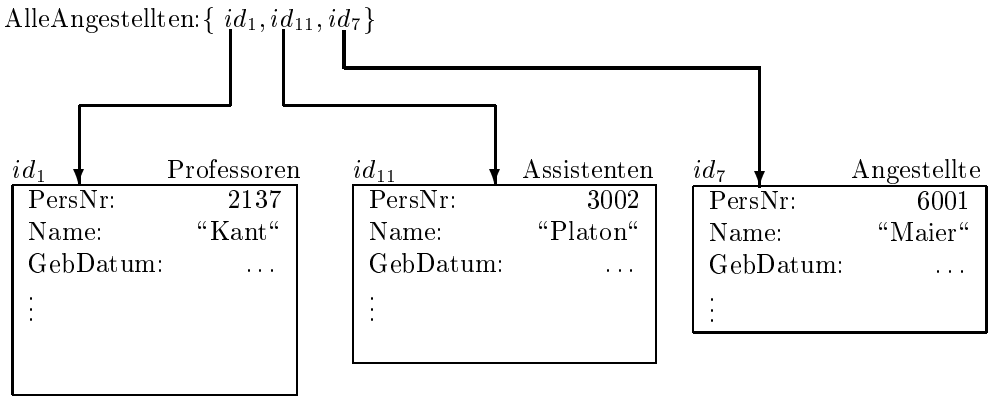


Abb. 124: Die Extension AlleAngestellten mit drei Objekten

Abbildung 124 zeigt die Extension *AlleAngestellten* mit drei Elementen:

- Objekt id_1 ist eine direkte *Professoren*-Instanz,
- Objekt id_1 ist eine direkte *Assistenten*-Instanz,
- Objekt id_7 ist eine direkte *Angestellte*-Instanz.

Es werde nun die folgende Query abgesetzt:

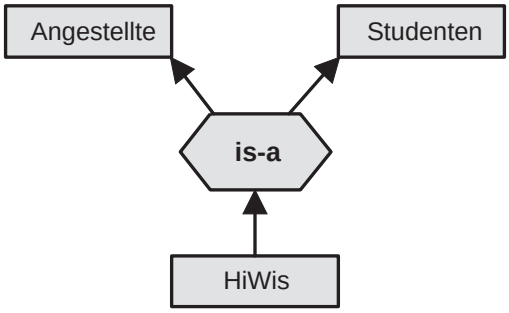
```
select sum(a.Gehalt())
from a in AlleAngestellten;
```

Offensichtlich kann erst zur Laufzeit die jeweils spezialisierteste Version von *Gehalt* ermittelt werden. Diesen Vorgang bezeichnet man als *spätes Binden* (engl. *late binding*).

16.12 Mehrfachvererbung

Bei der Mehrfachvererbung erbt ein Objekttyp die Eigenschaften von mehreren Obertypen. Abbildung 16.13 zeigt ein Beispiel dafür. Der Objekttyp *Hiwi* erbt

- von *Angestellte* die Attribute *PersNr*, *Name* und *GebDatum* sowie die Operationen *Gehalt()* und *Alter()*,
- von *Studenten* die Attribute *MatrNr, Name, Semester, hört* und *wurdeGeprüft*.



Beispiel für Mehrfachvererbung

Die Syntax könnte lauten:

```
class HiWis extends Studenten, Angestellte (extent AlleHiwis) {
    attribute short Arbeitsstunden;
    ...
};
```

Nun wird allerdings das Attribut *Name* sowohl von *Angestellte* als auch von *Studenten* geerbt. Um solchen Mehrdeutigkeiten und den damit verbundenen Implementationsproblemen aus dem Weg zu gehen, wurde in der Version 2.0 von ODL das Schnittstellen-Konzept (engl *interface*) eingeführt, das es in ähnlicher Form auch in der Programmiersprache *Java* gibt.

Eine **interface**-Definition ist eine abstrakte Definition der Methoden, die alle Klassen besitzen müssen, die diese Schnittstelle implementieren. Eine Klasse im ODBG-Modell kann mehrere Schnittstellen implementieren, darf aber nur höchstens von einer Klasse mit **extends** abgeleitet werden

Also würde man für die Angestellten lediglich die Schnittstelle *AngestellteIF* festlegen. Die Klasse *HiWis* implementiert diese Schnittstelle und erbt den Zustand und die Methoden der Klasse *Studenten*. Die Liste der Schnittstellen, die eine Klasse implementiert, wird in der Klassendefinition nach dem Klassennamen und der möglichen **extends**-Anweisung hinter einem Doppelpunkt angegeben. Zusätzlich mus der nicht mitgeerbte, aber benötigte Teil des Zustandes der ursprünglichen *Angestellten*-Klasse nachgereicht werden.

```
interface AngestellteIF {
    short Alter();
    long Gehalt();
};
```

```
class Angestellte : AngestellteIF (extent AlleAngestellte) {
    attribute long PersNr;
    attribute string Name;
    attribute date GebDatum;
};


class Hiwis extends Studenten : AngestellteIF (extent AlleHiwis) {
    attribute long PersNr;
    attribute date GebDatum;
    attribute short Arbeitsstunden;
};
```

Man beachte, dass die *HiWis* nun nicht in der Extension *AlleAngestellten* enthalten sind. Dazu müste man diese Extension der Schnittstelle *AngestellteIF* zuordnen, was aber nach ODMG-Standard nicht möglich ist. Konflikte bei gleichbenannten Methoden werden im ODBG-Modell dadurch vermieden, das Ableitungen, bei denen solche Konflikte auftreten würden, verboten sind.

16.13 Die Anfragesprache OQL

OQL (Object Query Language) ist eine an SQL angelehnte Abfragesprache. Im Gegensatz zu SQL existiert kein Update-Befehl. Veränderungen an der Datenbank können nur über die Mutatoren der Objekte durchgeführt werden.

Liste alle C4-Professoren (als Objekte):

```
select p
from p in AlleProfessoren
where p.Rang = \"{C}4";
```

Liste Name und Rang aller C4-Professoren (als Werte):

```
select p.Name, p.Rang
from p in AlleProfessoren
where p.Rang = "C4";
```

Liste Name und Rang aller C4-Professoren (als Tupel):

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

Liste alle Angestellte mit einem Gehalt über 100.000 Euro:

```
select a.Name
from a in AlleAngestellte
where a.Gehalt() > 100.000;
```

Liste Name und Lehrbelastung aller Professoren:

```
select struct (n: p.Name, a: sum(select v.SWS from v in p.liest))
from p in AlleProfessoren;
```

Gruppiere alle Vorlesungen nach der Semesterstundenzahl:

```
select *
from v in AlleVorlesungen
group by kurz: v.SWS <= 2, mittel: v.SWS = 3, lang: v.SWS >= 4;
```

Das Ergebnis sind drei Tupel vom Typ

```
struct (kurz: boolean,
        mittel: boolean,
        lang: boolean,
        partition : bag(struct(v: Vorlesungen)))
```

mit dem mengenwertigen Attribut *partition*, welche die in die jeweilige Partition fallenden Vorlesungen enthält. Die booleschen Werte markieren, um welche Partition es sich handelt. Liste die Namen der Studenten, die bei Sokrates Vorlesungen hören:

```
select s.Name
from s in AlleStudenten, v in s.hoert
where v.gelesenVon.Name = "Sokrates"
```

Die im relationalen Modell erforderlichen Joins wurden hier mit Hilfe von Pfadausdrücken realisiert. Abbildung 125 zeigt die graphische Darstellung des Pfadausdrucks von *Studenten* über *Vorlesungen* zu *Professoren*.

