Datenbanksysteme

Vorlesung im SS '99

[Version vom 22. Juli 1999]

Oliver Vornberger

Praktische Informatik
Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

- Date, C.J:
  An Introduction to Database Systems,
  Addison-Wesley, 6. Auflage 1995.

- Elmasri R. & S. Navathe:
  Fundamentals of Database Systems ,
  Benjamin/Cummings Publishing Company, 2. Auflage 1994 (3. Auflage erscheint Herbst 99).

- Hamilton G., R. Cattell, M. Fisher:
  JDBC. Datenbankzugriff mit Java ,
  Addison-Wesley, Bonn, 1998

- Heuer, A. & G. Saake:
  Datenbanken - Konzepte und Sprachen ,
  International Thompson Publishing, 1. korrigierter Nachdruck 1997.

- Kemper, A. & A. Eickler:
  Datenbanksysteme - Eine Einführung
  Oldenbourg, 2. aktualisierte Auflage 1997.

- Schlager, G. & W. Stucky:
  Datenbanksysteme: Konzepte und Modelle
  Teubner Studienbuch Informatik, 2. Auflage 1983

- Silberschatz, A. & H.F. Korth & S. Sudarshan:
  Database System Concepts,
  Mc Graw-Hill, 1991.

- Ullman, J. D.:
  Principles of Data and Knowledge-Base Systems,
  Computer Science Press, 1988.

- Vossen, G.:
  Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme,
  Addison-Wesley, 2.aktualisierte Ausgabe 1994.

Die Vorlesung orientiert sich überwiegend an dem Buch von Kemper/Eickler. Zahlreiche Beispiele und Grafiken wurden von dort übernommen. 10 Exemplare sind in der Lehrbuchsammlung vorhanden (Standort: N-LB, Signatur: TWY/Kem).

HTML-Version
Der Inhalt dieser Vorlesung und die dazu gehörenden Übungsaufgaben können online abgerufen werden unter `http://www-lehre.informatik.uni-osnabrueck.de/~dbs`

Danksagung

Ich danke ...

... Frau Astrid Heinze für sorgfältiges Erfassen zahlreicher Texte, Grafiken und Tabellen.

... den Studenten Ralf Kunze, Stefan Rauch und Benjamin Stark für Installation und Erprobung diverser Software-Beispiele.

... Herrn Viktor Herzog für die Konvertierung des Skripts nach HTML.

Osnabrück, im Juli 1999

Oliver Vornberger

(Oliver Vornberger)

Inhaltsverzeichnis

Kapitel 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 1.1).

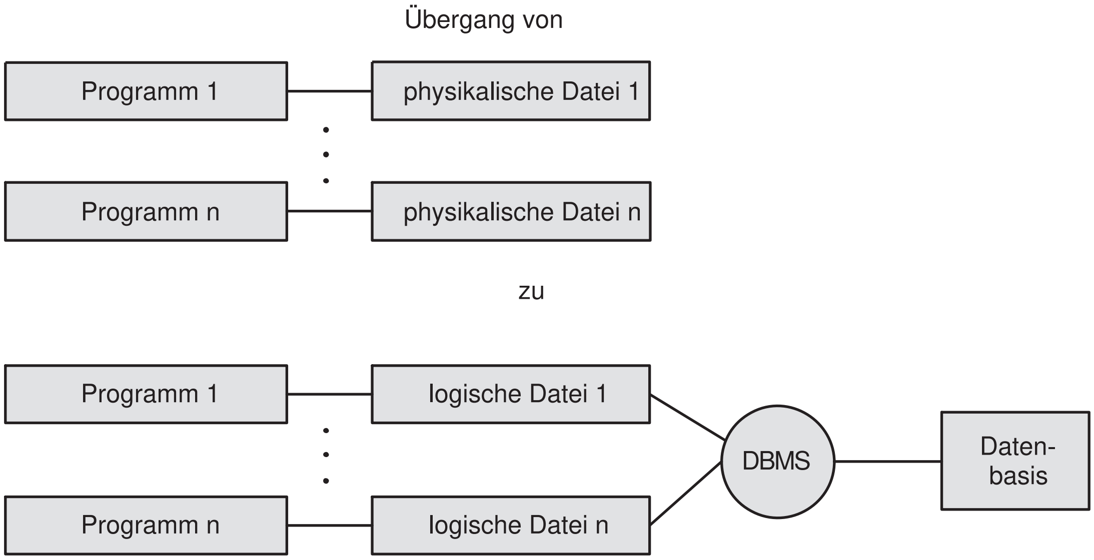


Abbildung 1.1: Isolierte Dateien versus zentrale Datenbasis

1.3 Datenabstraktion

Man unterscheidet drei Abstraktionsebenen im Datenbanksystem (Abbildung 1.2):

- **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.

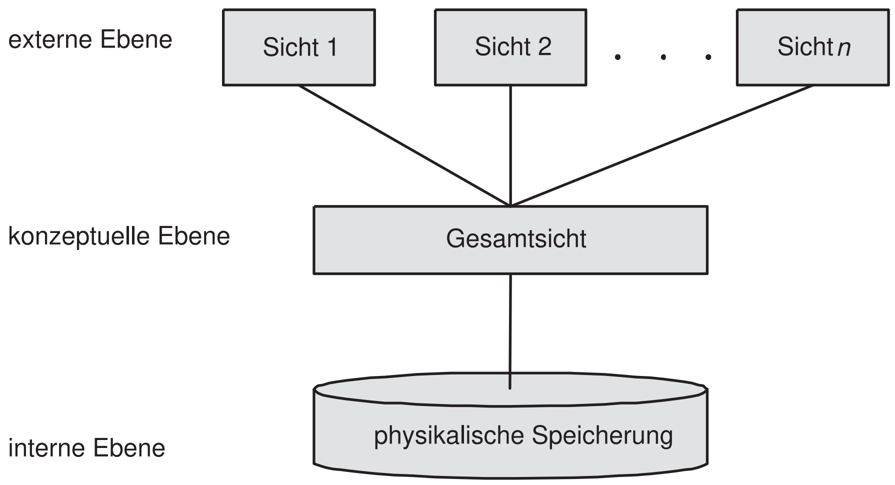


Abbildung 1.2: Drei Abstraktionsebenen eines Datenbanksystems

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* 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

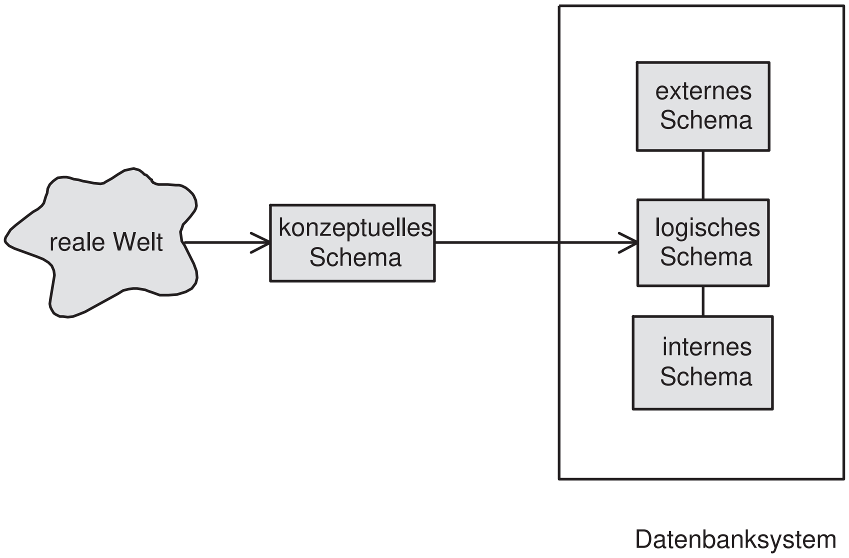


Abbildung 1.3: 2-stufige Modellierung

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 (Abbildung 1.3).

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. wohnt 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, dem Entity-Typ, zusammengefaßt werden. Entsprechend entstehen Relationship-Typen.

- **Beispiel:**
  Entity-Typ `Student` habe die Attribute `Mat-Nr.`, `Name`, `Hauptfach`.
  Entity-Typ `Ort` habe die Attribute `PLZ`, `Name`.
  Relationship-Typ `wohnt` setzt `Student` und `Ort` 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 1.4 zeigt ein Beispiel.

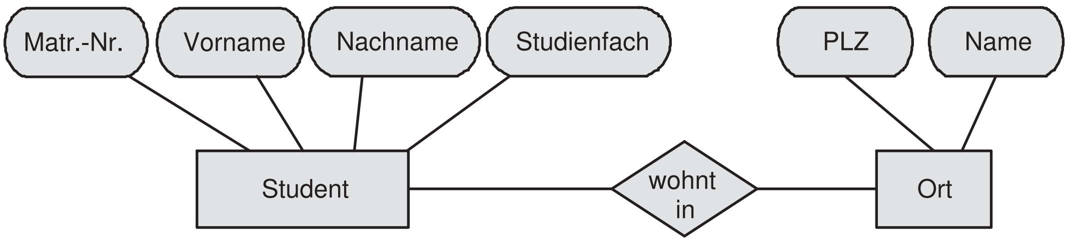


Abbildung 1.4: Beispiel für E-R-Diagramm

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 1.5 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.

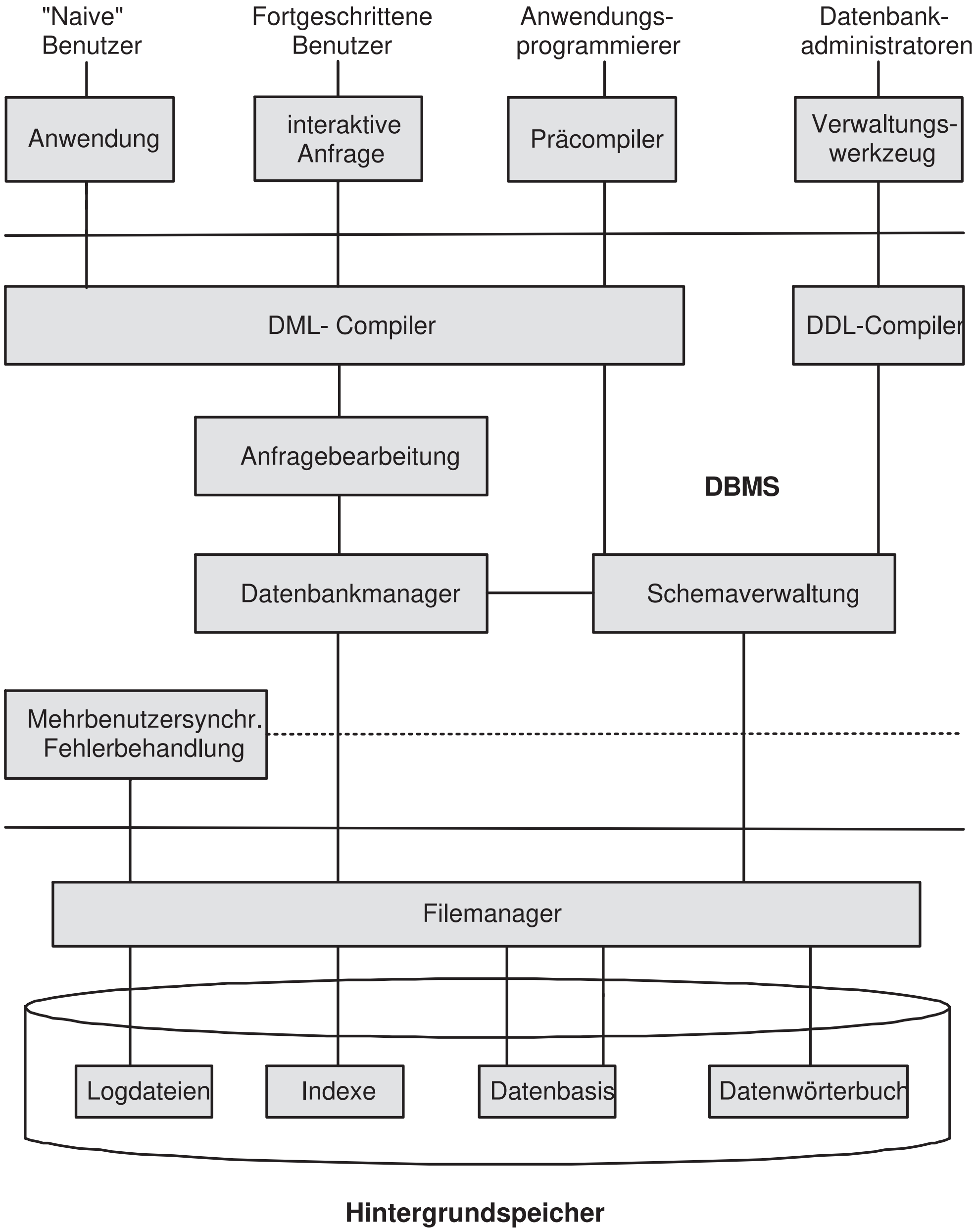


Abbildung 1.5: Architektur eines DBMS

Kapitel 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.