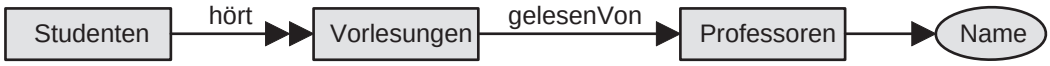


Abb. 125: Graphische Darstellung eines Pfadausdrucks

Der Ausdruck *s.hoert* ergibt die Menge von Vorlesungen des Studenten s. Pfadausdrücke können beliebige Länge haben, dürfen aber keine mengenwertigen Eigenschaften verwenden. Verboten ist daher eine Formulierung der Form

```
s.hoert.gelesenVon.Name
```

da *hoert* mengenwertig ist. Stattdessen wurde in der from-Klausel die Variable v eingeführt, die jeweils an die Menge *s.hoert* gebunden wird.

Die Erzeugung von Objekten geschieht mit Hilfe des Objektkonstruktors:

```
select p.wieHartAlsPruefer()
from p in AlleProfessoren
where p.Name ='Kant';
```

16.14 C++-Einbettung

Zur Einbindung von objektorientierten Datenbanksystemen in eine Programmiersprache gibt es drei Ansätze:

- **Entwurf einer neuen Sprache**
- **Erweiterung einer bestehenden Sprache**
- **Datenbankfähigkeit durch Typ-Bibliothek**

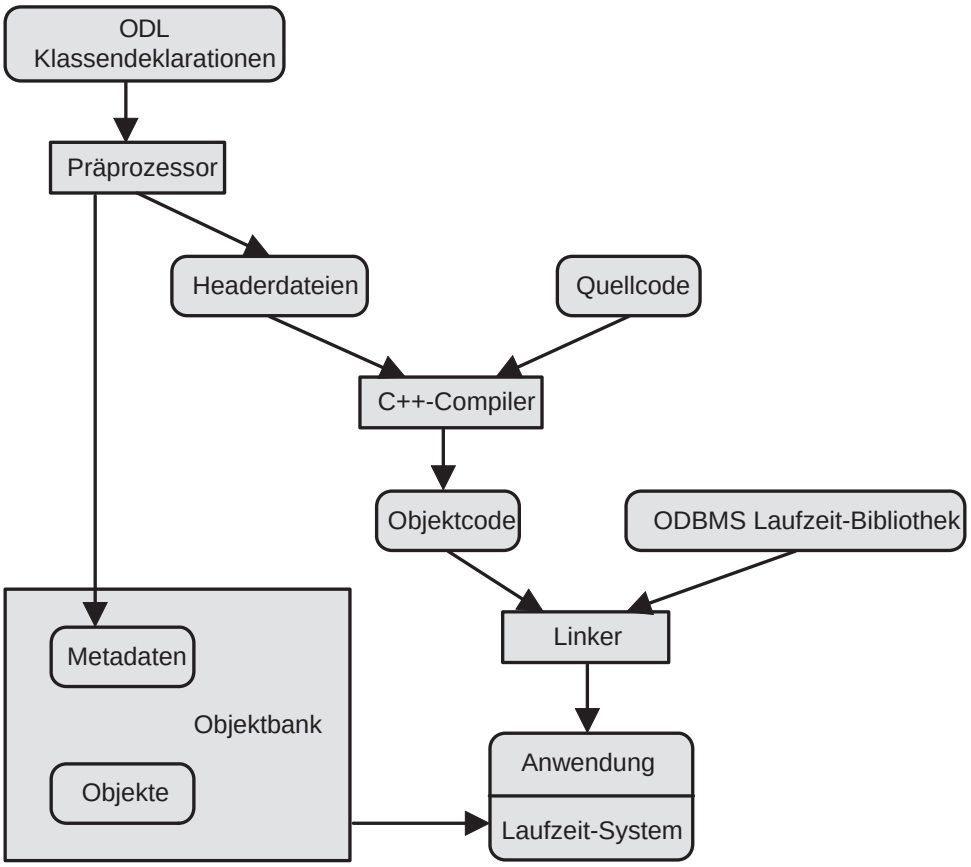


Abb. 126: C++-Einbindung

Die von der ODMG gewählte Einbindung entspricht dem dritten Ansatz. Ihre Realisierung ist in 126 dargestellt. Der Benutzer erstellt die Klassendeklarationen und den Quellcode der Anwendung. Die Klassendeklarationen werden mit Hilfe eines Präprozessors in die Datenbank eingetragen. Zusätzlich werden Header-Dateien in Standard-C++ erzeugt, die durch einen handelsüblichen C++-Compiler übersetzt werden können. Der Quellcode enthält die Realisierungen der den Objekttypen zugeordneten Operationen. Der übersetzte Quellcode wird mit dem Laufzeitsystem gebunden. Das Laufzeitsystem sorgt in der fertigen Anwendung für die Kommunikation mit der Datenbank.

Zur Formulierung von Beziehungen zwischen persistenten Objekten bietet die C++-Einbettung die Typen *d_Rel_Ref* und *d_Rel_Set* an:

```
const char _liest[] = "liest";
const char _gelesenVon[] = "gelesenVon";

class Vorlesungen : public d_Object {
    d_String Titel;
    d_Short SWS;
    ...
    d_Rel_ref <Professoren, _liest> gelesenVon;
}

class Professoren : public Angestellte {
    d_Long PersNr;
    ...
    d_Rel_Set <Vorlesungen, _gelesenVon> liest;
}
```

Es wurden hier zwei Klassen definiert. *Vorlesungen* ist direkt vom Typ *d_Object* abgeleitet, *Professoren* ist über *d_Object* und dann über *Angestellte* (nicht gezeigt) abgeleitet. Der Typ *d_object* sorgt dafür, das von *Vorlesungen* und *Professoren* nicht nur transiente, sondern auch persistente Instanzen gebildet werden können. Die Typen *d_String*, *d_Short* und *d_Long* sind die C++-Versionen der ODL-Typen **string**, **short** und **long**.

In der Klasse *Vorlesungen* referenziert das Attribut *gelesenVon* durch *d_Rel_Ref* ein Objekt vom Typ *Professoren*. Als zweites Argument in der Winkelklammer wird die entsprechende inverse Abbildung *liest* angegeben. In der Klasse *Professoren* referenziert das Attribut *liest* durch *d_Rel_Set* eine Menge von Objekten vom Typ *Vorlesungen*. Als zweites Argument in der Winkelklammer wird die entsprechende inverse Abbildung *gelesenVon* angegeben.

Zum Erzeugen eines persistenten Objekts verwendet man den Operator **new**:

```
d_Ref <Professoren> Russel = new(UniDB,"Professoren") Professoren (2126,"Russel","C4", ...);
```

Hierbei ist *Russel* eine Variable vom Typ *d_Ref* bezogen auf *Professoren*, die auf das neue Objekt verweist (im Gegensatz zu *d_Rel_Ref* ohne inverse Referenz). Als zweites Argument wird der Name des erzeugten Objekttypen als Zeichenkette angegeben. Als Beispiel einer Anfrage wollen wir alle Schüler eines bestimmten Professors ermitteln:

```
d_Bag <Studenten> Schüler;
char * profname = ...;
d_OQL_Query anfrage ("select s
    from s in v.Hörer,
    v in p.liest,
    p in AlleProfessoren
    where p.Name = $1");
anfrage << profname;
d_oql_execute(anfrage, Schüler);
```

Zunächst wird ein Objekt vom Typ *d_OQL_Query* erzeugt mit der Anfrage als Argument in Form eines Strings.Hierbei können Platzhalter für Anfrageparameter stehen; an Stelle von $1 wird der erste übergebene Parameter eingesetzt. Dies geschieht mit dem <<-Operator der Klasse *d_OQL_Query*. Die Anfrage wird mit der Funktion *d_oql_execute* ausgeführt und das Ergebnis in der Kollektionsvariablen vom Typ *d_Bag* (Multimenge mit Duplikaten) zurückgeliefert.

17. Sicherheit

In diesem Kapitel geht es um den Schutz gegen absichtliche Beschädigung oder Enthüllung von sensiblen Daten. Abbildung 127 zeigt die hierarchische Kapselung verschiedenster Masnahmen.

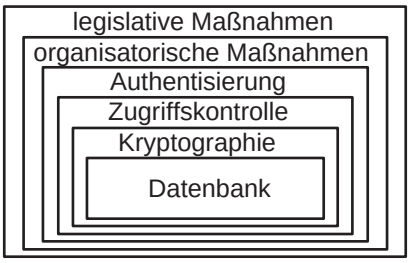


Abb. 127: Ebenen des Datenschutzes

17.1 Legislative Masnahmen

Im *Gesetz zum Schutz vor Misbrauch personenbezogener Daten bei der Datenverarbeitung* wurde 1978 festgelegt, welche Daten in welchem Umfang schutzbedürftig sind.

17.2 Organisatorische Masnahmen

Darunter fallen Masnahmen, um den persönlichen Zugang zum Computer zu regeln:

- bauliche Masnahmen
- Pförtner
- Ausweiskontrolle
- Diebstahlsicherung
- Alarmanlage

17.3 Authentisierung

Darunter fallen Masnahmen zur Überprüfung der Identität eines Benutzers:

- Magnetkarte
- Stimmanalyse/Fingerabdruck
- Paswort: w ohne Echo eintippen, System überprüft, ob $f(w)$ eingetragen ist, f^{-1} aus f nicht rekonstruierbar
- dynamisches Paswort: vereinbare Algorithmus, der aus Zufallsstring gewisse Buchstaben heraussucht

Paswortverfahren sollten mit Überwachungsmasnahmen kombiniert werden (Ort, Zeit, Fehleingabe notieren)

17.4 Zugriffskontrolle

Verschiedene Benutzer haben verschiedene Rechte bzgl. derselben Datenbank. Tabelle 14.1 zeigt eine Berechtigungsmatrix (wertunabhängig):

Benutzer	Ang-Nr	Gehalt	Leistung
A (Manager)	R	R	RW
B (Personalchef)	RW	RW	R
C (Lohnbüro)	R	R	--

Tabelle 14.1: Berechtigungsmatrix

Bei einer wertabhängigen Einschränkung wird der Zugriff von der aktuellen Ausprägung abhängig gemacht:

Zugriff (A, Gehalt): R: Gehalt $<$ 10.000
W: Gehalt $<$ 5.000

Dies ist natürlich kostspieliger, da erst nach Lesen der Daten entschieden werden kann, ob der Benutzer die Daten lesen darf. Ggf. werden dazu Tabellen benötigt, die für die eigentliche Anfrage nicht verlangt waren. Beispiel: Zugriff verboten auf Gehälter der Mitarbeiter an Projekt 007. Eine Möglichkeit zur Realisierung von Zugriffskontrollen besteht durch die Verwendung von Sichten:

```
define view v(angnr, gehalt) as
select angnr, gehalt from angest where
gehalt < 3000
```

Eine andere Realisierung von Zugriffskontrollen besteht durch eine Abfragemodifikation.

Beispiel:

Die Abfrageeinschränkung

```
deny (name, gehalt) where gehalt > 3000
```

liefert zusammen mit der Benutzer-Query

```
select gehalt from angest where name = 'Schmidt'
```

die generierte Query

```
select gehalt from angest
where name = 'Schmidt' and not gehalt > 3000
```

In statistischen Datenbanken dürfen Durchschnittswerte und Summen geliefert werden, aber keine Aussagen zu einzelnen Tupeln. Dies ist sehr schwer einzuhalten, selbst wenn die Anzahl der referierten Datensätze gros ist.

Beispiel:

Es habe Manager X als einziger eine bestimmte Eigenschaft, z. B. habe er das höchste Gehalt. Dann läst sich mit folgenden beiden Queries das Gehalt von Manager X errechnen, obwohl beide Queries alle bzw. fast alle Tupel umfassen:

```
select sum (gehalt) from angest;
select sum (gehalt) from angest
where gehalt < (select max(gehalt) from angest);
```

In SQL-92 können Zugriffsrechte dynamisch verteilt werden, d. h. der Eigentümer einer Relation kann anderen Benutzern Rechte erteilen und entziehen.

Die vereinfachte Syntax lautet:

`grant { select | insert | delete | update | references | all } on <relation> to <user> [with grant option]`

Hierbei bedeuten

`select`:	darf Tupel lesen
`insert`:	darf Tupel einfügen
`delete`:	darf Tupel löschen
`update`:	darf Tupel ändern
`references`:	darf Fremdschlüssel anlegen
`all` :	`select + insert + delete + update + references`
`with grant option`:	`<user>` darf die ihm erteilten Rechte weitergeben

Beispiel:

A: `grant read, insert on angest to` B `with grant option`

B: `grant read on angest to` C `with grant option`

B: `grant insert on angest to` C

Das Recht, einen Fremdschlüssel anlegen zu dürfen, hat weitreichende Folgen: Zum einen kann das Entfernen von Tupeln in der referenzierten Tabelle verhindert werden. Zum anderen kann durch das probeweise Einfügen von Fremdschlüsseln getestet werden, ob die (ansonsten lesegeschützte) referenzierte Tabelle gewisse Schlüsselwerte aufweist:

```
create table Agententest(Kennung character(3) references Agenten);
```

Jeder Benutzer, der ein Recht vergeben hat, kann dieses mit einer *Revoke*-Anweisung wieder zurücknehmen:

`revoke { select | insert | delete | update | references | all } on <relation> from <user>`

Beispiel:

B: `revoke all on angest from` C Es sollen dadurch dem Benutzer C alle Rechte entzogen werden, die er von B erhalten hat, aber nicht solche, die er von anderen Benutzern erhalten hat. Auserdem erlöschen die von C weitergegebenen Rechte.

Der Entzug eines Grant `G` soll sich so auswirken, als ob `G` niemals gegeben worden wäre!

Beispiel:

A: `grant read, insert, update on angest to` D

B: `grant read, update on angest to` D `with grant option`

D: `grant read, update on angest to` E

A: `revoke insert, update on angest from` D

Hierdurch verliert D sein `insert`-Recht, darf aber sein `update`-Recht behalten, weil es auch noch von B verliehen worden war. E verliert keine Rechte.

Folgt nun der Befehl

B: `revoke update on angest from` D

so muss D sein Update-Recht abgeben und als Konsequenz daraus muss auch E sein Update-Recht abgeben.

17.5 Auditing

Auditing bezeichnet die Möglichkeit, über Operationen von Benutzern Buch zu führen. Einige (selbsterklärende) Kommandos in SQL-92:

```
audit delete any table;
noaudit delete any table;
audit update on erika.professoren whenever not successful;
```

Der resultierende *Audit-Trail* wird in diversen Systemtabellen gehalten und kann von dort durch spezielle Views gesichtet werden.

17.6 Kryptographie

Da die meisten Datenbanken in einer verteilten Umgebung (Client/Server) betrieben werden, ist die Gefahr des Abhörens von Kommunikationskanälen sehr hoch. Zur Authentisierung von Benutzern und zur Sicherung gegen den Zugriff auf sensible Daten werden daher *kryptographische Methoden* eingesetzt.