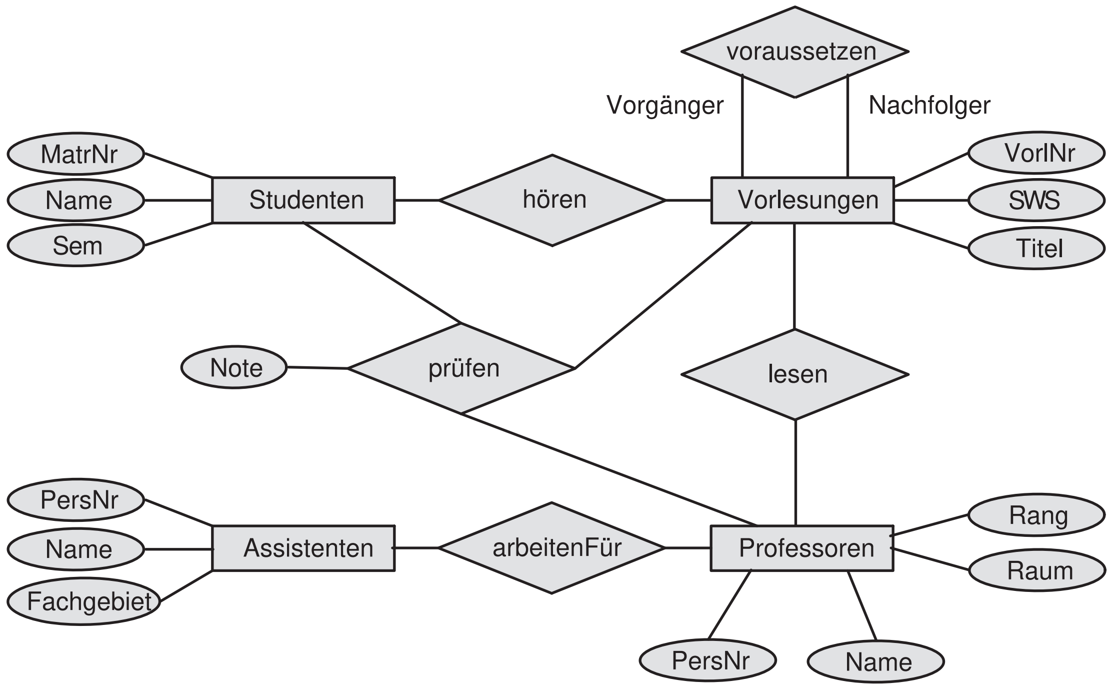


Abbildung 2.1: ER-Diagramm für Universität

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 2.1 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.* 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 2.2). 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*.

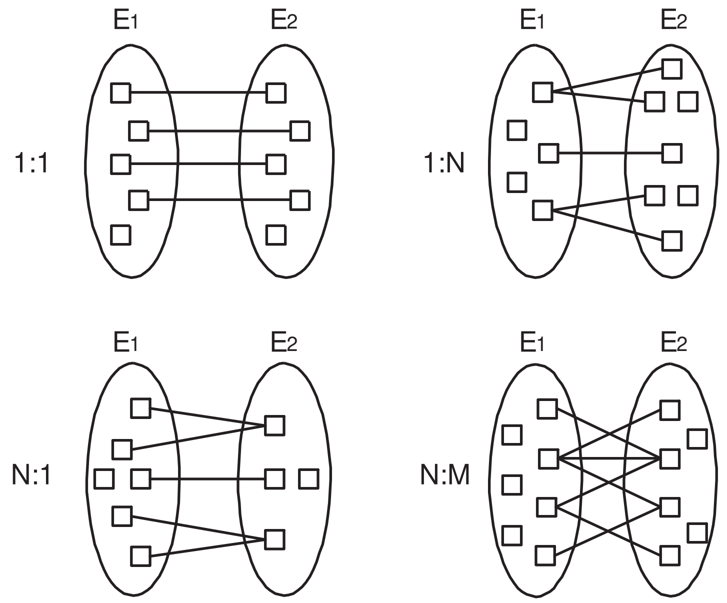


Abbildung 2.2: Mögliche Funktionalitäten von binären Beziehungen

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.

$$\text{beschäftigt\_bei} : \text{Personen} \rightarrow \text{Firmen}$$

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 2.3 zeigt die Verwendung der *(min, max)*-Notation anhand der Begrenzungsflächenmodellierung von Polyedern.

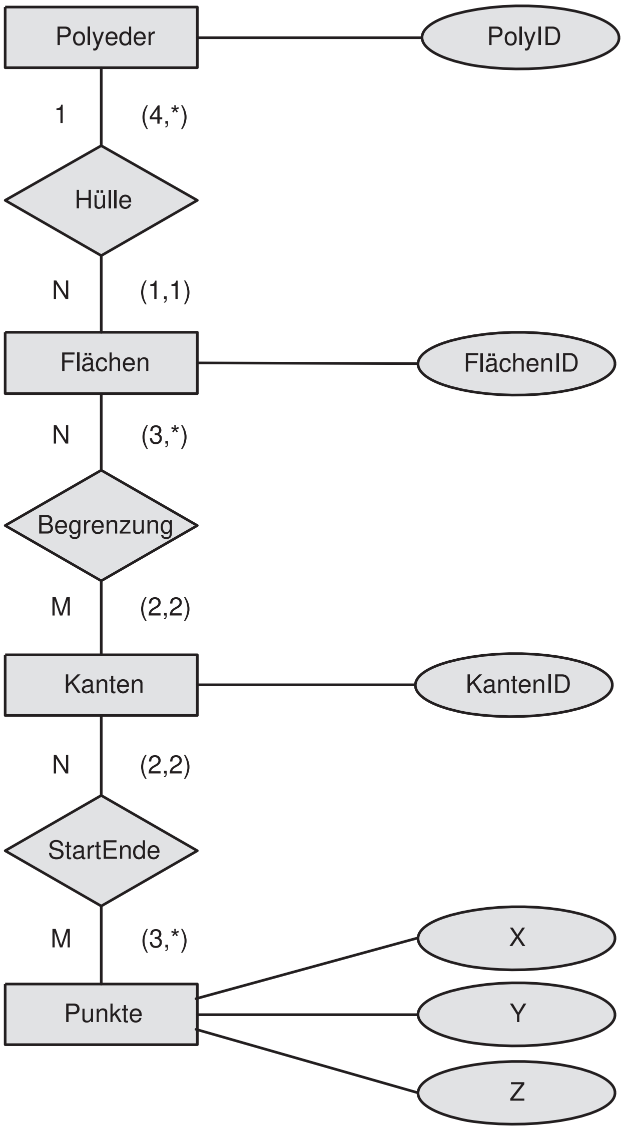


Abbildung 2.3: ER-Diagramm für Begrenzungsflächendarstellung von Polyedern

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 2.4 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.

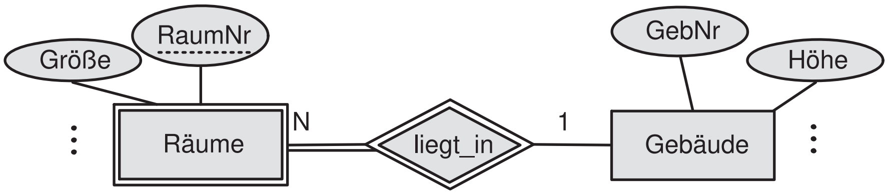


Abbildung 2.4: Ein existenzabhängiger (schwacher) Entity-Typ

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

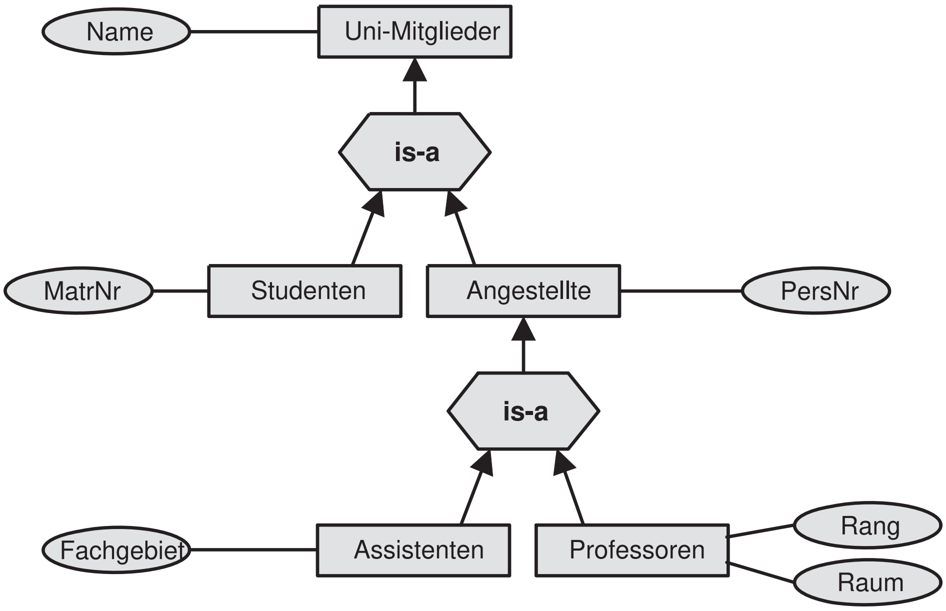


Abbildung 2.5: Spezialisierung der Universitätsmitglieder

Bezüglich der Teilmengensicht ist von Interesse:

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

In Abbildung 2.5 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 2.6 zeigt die Aggregationshierarchie eines Fahrrads. Zum Beispiel sind *Rohre* und *Lenker* Bestandteile des *Rahmen*; *Felgen* und *Speichen* sind Bestandteile der *Räder*.

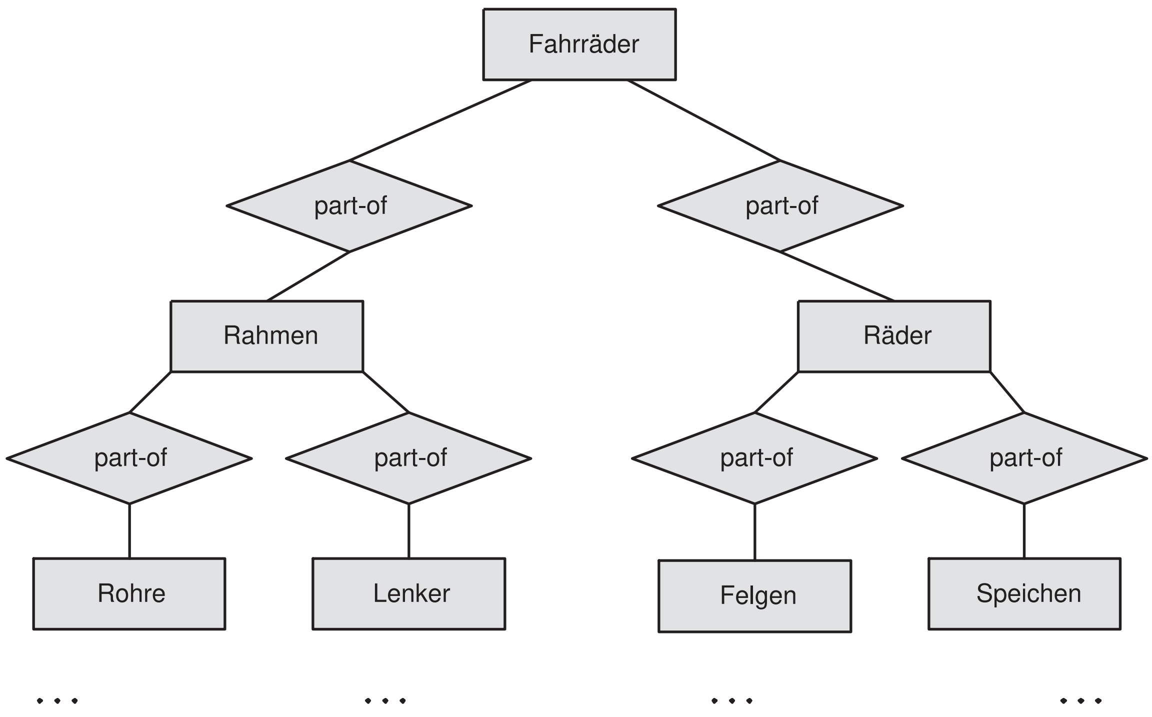


Abbildung 2.6: Aggregationshierarchie eines Fahrrads

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 2.7).

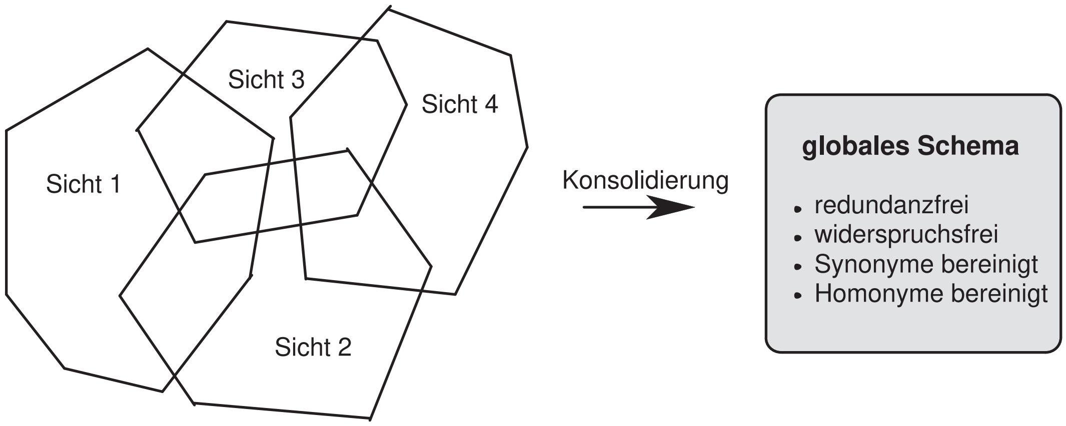


Abbildung 2.7: Konsolidierung überlappender Sichten

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.

Abbildung 2.8 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 2.9 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.

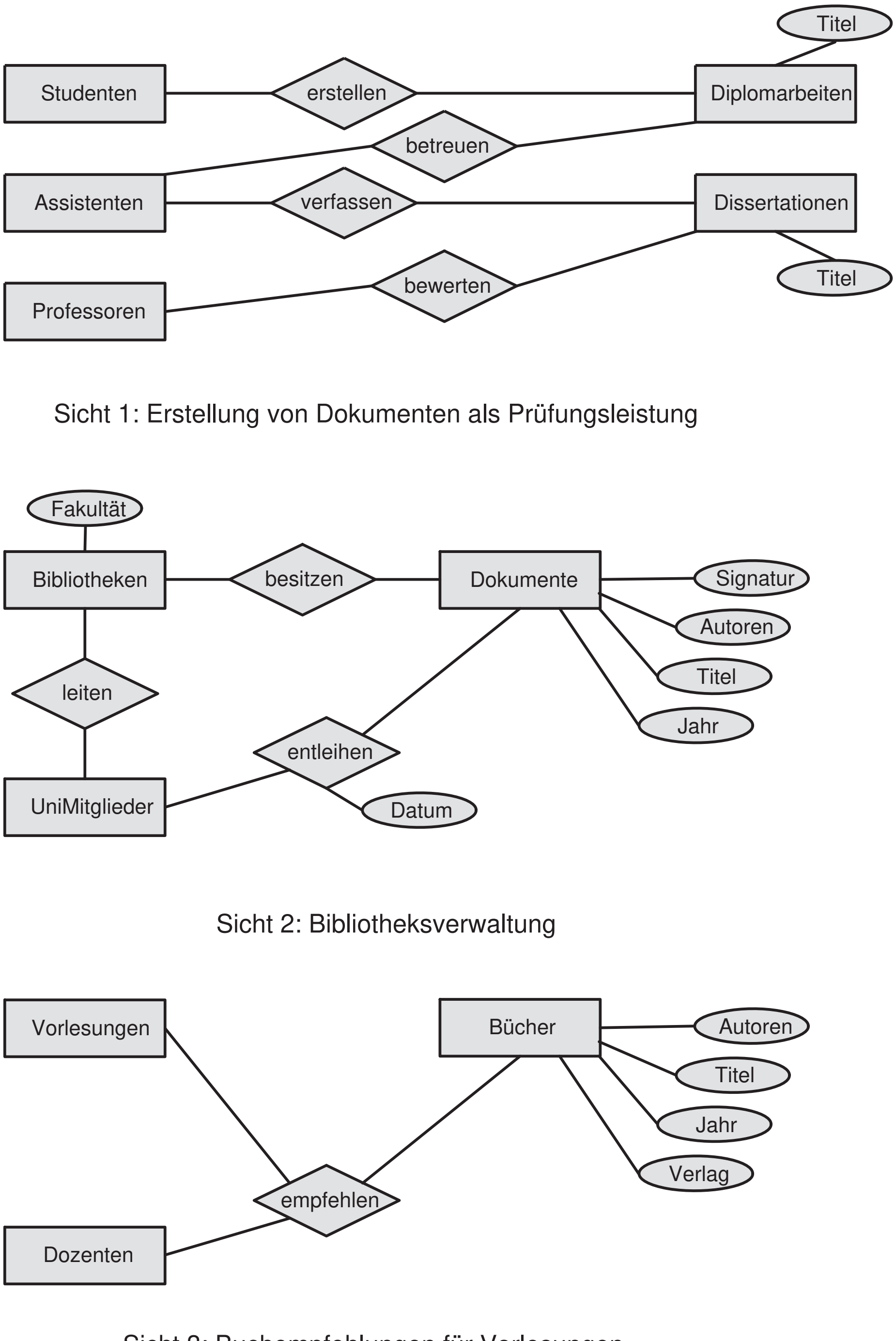


Abbildung 2.8: Drei Sichten einer Universitätsdatenbank

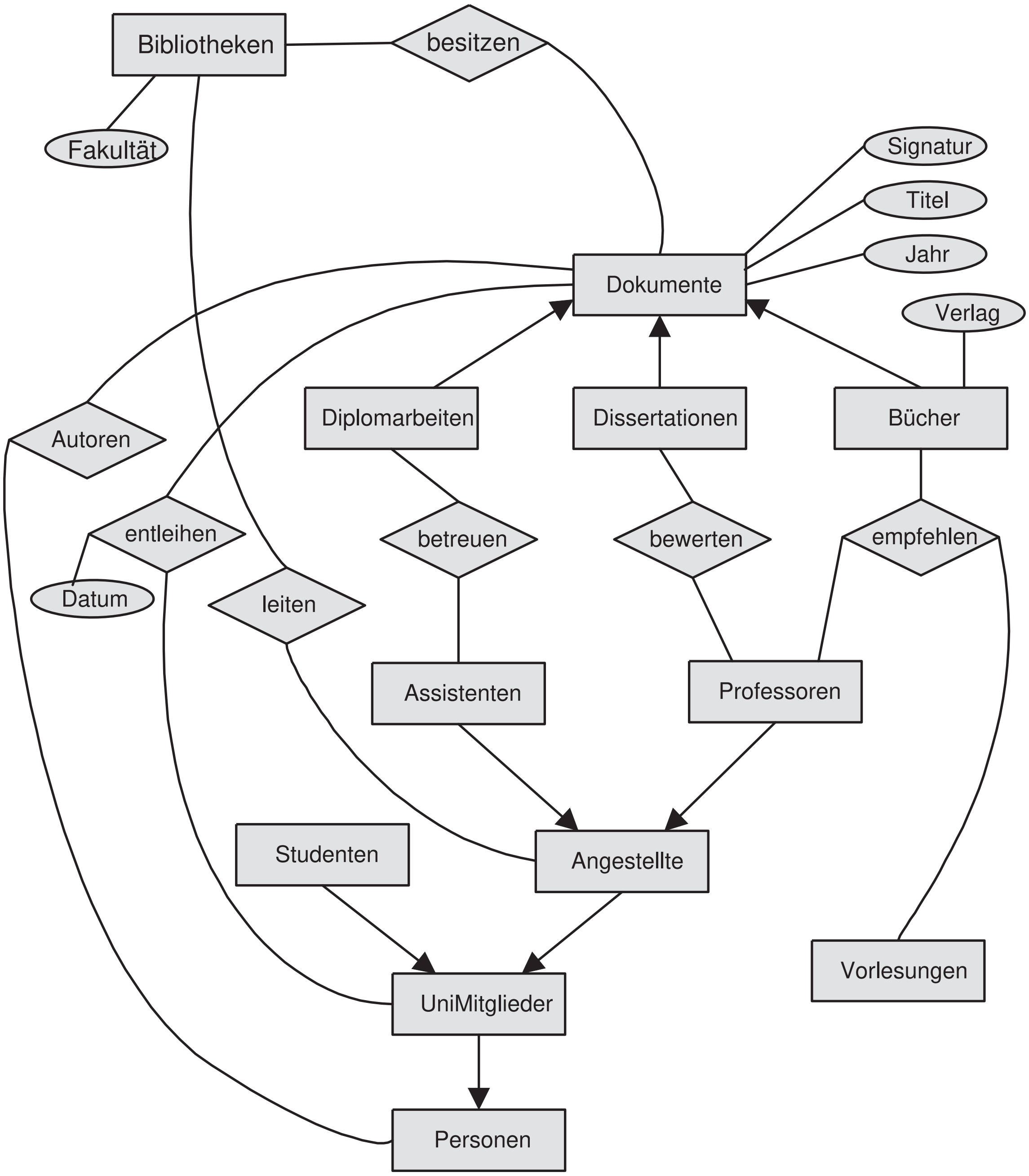


Abbildung 2.9: Konsolidiertes Schema der Universitätsdatenbank

Kapitel 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 3.1 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.