Der prinzipielle Ablauf ist in Abbildung 128 skizziert: Der Klartext x dient als Eingabe für ein Verschlüsselungsverfahren *encode*, welches über einen Schlüssel e parametrisiert ist. Das heist, das grundsätzliche Verfahren der Verschlüsselung ist allen Beteiligten bekannt, mit Hilfe des Schlüssels e kann der Vorgang jeweils individuell beeinflust werden. Auf der Gegenseite wird mit dem Verfahren *decode* und seinem Schlüssel d der Vorgang umgekehrt und somit der Klartext rekonstruiert.

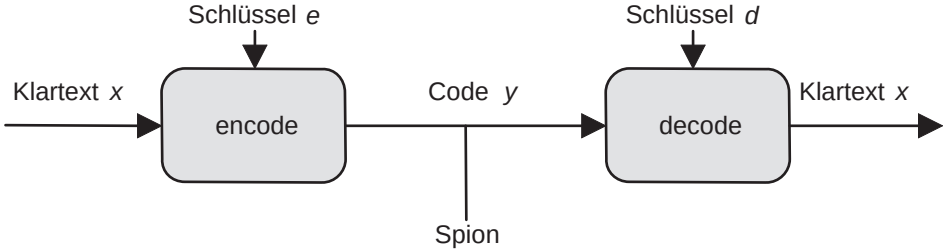


Abb. 128: Ablauf beim Übertragen einer Nachricht

Zum Beispiel kann eine Exclusive-OR-Verknüpfung des Klartextes mit dem Schlüssel verwendet werden, um die Chiffre zu berechnen. Derselbe Schlüssel erlaubt dann die Rekonstruktion des Klartextes.

Klartext	`010111001`	
Schlüssel	`111010011`	
Chiffre	`101101010`	= Klartext $\oplus$ Schlüssel
Schlüssel	`111010011`	
Klartext	`010111001`	= Chiffre $\oplus$ Schlüssel

Diese Technik funktioniert so lange gut, wie es gelingt, die zum Bearbeiten einer Nachricht verwendeten Schlüssel e und d auf einem sicheren Kanal zu übertragen, z. B. durch einen Kurier. Ein Spion, der ohne Kenntnis der Schlüssel die Leitung anzapft, ist dann nicht in der Lage, den beobachteten Code zu entschlüsseln (immer vorausgesetzt, der Raum der möglichen Schlüssel wurde zur Abwehr eines vollstängigen Durchsuchens gros genug gewählt). Im Zeitalter der globalen Vernetzung besteht natürlich der Wunsch, auch die beiden Schlüsselpaare e und d per Leitung auszutauschen. Nun aber laufen wir Gefahr, das der Spion von ihnen Kenntnis erhält und damit den Code knackt.

Dieses (auf den ersten Blick) unlösbare Problem wurde durch die Einführung von *Public Key Systems* behoben.

Public Key Systems

Gesucht sind zwei Funktionen $enc,\ dec : \mathbb{N} \to \mathbb{N}$ mit folgender Eigenschaft:

1. $dec((enc(x)) = x$
2. effizient zu berechnen
3. aus der Kenntnis von enc läst sich dec nicht effizient bestimmen

Unter Verwendung dieser Funktionen könnte die Kommunikation zwischen den Partner Alice und Bob wie folgt verlaufen:

1. Alice möchte Bob eine Nachricht schicken.
2. Bob veröffentlicht sein enc_B.
3. Alice bildet $y := enc_B(x)$ und schickt es an Bob.
4. Bob bildet $x := dec_B(y)$.

Das RSA-Verfahren

Im Jahre 1978 schlugen Rivest, Shamir, Adleman folgendes Verfahren vor:

geheim:	Wähle zwei grose Primzahlen $p,\ q$ (je 500 Bits)
öffentlich:	Berechne $n := p \cdot q$
geheim:	Wähle d teilerfremd zu $\varphi(n) = (p-1) \cdot (q-1)$
öffentlich:	Bestimme d^{-1}, d.h. e mit $e \cdot d \equiv 1$ mod $\varphi(n)$
öffentlich:	enc(x):= x^e mod n
geheim:	dec(y):= y^d mod n

Beispiel:

$$p = 11,\ q = 13,\ d = 23\ \Rightarrow$$
$$n = 143,\ e = 47$$
$$enc(x) := x^{47} \bmod 143$$
$$dec(y) := y^{23} \bmod 143$$

Korrektheit des RSA-Verfahrens

Die Korrektheit stützt sich auf den **Satz von Fermat/Euler:**

x rel. prim zu $n \Rightarrow x^{\varphi(n)} \equiv 1$ mod n

Effizienz des RSA-Verfahrens

Die Effizienz stützt sich auf folgende Überlegungen:

a) Potenziere mod n

Nicht e-mal mit x malnehmen, denn Aufwand wäre $O(2^{500})$, sondern:

$$x^e := \begin{cases} (x^{e/2})^2 & \text{falls e gerade} \\ (x^{e/2})^2 cdot x & \text{falls e ungerade} \end{cases}$$

Aufwand: $O\ (\log e)$, d.h. proportional zur Anzahl der Dezimalstellen.

b) Bestimme $e := d^{-1}$ Algorithmus von Euklid zur Bestimmung des ggt:

$$ggt(a,b) := \begin{cases} a & \text{falls } b = 0 \\ ggt(b,\ a \bmod b) & \text{sonst} \end{cases}$$

Bestimme $ggt(\varphi(n),\ d)$ und stelle den auftretenden Rest als Linearkombination von $\varphi(n)$ und d dar.

Beispiel:

$$\begin{array}{rcrcl} & & 120 & = & \varphi(n) \\ & & 19 & = & d \\ 120 \bmod 19 & = & 6 & = & \varphi(n) - 6 \cdot d \\ 19 \bmod 6 & = & 1 & = & d - 3 \cdot (\varphi(n) - 6d) = 19d - 3 \cdot \varphi(n) \\ & & & \Rightarrow & e = 19 \end{array}$$

c) Konstruktion einer grosen Primzahl

Wähle 500 Bit lange ungerade Zahl x.

Teste, ob $x,\ x+2,\ x+4,\ x+6, \ldots$ Primzahl ist.

Sei $\Pi(x)$ die Anzahl der Primzahlen unterhalb von x. Es gilt: $\Pi(x) \approx \frac{x}{\ln x} \Rightarrow$ Dichte $\approx \frac{1}{\ln x}\ \Rightarrow$ mittlerer Abstand $\approx \ln x$

Also zeigt sich Erfolg beim Testen ungerader Zahlen der Gröse $n\ =\ 2^{500}$ nach etwa $\frac{\ln 2^{500}}{4}\ =\ 86$ Versuchen.

Komplexitätsklassen für die Erkennung von Primzahlen:

$$\begin{aligned}\text{Prim} &\overset{?}{\in} \mathbb{P}\\ \text{Prim} &\in \mathbb{NP}\\ \overline{\text{Prim}} &\in \mathbb{NP}\\ \overline{\text{Prim}} &\in \mathbb{RP}\end{aligned}$$

$L \in \mathbb{RP} : \Longleftrightarrow$ es gibt Algorithmus A, der angesetzt auf die Frage, ob $x \in L$, nach polynomialer Zeit mit ja oder nein anhält und folgende Gewähr für die Antwort gilt:

$$\begin{aligned}x \notin L \Rightarrow&\ \text{Antwort: nein}\\ x \in L \Rightarrow&\ \text{Antwort: } \underbrace{\text{ja}}_{>1-\varepsilon} \text{ oder } \underbrace{\text{nein}}_{<=\varepsilon}\end{aligned}$$

Antwort: ja $\Rightarrow x$ ist zusammengesetzt.

Antwort: nein $\Rightarrow x$ ist höchstwahrscheinlich prim.