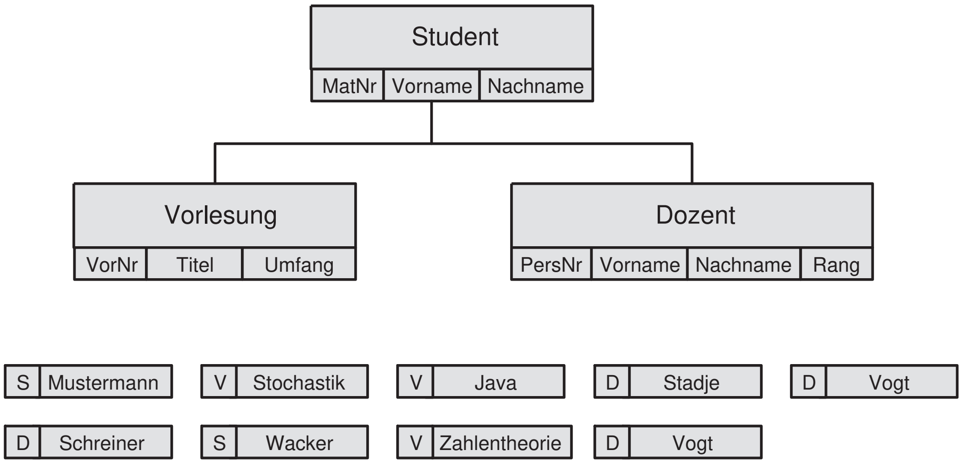


Abbildung 3.1: Hierarchisches Schema und eine Ausprägung.
Der Recordtyp sei erkennbar an den Zeichen S (Studenten), V (Vorlesungen) und D (Dozenten)

Die typische Operation besteht aus dem Traversieren der Hierarchie unter Berücksichtigung der jeweiligen Vaterschaft, d. h. der Befehl

```
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.

Beispiel-Query: Welche Hörer sitzen in der Vorlesung Zahlentheorie ?

1.) Einstieg in Baum mit Wurzel *Studenten*
2.) suche nächsten Student
3.) suche unter seinen Söhnen eine Vorlesung mit Titel `Zahlentheorie`
4.) falls gefunden: gib den Studenten aus
5.) gehe nach 2.)

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.

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 3.2).

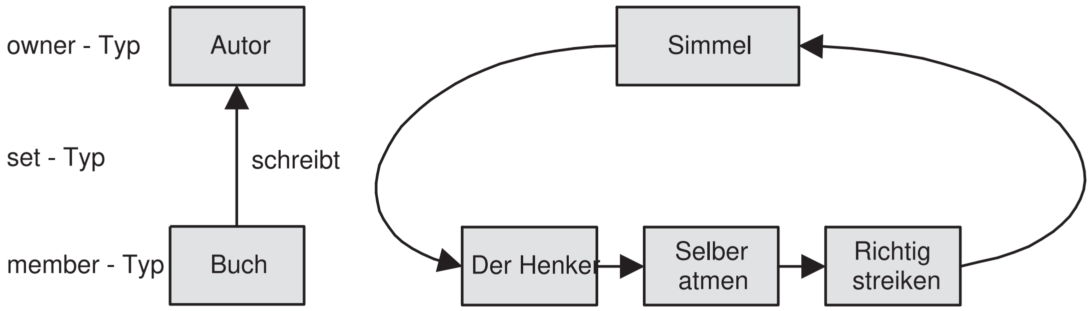


Abbildung 3.2: Netzwerkschema und eine Ausprägung

Bei nicht binären Beziehungen oder nicht many-one-Beziehungen hilft man sich durch Einführung von künstlichen *Kett-Records*. Abbildung 3.3 zeigt ein entsprechendes Netzwerkschema und eine Ausprägung, bei der zwei Studenten jeweils zwei Vorlesungen hören.

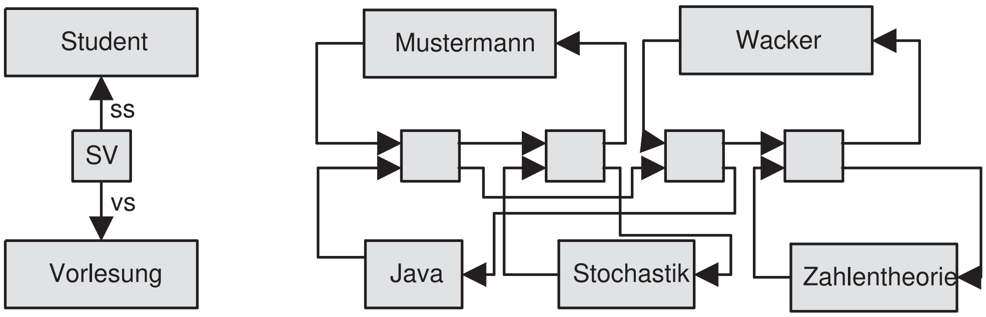


Abbildung 3.3: Netzwerkschema mit Kett-Record und eine Ausprägung

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.

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 3.4 zeigt ein Schema zum Vorlesungsbetrieb und eine Ausprägung.

Student

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

Hoert

MatNr	VorNr
653467	6.712
875462	6.712
432788	6.712
875462	6.102

Vorlesung

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

Abbildung 3.4: 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.Nachname from Student, Hoert, Vorlesung
WHERE  Student.MatNr   = Hoert.MatNr
AND    Hoert.VorNr     = Vorlesung.VorNr
AND    Vorlesung.Titel = "Zahlentheorie"
```

3.4 Das Objektorientierte Datenmodell

Eine Klasse repräsentiert einen Entity-Typ zusammen mit darauf erlaubten Operationen. Attribute müssen nicht atomar sein, sondern bestehen ggf. aus Tupeln, Listen und Mengen. Die Struktur einer Klasse kann an eine Unterklasse vererbt werden. Binäre Beziehungen können durch mengenwertige Attribute modelliert werden.

Die Definition des Entity-Typen *Person* mit seiner Spezialisierung *Student* incl. der Beziehung *hoert* sieht im objektorientierten Datenbanksystem O_2 wie folgt aus:

```
class Person
  type tuple (name        : String,
              geb_datum   : Date,
              kinder      : list(Person))
end;

class Student inherit Person
  type tuple (mat_nr      : Integer,
              hoert       : set (Vorlesung))
end;

class Vorlesung
  type tuple (titel       : String,
              gehoert_von : set (Student))
end;
```

Kapitel 4

Physikalische Datenorganisation

4.1 Grundlagen

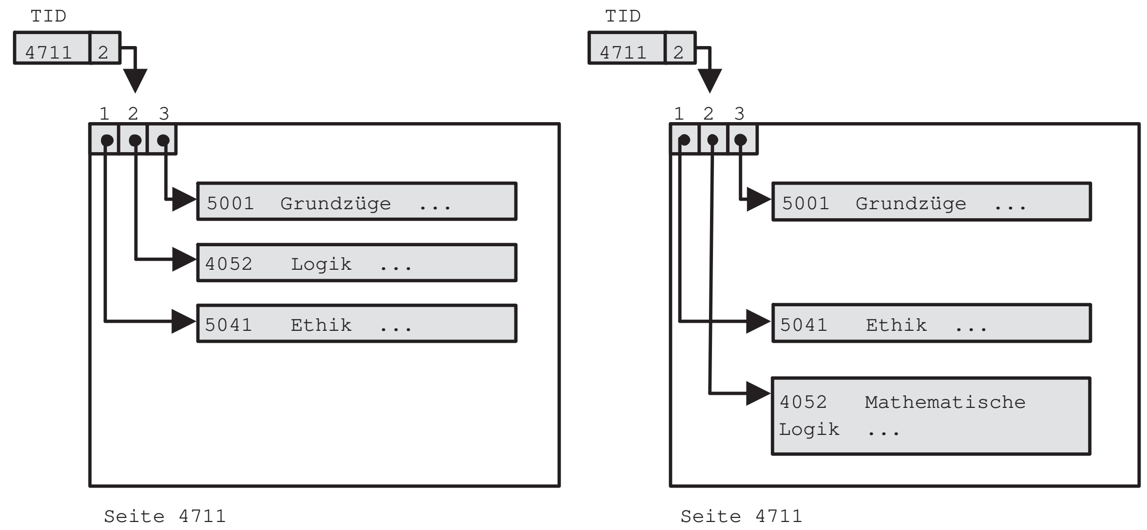


Abbildung 4.1: Verschieben eines Tupels innerhalb einer Seite

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.

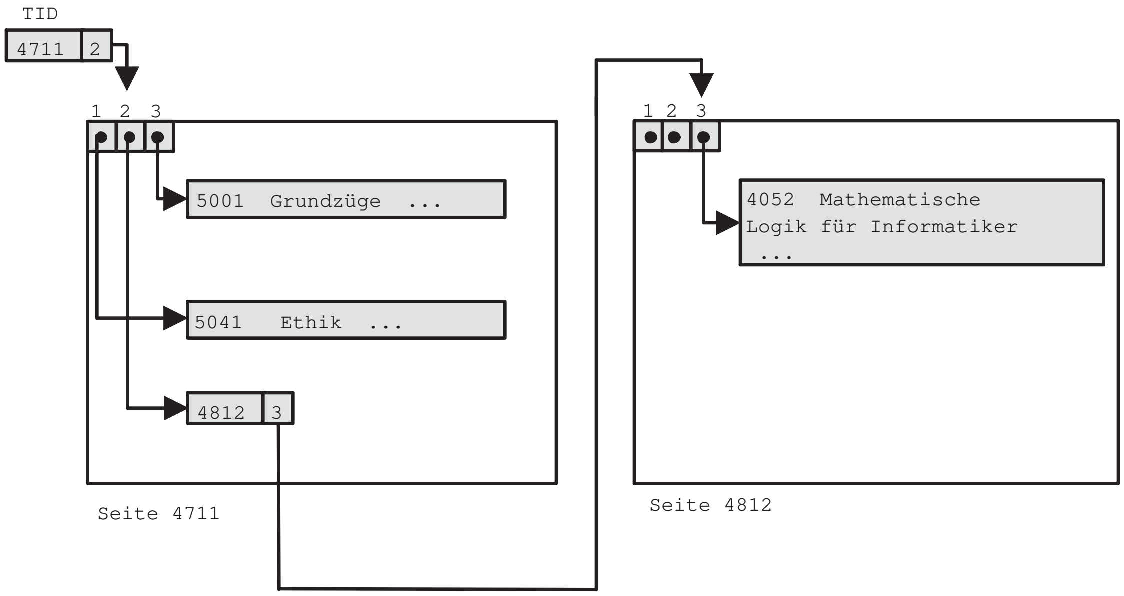


Abbildung 4.2: Verdrängen eines Tupels von einer Seite

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*).