Bei 50 Versuchen $\Rightarrow$ Fehler $\leq \varepsilon^{50}$.

Satz von Rabin:

Sei $n = 2^k \cdot q + 1$ eine Primzahl, $x < n$

1. $x^q \equiv 1$ mod n oder
2. $x^{q \cdot 2^i} \equiv -1$ mod n für ein $i \in \{0,\ \ldots,\ k-1\}$

Beispiel: Sei $n = 97 = 2^5 \cdot 3 + 1$, sei x = 2.

Folge der Potenzen	x	x^3	x^6	x^{12}	x^{24}	x^{48}	x^{96}
Folge der Reste	2	8	64	22	-1	1	1

Definition eines Zeugen: Sei $n = 2^k \cdot q + 1$. Eine Zahl $x < n$ heist Zeuge für die Zusammengesetztheit von n

1. $ggt(x, n)\ \neq 1$ oder
2. $x^q \not\equiv 1$ mod n und $x^{q \cdot 2^i} \not\equiv -1$ für alle $i \in \{0, \ldots, k-1\}$

Satz von Rabin: Ist n zusammengesetzt, so gibt es mindestens $\frac{3}{4}n$ Zeugen.

```
function prob-prim (n: integer): boolean
z:=0;
repeat
   z=z+1;
   wuerfel x;
until (x ist Zeuge fuer n) OR (z=50);
return (z=50)
```

Fehler: $(\frac{1}{4})^{50} \sim 10^{-30}$

Sicherheit des RSA-Verfahrens

Der Code kann nur durch das Faktorisieren von n geknackt werden. Schnellstes Verfahren zum Faktorisieren von n benötigt

$$n^{\sqrt{\frac{\ln\ln(n)}{\ln(n)}}} \text{ Schritte.}$$

Für $n = 2^{1000} \Rightarrow \ln(n) = 690, \ln\ln(n) = 6.5$

Es ergeben sich $\approx \sqrt[10]{n}$ Schritte $\approx 10^{30}$ Schritte $\approx 10^{21}$ sec (bei 10^9 Schritte pro sec) $\approx 10^{13}$ Jahre.

Implementation des RSA-Verfahrens

Abbildung 129 zeigt einen Screenshot von der Ausführung des RSA-Applet[40]

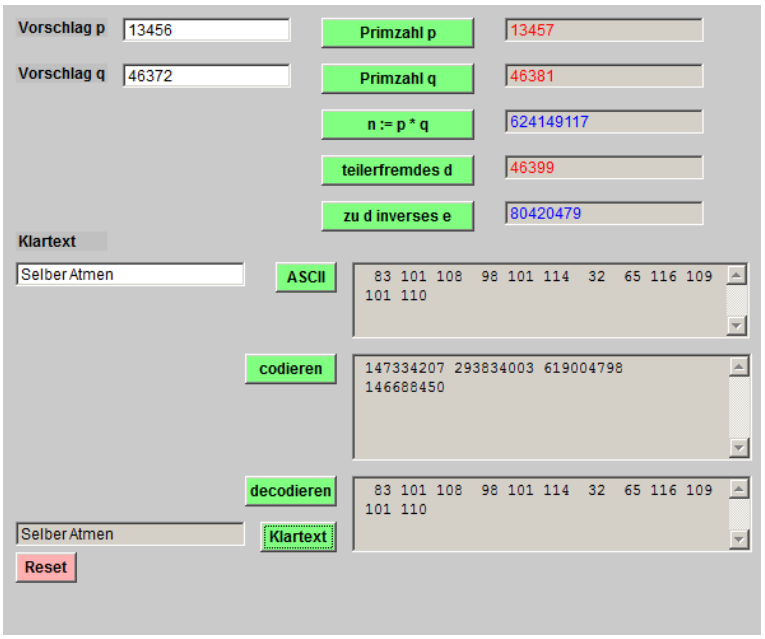


Abb. 129: Screenshot vom RSA Applet

40 http://www-lehre.inf.uos.de/~dbs/2011/rsa/rsa.html

18. Data Warehouse

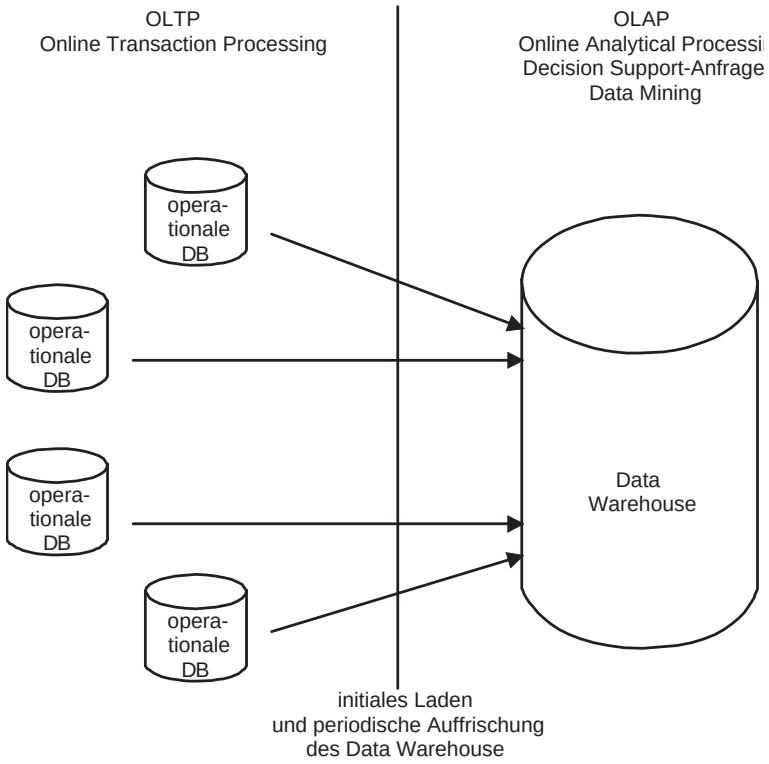


Abb. 130: Zusammenspiel zwischen OLTP und OLAP

Man unterscheidet zwei Arten von Datenbankanwendungen:

- **OLTP** (*Online Transaction Processing*):
  Hierunter fallen Anwendungen wie zum Beispiel das Buchen eines Flugs in einem Flugreservierungssystem oder die Verarbeitung einer Bestellung in einem Handelsunternehmen. OLTP-Anwendungen verarbeiten nur eine begrenzte Datenmenge und operieren auf dem jüngsten, aktuell gültigen Zustand der Datenbasis.
- **OLAP** (*Online Analytical Processing*):
  Eine typische OLAP-Query fragt nach der Auslastung der Transatlantikflüge der letzten zwei Jahre oder nach der Auswirkung gewisser Marketingstrategien. OLAP-Anwendungen verarbeiten sehr grose Datenmengen und greifen auf historische Daten zurück. Sie bilden die Grundlage für *Decision-Support-Systeme*.

OLTP- und OLAP-Anwendungen sollten nicht auf demselben Datenbestand arbeiten aus folgenden Gründen:

- OLTP-Datenbanken sind auf Änderungstransaktionen mit begrenzten Datenmengen hin optimiert.
- OLAP-Auswertungen benötigen Daten aus verschiedenen Datenbanken in konsolidierter, integrierter Form.

Daher bietet sich der Aufbau eines *Data Warehouse* an, in dem die für Decision-Support-Anwendungen notwendigen Daten in konsolidierter Form gesammelt werden. Abbildung 130 zeigt das Zusammenspiel zwischen operationalen Datenbanken und dem Data Warehouse. Typischerweise wird beim Transferieren der Daten aus den operationalen Datenbanken eine Verdichtung durchgeführt, da nun nicht mehr einzelne Transaktionen im Vordergrund stehen, sondern ihre Aggregation.