Abbildung 4.1 zeigt das Verschieben eines Datentupels innerhalb seiner ursprünglichen Seite; in Abbildung 4.2 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

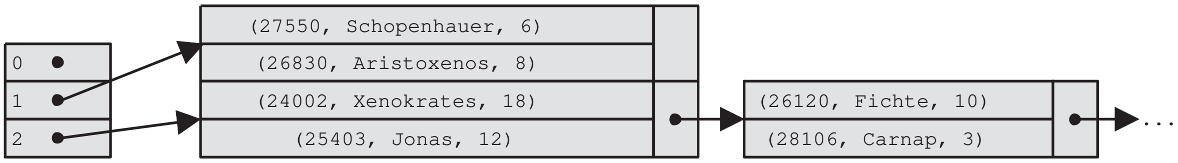


Abbildung 4.3: Hash-Tabelle mit Einstieg in Behälter

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 Anzahl 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

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

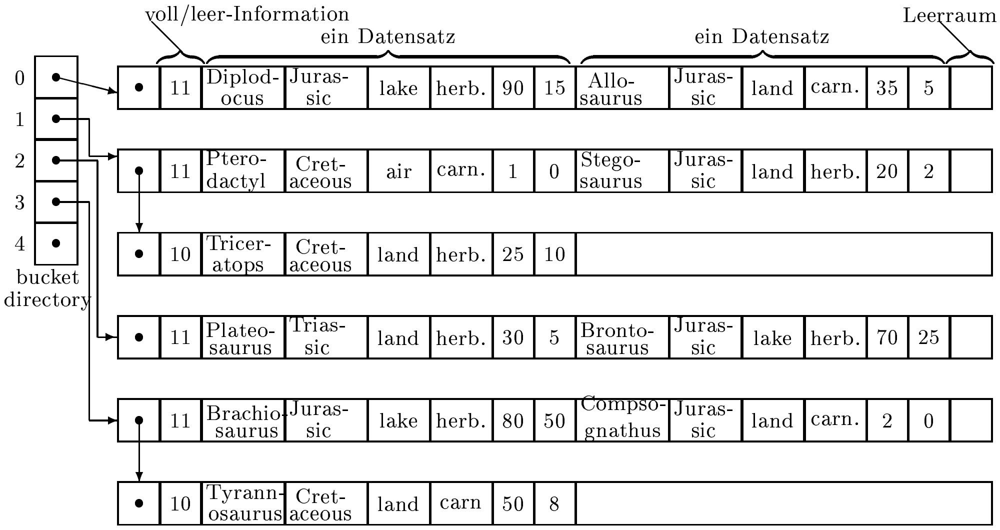


Abbildung 4.4: Hash-Organisation vor Einfügen von Elasmosaurus

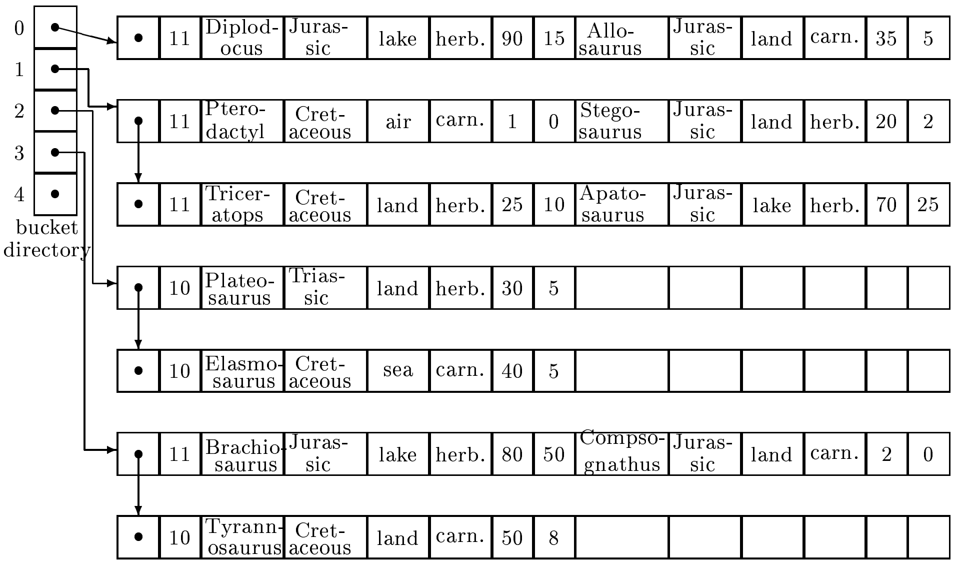


Abbildung 4.5: Hash-Organisation nach Einfügen von Elasmosaurus und Umbenennen

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 4.3 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 2 Bits pro Subblock

(Platz für ein Record), die angeben, ob dieser Subblock leer (also beschreibbar), gefüllt (also lesbar) oder gelöscht (also nicht zum Lesen geeignet) ist. Gelöschte Records werden wegen der Gefahr hängender Zeiger bis zum generellen Aufräumen nicht wieder verwendet.

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 beteiligt: DELETE und INSERT durchführen. Falls Schlüssel nicht beteiligt: 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 4.4 und 4.5 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 4.4 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 4.5 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. Da dies sehr zeitaufwendig ist, wurde als Alternative das *erweiterbare Hashing* entwickelt.

4.4 Erweiterbares Hashing

Durch $h(x) = dp$ wird nun die binäre Darstellung des Funktionswerts der Hash-Funktion in zwei Teile zerlegt. Der vordere Teil d gibt die Position des zuständigen Behälters innerhalb des Hashing-Verzeichnis an. Die Größe von d wird die *globale Tiefe* t genannt. p ist der zur Zeit nicht benutzte Teil des Schlüssels.

	h(x)	
x	d	p
2125	1	01100100001
2126	0	11100100001
2127	1	11100100001

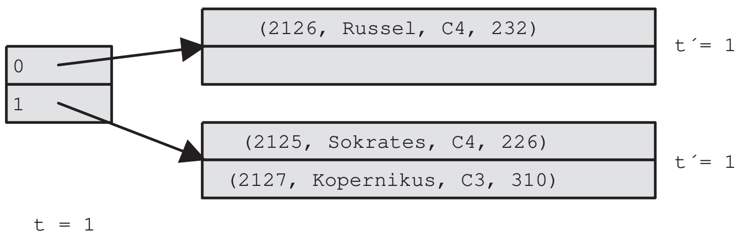


Abbildung 4.6: Hash-Index mit globaler Tiefe 1

	h(x)	
x	d	p
2125	10	1100100001
2126	01	1100100001
2127	11	1100100001
2129	10	0010100001

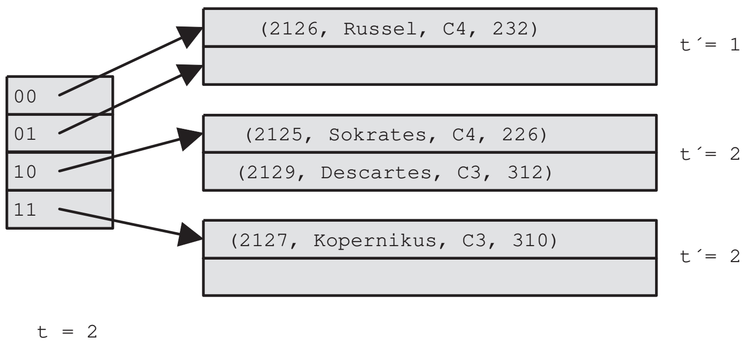


Abbildung 4.7: Hash-Index mit globaler Tiefe 2

Soll ein neuer Datensatz in einen bereits vollen Behälter eingetragen werden, erfolgt eine Aufteilung anhand eines weiteren Bit des bisher unbenutzen Teils p. Ist die globale Tiefe nicht ausreichend, um den Verweis auf den neuen Behälter eintragen zu können, muß das

Verzeichnis verdoppelt werden. Die *lokale Tiefe t'* eines Behälters gibt an, wieviele Bits des Schlüssels für diesen Behälter tatsächlich verwendet werden.

Abbildung 4.6 zeigt eine Hashing-Organisation für Datentupel aus dem Entity-Typ *Dozent* mit den Attributen *PersNr, Name, Rang, Raum*. Je zwei Records finden in einem Behälter Platz. Als Hash-Funktion wird die umgekehrte binäre Darstellung der Personalnummer verwendet. Die globale Tiefe beträgt zur Zeit 1, d.h. nur das vorderste Bit entscheidet über den Index des zuständigen Behälters.

Nun soll `Descartes` eingefügt werden. Das vorderste Bit seines Hash-Werts ist 1 und bildet ihn auf den bereits vollen, mit `Sokrates` und `Kopernikus` gefüllten Behälter ab. Da die globale Tiefe mit der lokalen Tiefe des Behälters übereinstimmt, muß das Verzeichnis verdoppelt werden. Anschließend kann `Descartes` in den zuständigen Behälter auf Position `10` eingefügt werden.

Abbildung 4.7 zeigt, daß der Behälter mit `Russel` weiterhin die lokale Tiefe 1 besitzt, also nur das vorderste Bit zur Adressierung verwendet. Bei zwei weiteren Dozenten mit den Personalnummern `2124` bzw. `2128` würde durch das erste Bit ihrer Hash-Werte `00110010001` bzw. `000010110000` der nullte Behälter überlaufen. Da dessen lokale Tiefe 1 kleiner als die globale Tiefe 2 ist, braucht das Verzeichnis nicht verdoppelt zu werden, sondern ein neues Verteilen aller drei mit `0` beginnenden Einträge auf zwei Behälter mit Index `00` und `01` reicht aus.

Werden Daten gelöscht, so können Behälter verschmolzen werden, wenn ihre Inhalte in einem Behälter Platz haben, ihre lokalen Tiefen übereinstimmen und der Wert der ersten $t' - 1$ Bits ihrer Hash-Werte übereinstimmen Dies wäre zum Beispiel in Bild 4.7 der Fall für die Behälter mit Index `10` und `11` nach dem Entfernen von `Kopernikus`. Das Verzeichnis kann halbiert werden, wenn alle lokalen Tiefen kleiner sind als die globale Tiefe t.

4.5 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*).

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

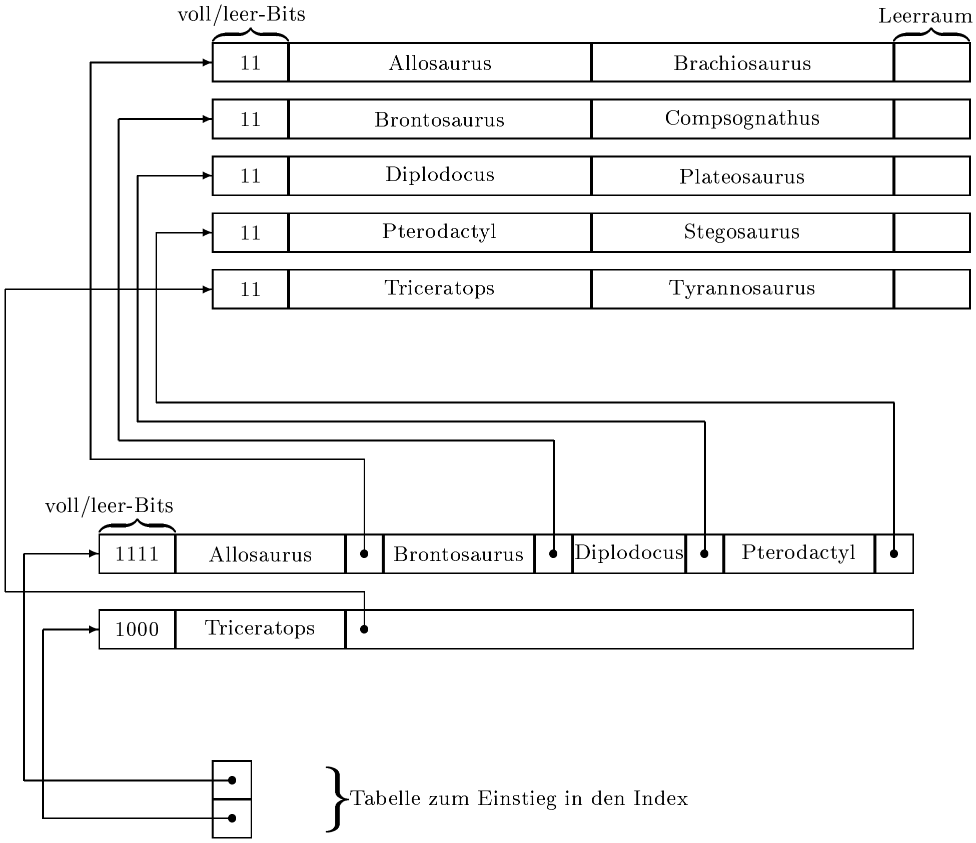


Abbildung 4.8: ISAM: Ausgangslage

- **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.