18.1 Datenbankentwurf für Data Warehouse

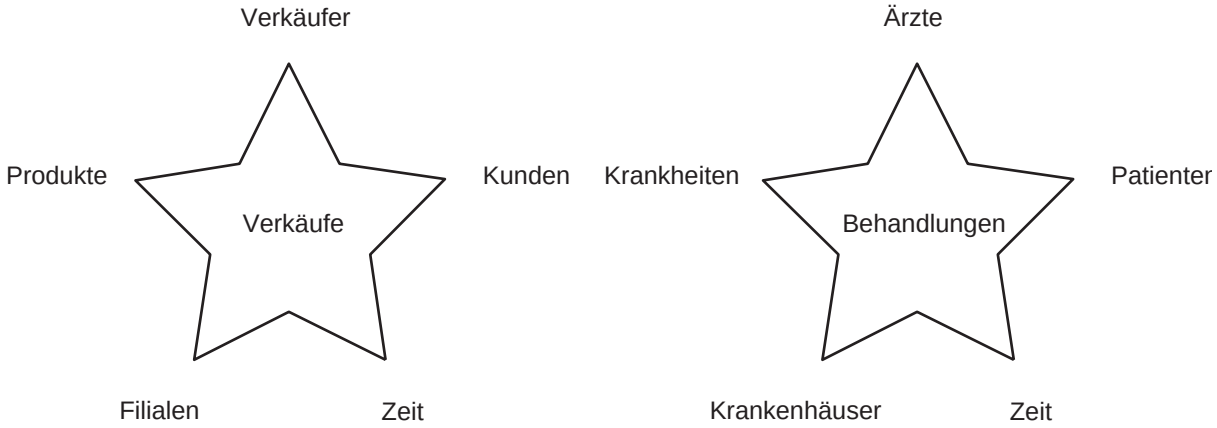


Abb. 131: Sternschemata für Handelsunternehmen und Krankenversicherung

Als Datenbankschema für Data Warehouse-Anwendungen hat sich das sogenannte *Sternschema* (engl.: *star scheme*) durchgesetzt. Dieses Schema besteht aus einer *Faktentabelle* und mehreren *Dimensionstabellen*. 131 zeigt die Sternschemata für zwei Beispielanwendungen in einem Handelsunternehmen und in einer Krankenversicherung.

Verkäufe					
VerkDatum	Filiale	Produkt	Anzahl	Kunde	Verkäufer
30-Jul-96	Passau	1347	1	4711	825
...	...	...	...	...	...

Filialen			
Filialenkennung	Land	Bezirk	...
Passau	D	Bayern	...
...	...	...	...

Kunden			
KundenNr	Name	wiealt	...
4711	Kemper	38	...
...	...	...	...

Verkäufer					
VerkäuferNr	Name	Fachgebiet	Manager	wiealt	...
825	Handyman	Elektronik	119	23	...
...	...	...	...	...	...

Zeit								
Datum	Tag	Monat	Jahr	Quartal	KW	Wochentag	Saison	...
...	...	...	...	...	...	...	...	...
30-Jul-96	30	Juli	1996	3	31	Dienstag	Hochsommer	...
...	...	...	...	...	...	...	...	...
23-Dec-97	27	Dezember	1997	4	52	Dienstag	Weihnachten	...
...	...	...	...	...	...	...	...	...

Produkte					
ProduktNr	Produkttyp	Produktgruppe	Produkthauptgruppe	Hersteller	...
1347	Handy	Mobiltelekom	Telekom	Siemens	...
...	...	...	...	...	...

Abb. 132: Ausprägung des Sternschemas in einem Handelsunternehmen

Bei dem Handelsunternehmen können in der Faktentabelle *Verkäufe* mehrere Millionen Tupel sein, während die Dimensionstabelle *Produkte* vielleicht 10.000 Einträge und die Dimensionstabelle *Zeit* vielleicht 1.000 Einträge (für die letzen drei Jahre) aufweist. 132 zeigt eine mögliche Ausprägung.

Die Dimensionstabellen sind in der Regel nicht normalisiert. Zum Beispiel gelten in der Tabelle *Produkte* folgende funktionale Abhängigkeiten: $ProduktNr \rightarrow Produkttyp$, $Produkttyp \rightarrow Produktgruppe$ und $Produktgruppe \rightarrow Produkthauptgruppe$. In der *Zeit*-Dimension lassen sich alle Attribute aus dem Schlüsselattribut *Datum* ableiten. Trotzdem ist die explizite Speicherung dieser Dimension sinnvoll, da Abfragen nach Verkäufen in bestimmten Quartalen oder an bestimmten Wochentagen dadurch effizienter durchgeführt werden können.

Die Verletzung der Normalformen in den Dimensionstabellen ist bei Decision-Support-Systemen nicht so gravierend, da die Daten nur selten verändert werden und da der durch die Redundanz verursachte erhöhte Speicherbedarf bei den relativ kleinen Dimensionstabellen im Vergleich zu der grosen (normalisierten) Faktentabelle nicht so sehr ins Gewicht fällt.

18.2 Star Join

Das Sternschema führt bei typischen Abfragen zu sogenannten *Star Joins*:

Welche Handys (d.h. von welchen Herstellern) haben junge Kunden in den bayrischen Filialen zu Weihnachten 1996 gekauft ?

```
select sum(v.Anzahl), p.Hersteller
from Verkäufe v, Filialen f, Produkte p, Zeit z, Kunden k
where z.Saison = 'Weihnachten' and z.Jahr = 1996 and k.wiealt < 30
and p.Produkttyp = 'Handy' and f.Bezirk = 'Bayern'
and v.VerkDatum = z.Datum and v.Produkt = p.ProduktNr
and v.Filiale = f.Filialenkennung and v.Kunde = k.KundenNr
group by Hersteller;
```

18.3 Roll-Up/Drill-Down-Anfragen

Der Verdichtungsgrad bei einer SQL-Anfrage wird durch die **group by**-Klausel gesteuert. Werden mehr Attribute in die **group by**-Klausel aufgenommen, spricht man von einem *drill down*. Werden weniger Attribute in die **group by**-Klausel aufgenommen, spricht man von einem *roll up*.

Wieviel Handys wurden von welchem Hersteller in welchem Jahr verkauft ?

```
select p.Hersteller, z.Jahr, sum(v.Anzahl)
from Verkäufe v, Produkte p, Zeit z
where v.Produkt = p.ProduktNr
and v.VerkDatum = z.Datum
and p.Produkttyp = 'Handy'
group by p.Hersteller, z.Jahr;
```

Handyverkäufe nach Hersteller und Jahr		
Hersteller	Jahr	Anzahl
Siemens	1994	2.000
Siemens	1995	3.000
Siemens	1996	3.500
Motorola	1994	1.000
Motorola	1995	1.000
Motorola	1996	1.500
Bosch	1994	500
Bosch	1995	1.000
Bosch	1996	1.500
Nokia	1994	1.000
Nokia	1995	1.500
Nokia	1996	2.000

Handyverkäufe nach Jahr	
Jahr	Anzahl
1994	4.500
1995	6.500
1996	8.500

Handyverkäufe nach Hersteller	
Hersteller	Anzahl
Siemens	8.500
Motorola	3.500
Bosch	3.000
Nokia	4.500

Abb. 133: Analyse der Handy-Verkäufe nach unterschiedlichen Dimensionen

Das Ergebnis wird in der linken Tabelle von 133 gezeigt. In der Tabelle rechts oben bzw. rechts unten finden sich zwei Verdichtungen.