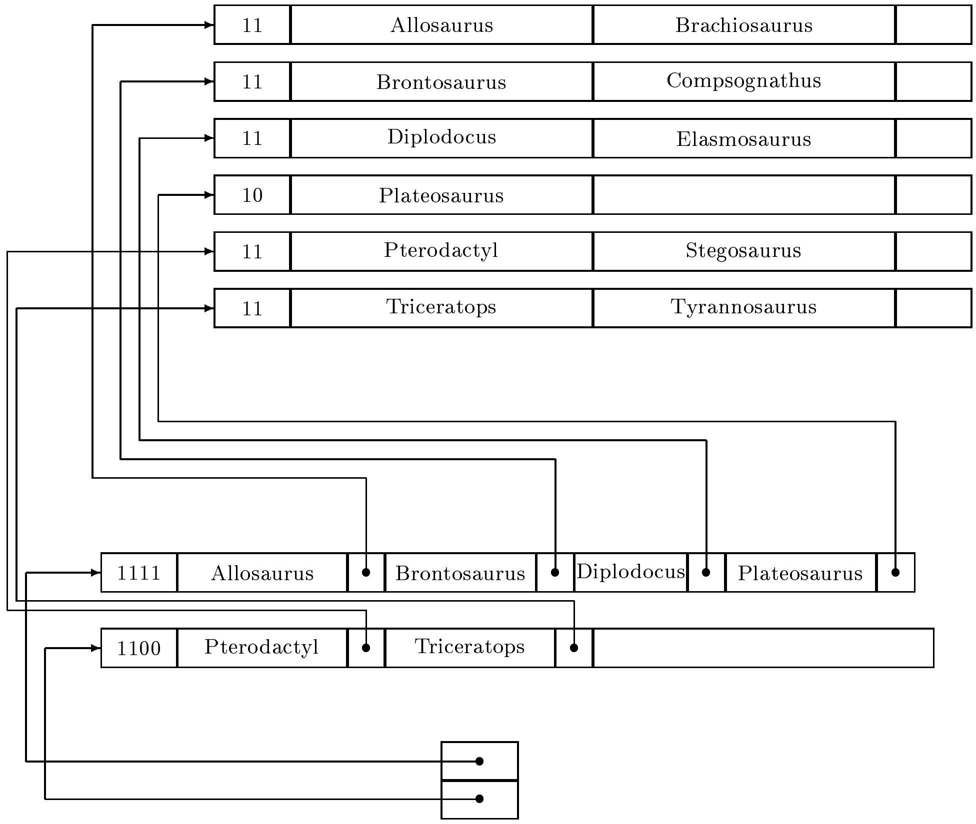


Abbildung 4.9: ISAM: nach Einfügen von Elasmosaurus

- **DELETE:** analog zu INSERT

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

Abbildung 4.8 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.

Abbildung 4.9 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`.

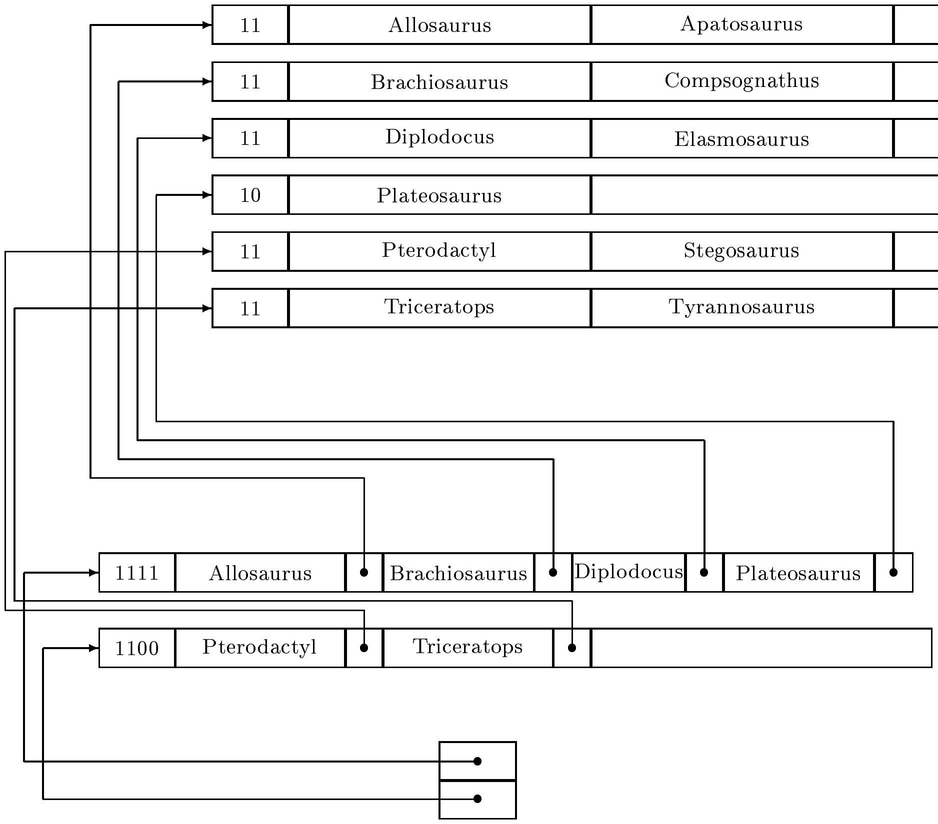


Abbildung 4.10: ISAM: nach Umbenennen von Brontosaurus

4.6 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.

Abbildung 4.11 zeigt das dynamische Verhalten eines B*-Baums mit dem Parameter $k = 2$. Es werden nacheinander die Schlüssel `3, 7, 1, 16, 4, 14, 12, 6, 2, 15, 13, 8, 10, 5, 11, 9` eingefügt und insgesamt 8 Schnappschüsse zu folgenden Zeitpunkten gezeichnet:

```
3,7,1,16,4,14,12,6,2,15, 13,8,10,5,11,9
        ↑ ↑      ↑ ↑  ↑    ↑  ↑      ↑
        1.2.     3.4. 5.   6. 7.     8.
```

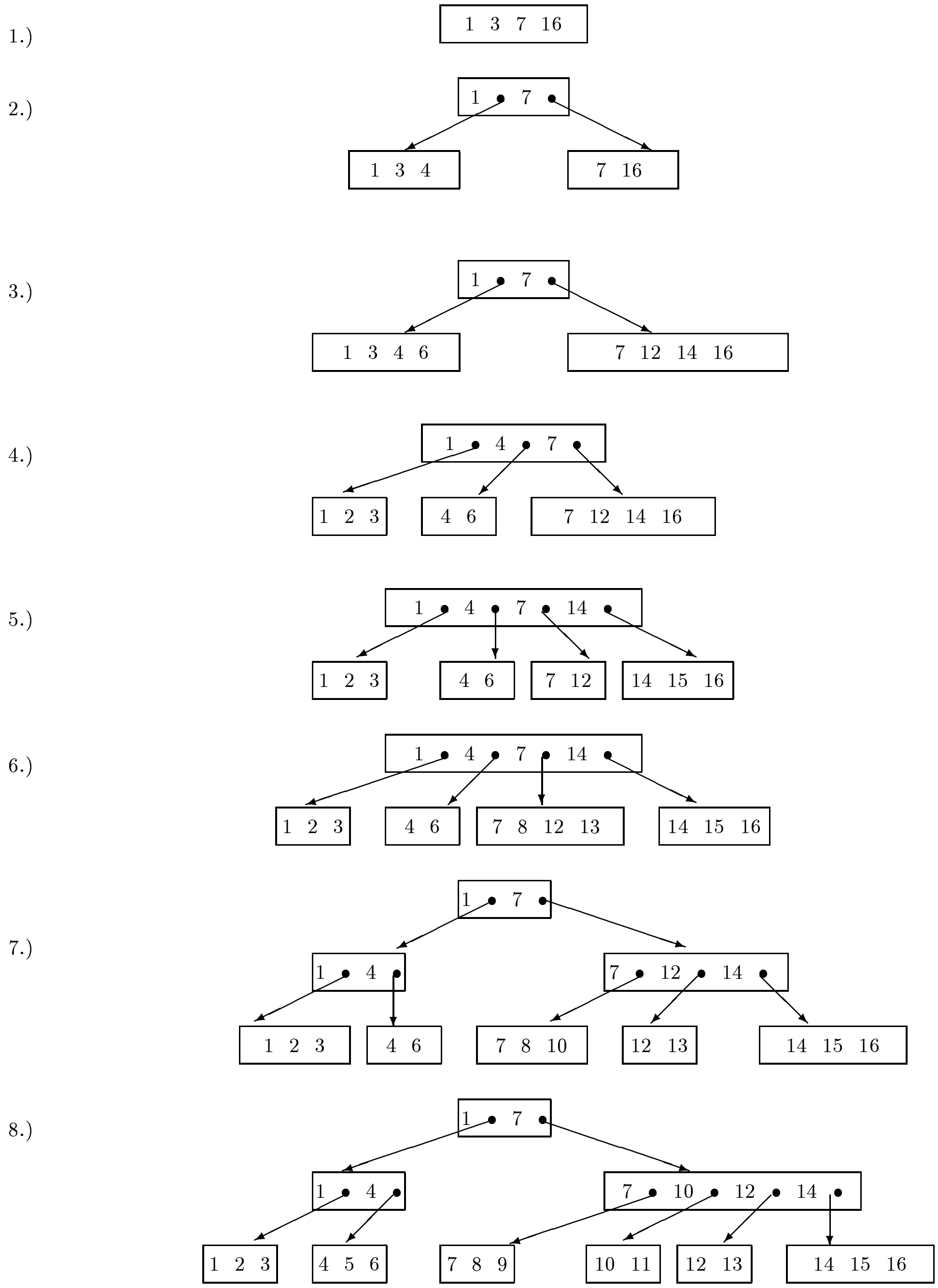


Abbildung 4.11: dynamisches Verhalten eines B*-Baums

Der Parameter k ergibt sich aus dem Platzbedarf für die Schlüssel/Adreßpaare 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:

Höhe	Anzahl Knoten	Anzahl Zeiger		
0	1	26		
1	26	$26 \cdot 39$	=	1.014
2	$26 \cdot 39$	$26 \cdot 39 \cdot 39$	=	39.546

Damit reicht die Höhe 2 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 $4,5 \cdot 10 = 45$ 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.

4.7 Sekundär-Index

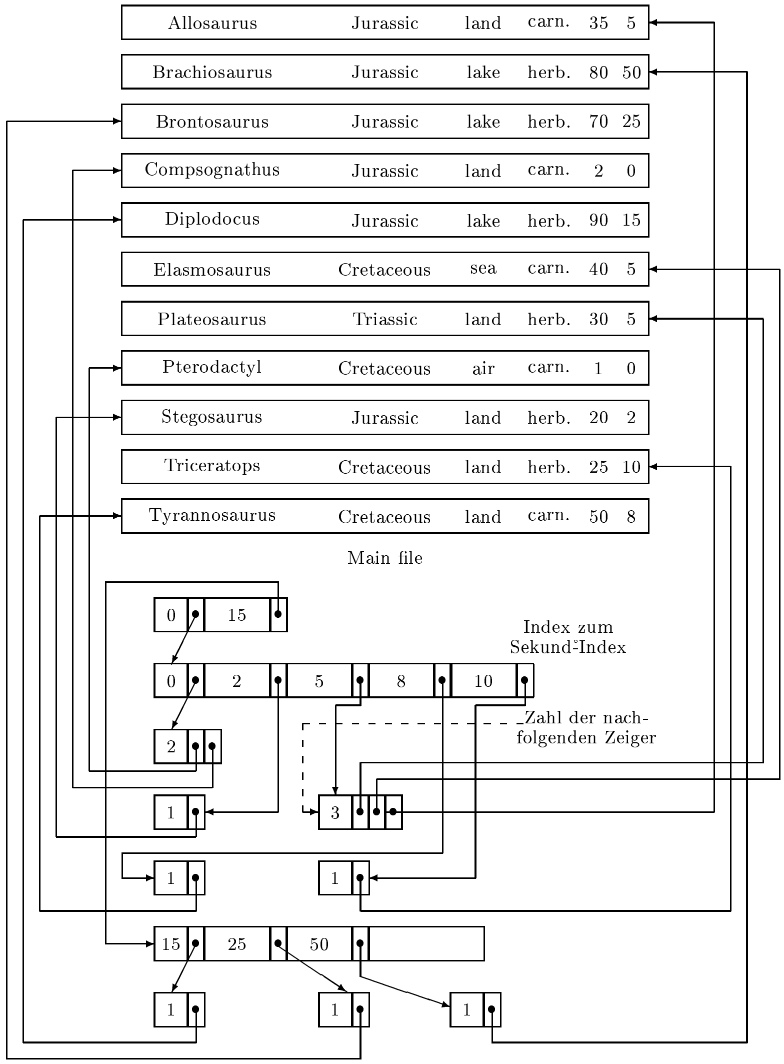


Abbildung 4.12: Sekundär-Index für GEWICHT

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 4.12 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.

Kapitel 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 DM
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.

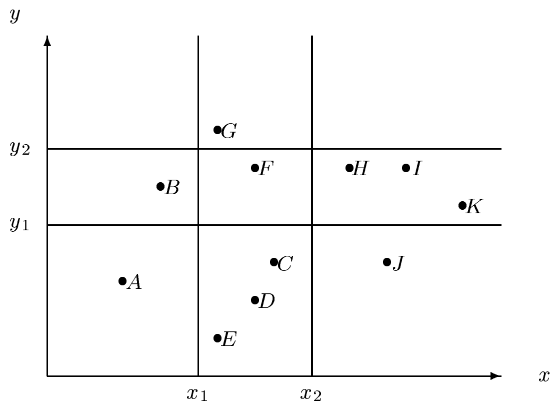


Abbildung 5.1: Fläche mit Datenpunkten

Abbildung 5-1 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 $\mathrm{O}(k + \log n)$ bei k Treffern in einem Suchbaum mit n Knoten.

5.2 k-d-Baum

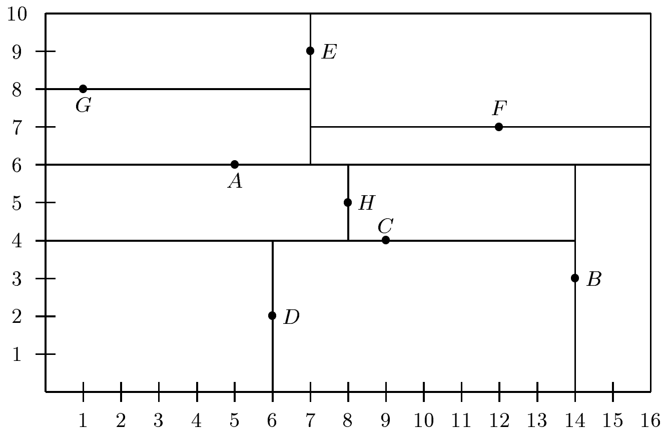


Abbildung 5.2: Durch die Datenpunkte A,B,C,D,E,F,G,H partitionierte Fläche

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 5.2 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 5.3). In der inhomogenen Variante enhält jeder Baumknoten nur einen Schlüssel und die Blätter verweisen auf die Datenrecords (Abbildung 5.4). 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.

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 $> y$

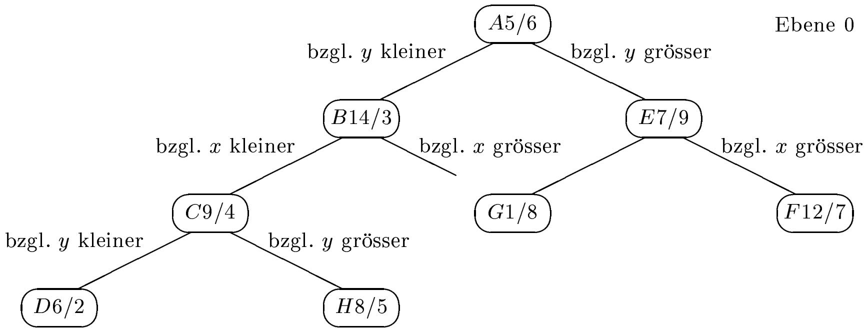


Abbildung 5.3: 2-d-Baum (homogen) zu den Datenpunkten A,B,C,D,E,F,G,H

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

- **Insert**:
  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.

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 5.5 zu sehen ist. In Abbildung 5.4 befinden sich z.B. die Datenrecords C, B und D in einem Block.

Abbildung 5.6 zeigt, daß neben der oben beschriebenen 2-d-Baum-Strategie eine weitere Möglichkeit existiert, den Datenraum zu partitionieren. Dies führt zu den sogenannten *Gitterverfahren*.

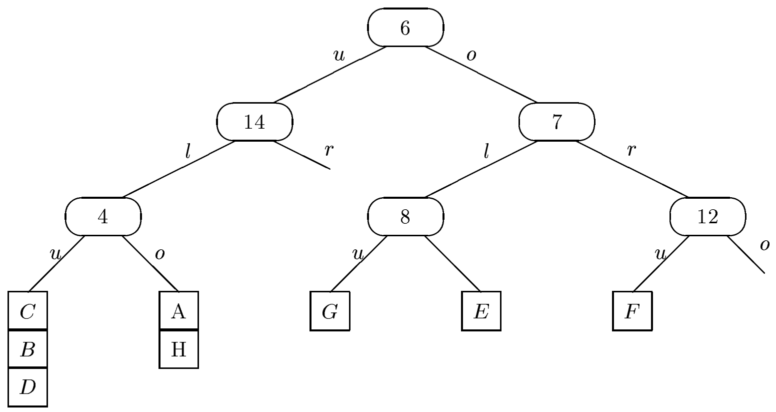


Abbildung 5.4: 2-d-Baum (inhomogen) zu den Datenpunkten A,B,C,D,E,F,G,H

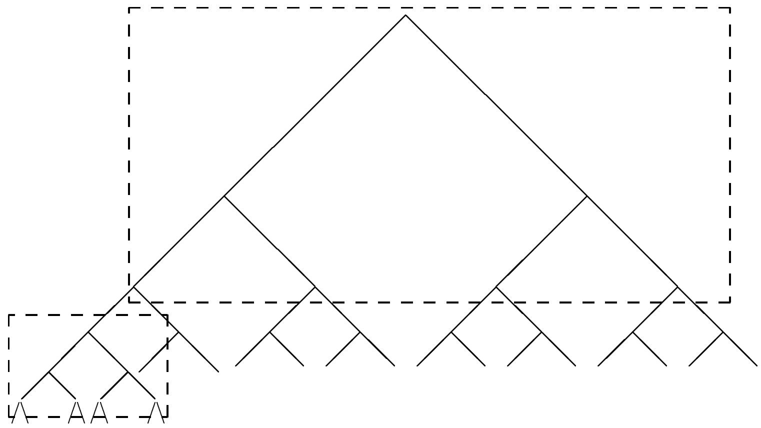


Abbildung 5.5: Zusammenfassung von je 7 inneren Knoten auf einem Index-Block

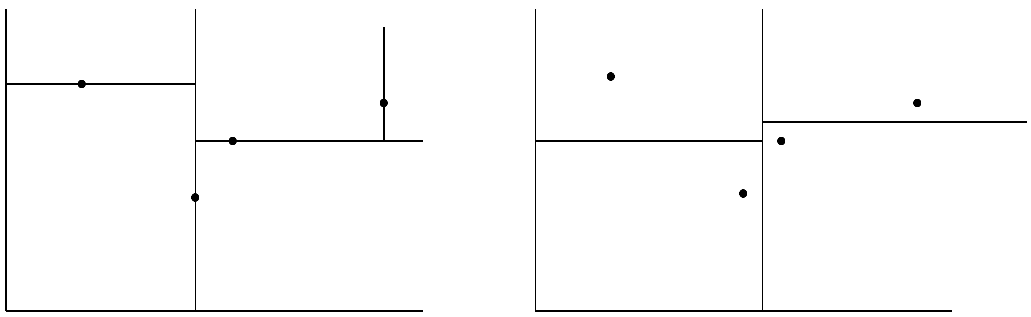


Abbildung 5.6: Partitionierungsmöglichkeiten des Raumes

5.3 Gitterverfahren mit konstanter Gittergröße

Gitterverfahren, die mit konstanter Gittergröße arbeiten, teilen den Datenraum in Quadrate fester Größe auf. Abbildung 5.7 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.

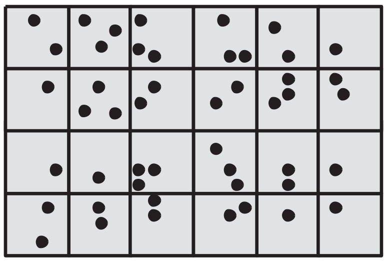


Abbildung 5.7: Partitionierung mit fester Gittergröße

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-Direktory 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 5.8 gestrichelt umrandet). Achtung: mehrere Gitterzellen können im selben Bucket liegen.

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

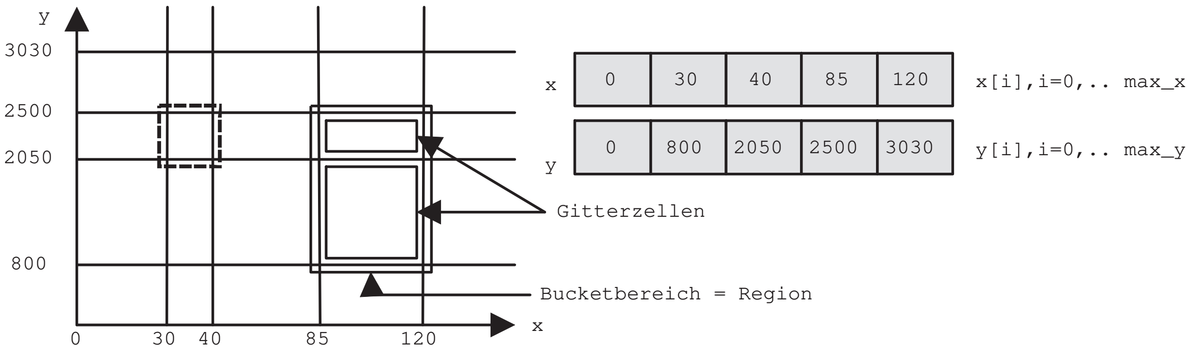


Abbildung 5.8: Skalen und resultierende Gitterzellen

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:
Vorhanden seien 1.000.000 Datentupel, jeweils 4 passen in einen Block. Die X- und die Y-Achse habe jeweils 500 Unterteilungen. Daraus ergeben sich 250.000 Einträge für das Bucket-Directory G. Bei 4 Bytes pro Zeiger und 1024 Bytes pro Block passen 250 Zeiger auf einen Directory-Block. Also gibt es 1000 Directory-Blöcke. D.h. $G[i, j]$ findet sich auf Block $2 \cdot j$ als i-te Adresse, falls $i < 250$ und befindet sich auf Block $2 \cdot j + 1$ als $(i - 250)$-te Adresse, falls $i \geq 250$

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

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

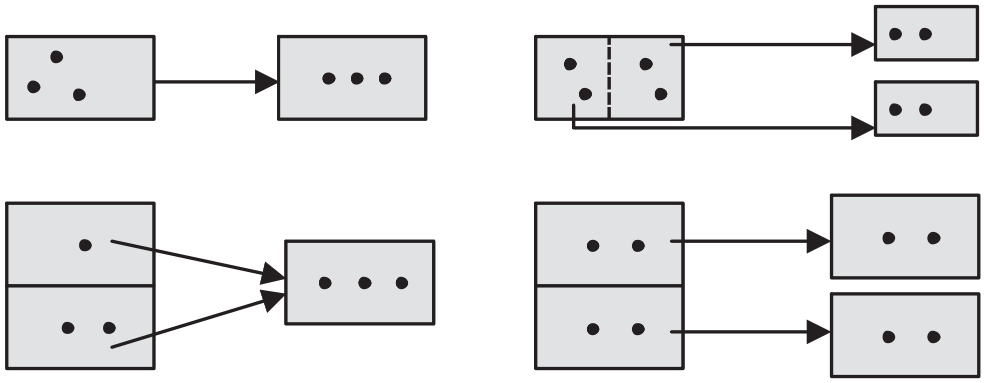


Abbildung 5.9: Konsequenzen eines Bucket-Überlauf (mit und ohne Gitterverfeinerung)