Durch das Weglassen der Herstellerangabe aus der **group by**-Klausel (und der **select**-Klausel) entsteht ein *roll up* entlang der Dimension *p.Hersteller*:

Wieviel Handys wurden in welchem Jahr verkauft ?

```
select z.Jahr, sum(v.Anzahl)
from Verkäufe v, Produkte p, Zeit z
where v.Produkt = p.ProduktNr
and v.VerkDatum = z.Datum
and p.Produkttyp = 'Handy'
group by z.Jahr;
```

Durch das Weglassen der Zeitangabe aus der **group by**-Klausel (und der **select**-Klausel) entsteht ein *roll up* entlang der Dimension *z.Jahr*:

Wieviel Handys wurden von welchem Hersteller verkauft ?

```
select p.Hersteller, sum(v.Anzahl)
from Verkäufe v, Produkte p
where v.Produkt = p.ProduktNr and v.VerkDatum = z.Datum
and p.Produkttyp = 'Handy'
group by p.Hersteller;
```

Die ultimative Verdichtung besteht im kompletten Weglassen der **group-by**-Klausel. Das Ergebnis besteht aus einem Wert, nämlich 19.500:

Wieviel Handys wurden verkauft ?

```
select sum(v.Anzahl)
from Verkäufe v, Produkte p
where v.Produkt = p.ProduktNr
and p.Produkttyp = 'Handy';
```

Hersteller \ Jahr	1994	1995	1996	Σ
Siemens	2.000	3.000	3.500	8.500
Motorola	1.000	1.000	1.500	3.500
Bosch	500	1.000	1.500	3.000
Nokia	1.000	1.500	2.000	4.500
Σ	4.500	6.500	8.500	19.500

Abb. 134: Handy-Verkäufe nach Jahr und Hersteller

Durch eine sogenannte *cross tabulation* können die Ergebnisse in einem n-dimensionalen Spreadsheet zusammengefast werden. 134 zeigt die Ergebisse aller drei Abfragen zu 133 in einem 2-dimensionalen Datenwürfel *data cube*.

18.4 Materialisierung von Aggregaten

Da es sehr zeitaufwendig ist, die Aggregation jedesmal neu zu berechnen, empfiehlt es sich, sie zu materialisieren, d.h. die vorberechneten Aggregate verschiedener Detaillierungsgrade in einer Relation abzulegen. Es folgen einige SQL-Statements, welche die linke Tabelle von 135 erzeugen. Mit dem **null**-Wert wird markiert, das entlang dieser Dimension die Werte aggregiert wurden.

```
create table Handy2DCube (Hersteller varchar(20),
                          Jahr integer,
                          Anzahl integer);

insert into Handy2DCube
(select p.Hersteller, z.Jahr, sum(v.Anzahl)
from Verkäufe v, Produkte p, Zeit z
where v.Produkt = p.ProduktNr and p.Produkttyp = 'Handy'
and v.VerkDatum = z.Datum
group by z.Jahr, p.Hersteller)
union
(select p.Hersteller, null, sum(v.Anzahl)
from Verkäufe v, Produkte p
where v.Produkt = p.ProduktNr and p.Produkttyp = 'Handy'
group by p.Hersteller)
union
(select null, z.Jahr, sum(v.Anzahl)
from Verkäufe v, Produkte p, Zeit z
where v.Produkt = p.ProduktNr and p.Produkttyp = 'Handy'
and v.VerkDatum = z.Datum
group by z.Jahr)
union
(select null, null, sum(v.Anzahl)
from Verkäufe v, Produkte p
where v.Produkt = p.ProduktNr and p.Produkttyp = 'Handy');
```

Handy2DCube		
Hersteller	Jahr	Anzahl
Siemens	1994	2.000
Siemens	1995	3.000
Siemens	1996	3.500
Motorola	1994	1.000
Motorola	1995	1.000
Motorola	1996	1.500
Bosch	1994	500
Bosch	1995	1.000
Bosch	1996	1.500
Nokia	1994	1.000
Nokia	1995	1.500
Nokia	1996	2.000
null	1994	4.500
null	1995	6.500
null	1996	8.500
Siemens	**null**	8.500
Motorola	**null**	3.500
Bosch	**null**	3.000
Nokai	**null**	4.500
null	**null**	19.500

Handy3DCube			
Hersteller	Jahr	Land	Anzahl
Siemens	1994	D	800
Siemens	1994	A	600
Siemens	1994	CH	600
Siemens	1995	D	1.200
Siemens	1995	A	800
Siemens	1995	CH	1.000
Siemens	1996	D	1.400
...	...	...	...
Motorola	1994	D	400
Motorola	1994	A	300
Motorola	1994	CH	300
...	...	...	...
Bosch	...	...	...
...	...	...	...
null	1994	D	...
null	1995	D	...
...	...	...	...
Siemens	**null**	**null**	8.500
...	...	...	...
null	**null**	**null**	19.500

Abb. 135: Materialisierung von Aggregaten in einer Relation

Offenbar ist es recht mühsam, diese Art von Anfragen zu formulieren, da bei n Dimensionen insgesamt 2^n Unteranfragen formuliert und mit **union** verbunden werden müssen. Auserdem sind solche Anfragen extrem zeitaufwendig auszuwerten, da jede Aggregation individuell berechnet wird, obwohl man viele Aggregate aus anderen (noch nicht so stark verdichteten) Aggregaten berechnen könnte.

18.5 Der Cube-Operator

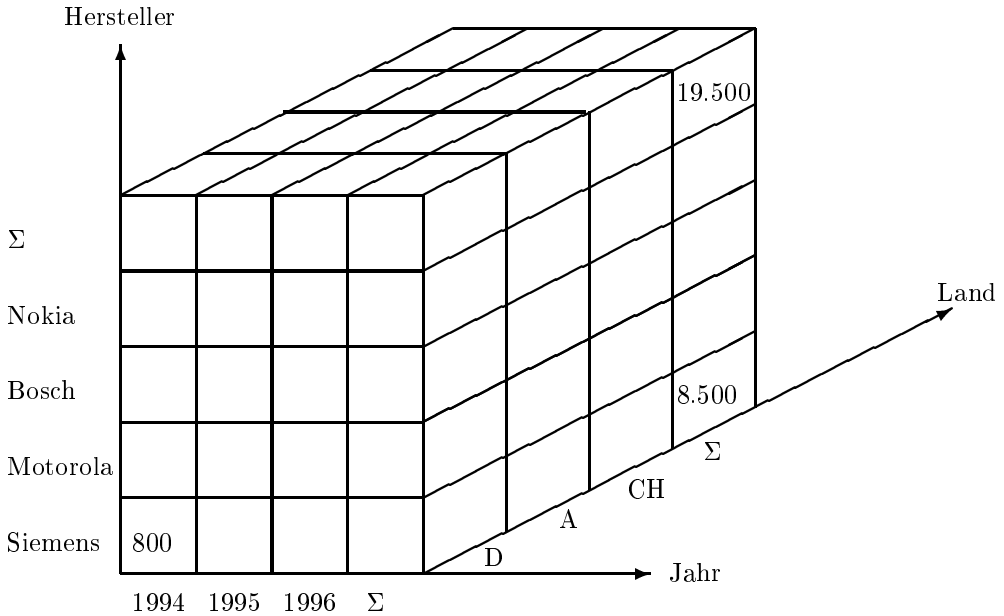


Abb. 136: Würfeldarstellung der Handyverkaufszahlen nach Jahr, Hersteller und Land

Um der mühsamen Anfrageformulierung und der ineffizienten Auswertung zu begegnen, wurde vor kurzem ein neuer SQL-Operator namens **cube** vorgeschlagen. Zur Erläuterung wollen wir ein 3-dimensionales Beispiel konstruieren, indem wir auch entlang der zusätzlichen Dimension *Filiale.Land* ein *drill down* vorsehen:

```
select p.Hersteller, z.Jahr, f.Land, sum(Anzahl)
from Verkäufe v, Produkte p, Zeit z, Filialen f
where v.Produkt = p.ProduktNr
and p.Produkttpy = 'Handy'
and v.VerkDatum = z.Datum
and v.Filiale = f.Filialenkennung
group by z.Jahr, p.Hersteller, f.Land with cube;
```

Die Auswertung dieser Query führt zu dem in 136 gezeigten 3D-Quader; die relationale Repräsentation ist in der rechten Tabelle von [?] zu sehen. Neben der einfacheren Formulierung erlaubt der Cube-Operator dem DBMS einen Ansatz zur Optimierung, indem stärker verdichtete Aggregate auf weniger starken aufbauen und indem die (sehr grose) *Verkäufe*-Relation nur einmal eingelesen werden mus.

18.6 Data Warehouse-Architekturen

Es gibt zwei konkurrierende Architekturen für Data Warehouse Systeme:

- **ROLAP**: Das Data Warehouse wird auf der Basis eines relationalen Datenmodells realisiert (wie in diesem Kapitel geschehen).
- **MOLAP**: Das Data Warehouse wird auf der Basis masgeschneiderter Datenstrukturen realisiert. Das heist, die Daten werden nicht als Tupel in Tabellen gehalten, sondern als Einträge in mehrdimensionalen Arrays. Probleme bereiten dabei dünn besetzte Dimensionen.

18.7 Data Mining

Beim *Data Mining* geht es darum, grose Datenmengen nach (bisher unbekannten) Zusammenhängen zu durchsuchen. Man unterscheidet zwei Zielsetzungen bei der Auswertung der Suche:

- Klassifikation von Objekten,
- Finden von Assoziationsregeln.

Bei der Klassifikation von Objekten (z. B: Menschen, Aktienkursen, etc.) geht es darum, Vorhersagen über das zukünftige Verhalten auf Basis bekannter Attributwerte zu machen. Tabelle 17.1 zeigt eine Ausschnitt aus der Liste der Schadensmeldungen einer KFZ-Versicherung. Durch sukzessives Wählen geeigneter Attribute entsteht ein Entscheidungsbaum (siehe 137). Für die Risikoabschätzung könnte man beispielsweise vermuten, das Männer zwischen 35 und 50 Jahren, die ein Coupé fahren, in eine hohe Risikogruppe gehören. Diese Klassifikation wird dann anhand einer repräsentativen Datenmenge verifiziert. Die Wahl der Attribute für die Klassifikation erfolgt benutzergesteuert oder auch automatisch durch ``Ausprobieren``.

Alter	Geschlecht	Autotyp	Schaden
45	w	Van	gering
18	w	Coupé	gering
22	w	Van	gering
19	m	Coupé	hoch
38	w	Coupé	gering
24	m	Van	Gering
40	m	Coupé	hoch
40	m	Van	gering

Tabelle 17.1 Beobachtete Schadensfälle

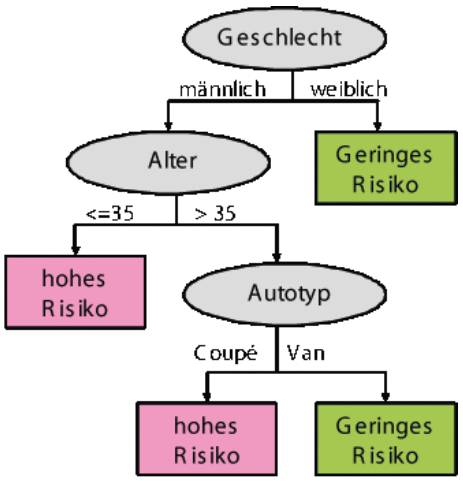


Abb. 137: Entscheidungsbaum

Bei der Suche nach Assoziativregeln geht es darum, Zusammenhänge bestimmter Objekte durch Implikationsregeln auszudrücken, die vom Benutzer vorgeschlagen oder vom System generiert werden. Zum Beispiel könnte eine Regel beim Kaufverhalten von Kunden folgende (informelle) Struktur haben:

```
Wenn jemand einen PC kauft
dann kauft er auch einen Drucker.
```

Bei der Verifizierung solcher Regeln wird keine 100 %-ige Einhaltung erwartet. Stattdessen geht es um zwei Kenngrösen:

- **Confidence:**
  Dieser Wert legt fest, bei welchem Prozentsatz der Datenmenge, bei der die Voraussetzung (linke Seite) erfüllt ist, die Regel (rechte Seite) auch erfüllt ist. Eine *Confidence* von 80% sagt aus, das vier Fünftel der Leute, die einen PC gekauft haben, auch einen Drucker dazu genommen haben.
- **Support:**
  Dieser Wert legt fest, wieviel Datensätze überhaupt gefunden wurden, um die Gültigkeit der Regel zu verifizieren. Bei einem Support von 1% wäre also jeder Hundertste Verkauf ein PC zusammen mit einem Drucker.

TransID	Produkt
111	Drucker
111	Papier
111	PC
111	Toner
222	PC
222	Scanner
333	Drucker
333	Papier
333	Toner
444	Drucker
444	PC
555	Drucker
555	Papier
555	PC
555	Scanner
555	Toner

Frequent Itemset-Kandidat	Anzahl
{Drucker}	4
{Papier}	3
{PC}	4
{Toner}	2
{Toner}	3
{Drucker, Papier}	3
{Drucker, PC}	3
{Drucker, Scanner}	
{Drucker, Toner}	3
{Papier, PC}	2
{Papier, Scanner}	
{Papier, Toner}	3
{PC, Scanner}	
{PC, Toner}	2
{Scanner, Toner}	

{Drucker, Papier, PC}	
{Drucker, Papier, Toner}	3
{Drucker, PC, Toner}	
{Papier, PC, Toner}	

Tabelle 17.2 Verkaufstransaktionen und Zwischenergebnisse des A-Priori-Algorithmus

Zur Ermittlung der Assoziationsregeln verwendet man den A-Priori-Algorithmus, welcher sogenannte *frequent itemsets* berechnet, also Produktgruppen, die häufig gemeinsam gekauft wurden. Tabelle 17.2 zeigt den Verlauf des Algorithmus, der aus den beobachteten Verkäufen sukzessive alle Frequent Itemsets mit mindestens 3 Items ermittelt. Aus der TransaktionsID lässt sich zunächst ermitteln, welche Produkte gemeinsam gekauft wurden. Danach werden die Frequent Itemsets der Mächtigkeit k erweitert zu Frequent Itemsets der Mächtigkeit $k+1$. Zum Schluss bleibt die Kombination {Drucker, Papier, Toner } als einzige Dreier-Kombination übrig.

Sei F ein Frequent Itemset. Dann gilt

$$support(F) := \frac{Anzahl\ des\ Vorkommens}{Gesamtzahl}$$

.

Wir betrachten alle disjunkten Zerlegungen von F in L und R.

Die Regel $L \Rightarrow R$ hat dann folgende Confidence

$$confidence(L \Rightarrow R) = \frac{support(F)}{support(R)}$$

Beispiel: Die Regel {Drucker} $\Rightarrow$ {Papier, Toner} hat

$$confidence = \frac{support(\{Drucker, Papier, Toner\})}{support(\{Drucker\})} = \frac{3/5}{4/5} = 0.75$$

Also haben 75 % der Kunden, die einen Drucker gekauft haben, auch Papier und Toner gekauft.