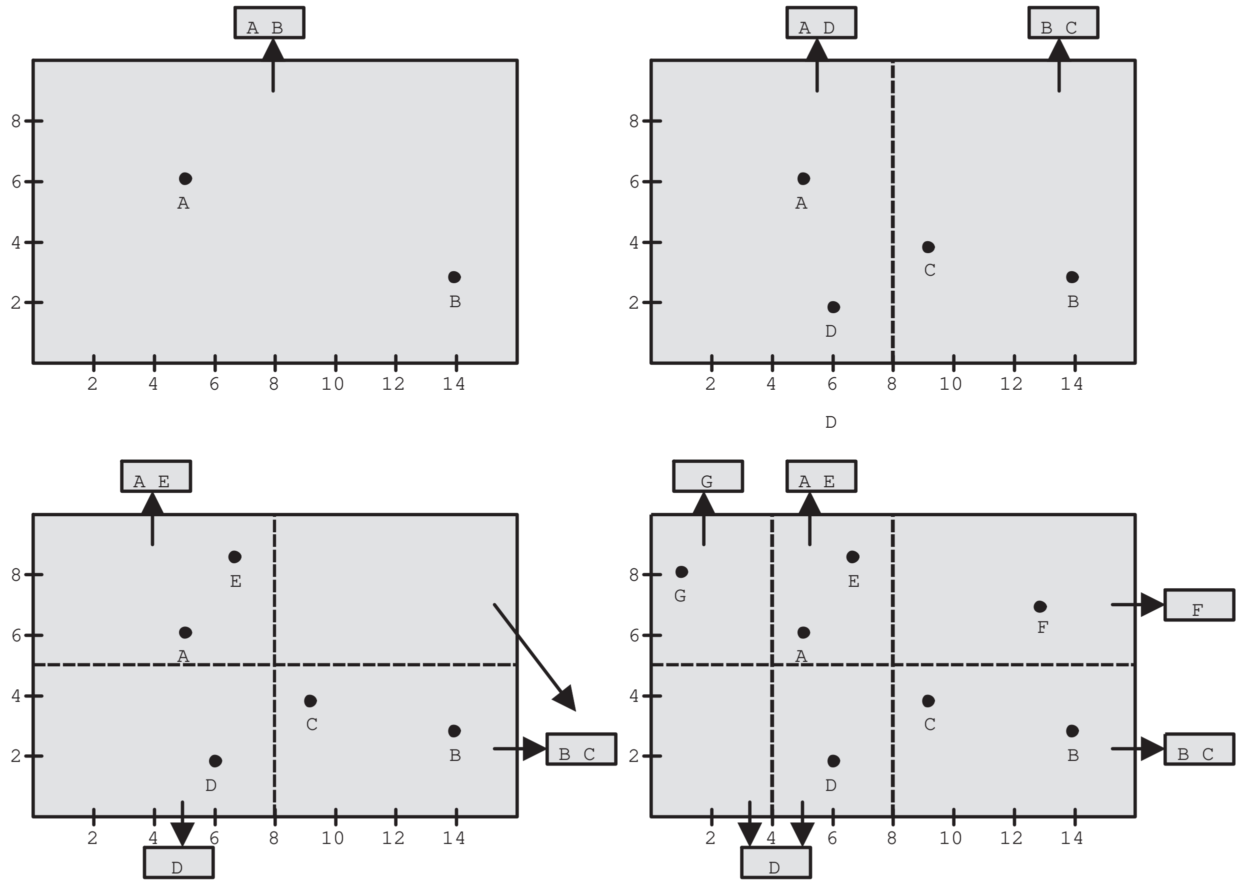


Abbildung 5.10: Aufspalten der Regionen für Datenpunkte A, B, C, D, E, F, G

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 5.9 demonstriert unter der Annahme, daß drei Datenrecords in ein Datenbucket passen. In der unteren Hälfte von Abbildung 5.9 ist zu sehen, daß bei Überlauf eines Buckets, dessen Region aus mehreren Gitterzellen besteht, keine Gitterverfeinerung erforderlich ist.

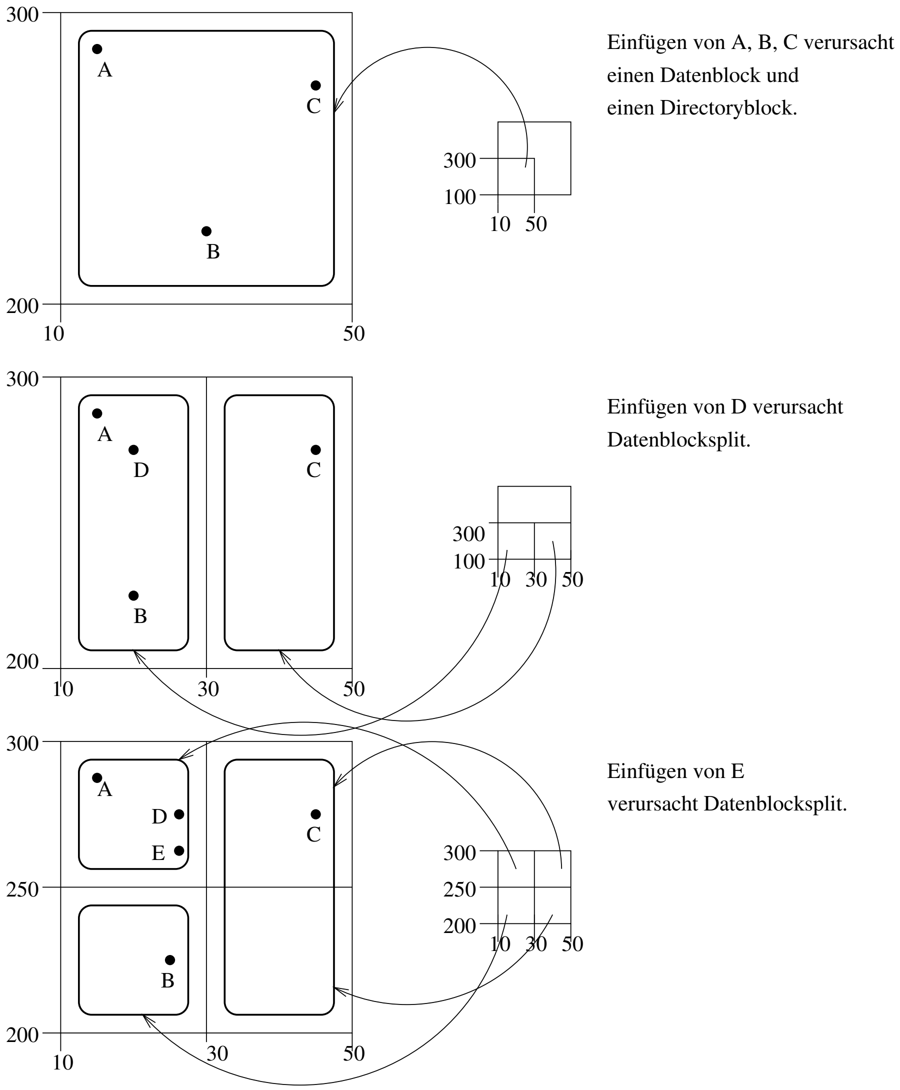


Abbildung 5.11: Dynamik des Grid File beim Einfügen der Datenpunkte A,B,C,D,E

Abbildung 5.10 zeigt die durch das sukzessive Einfügen in ein Grid File entwickelte Dynamik. Es handelt sich dabei um die in Abbildung 4.14 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

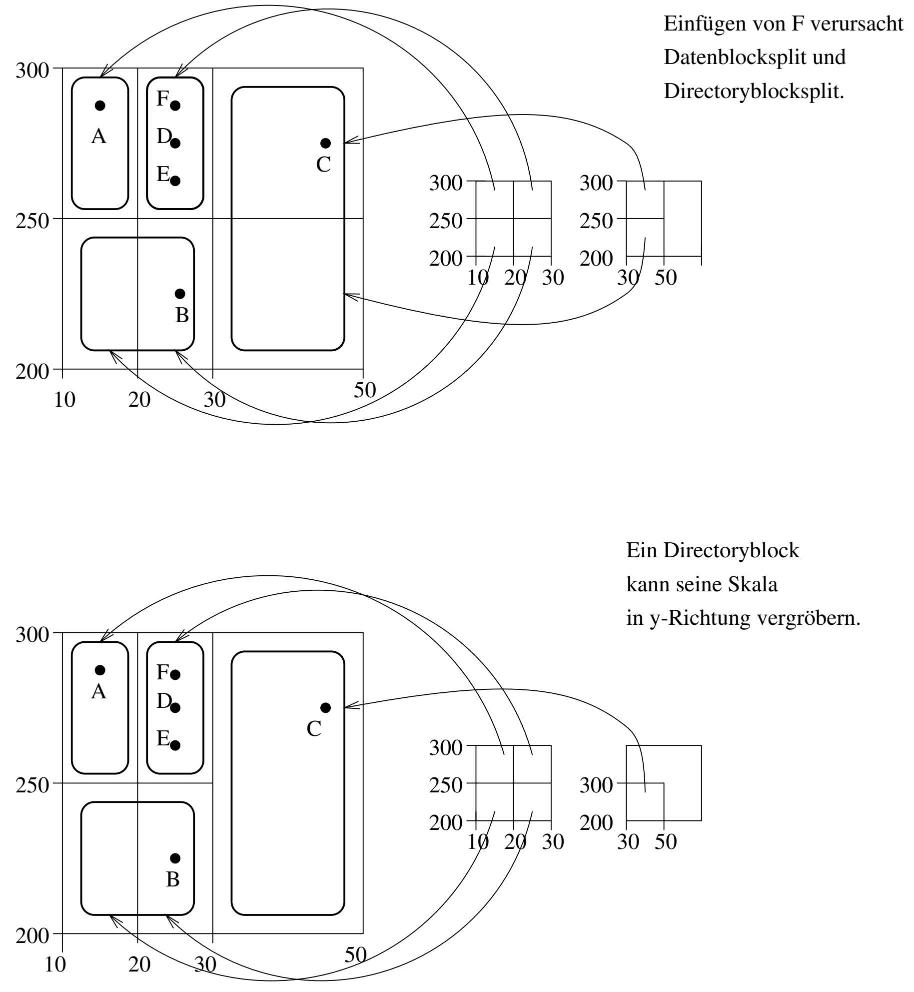


Abbildung 5.12: Vergröberung des Grid Directory nach Aufspalten

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 5.12 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 bilden (Abbildung 5.13). 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)

Abbildung 5.13: 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.

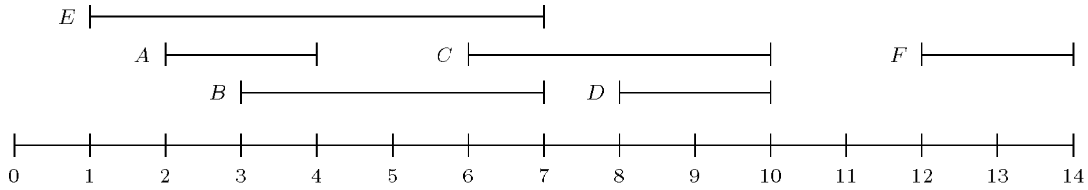


Abbildung 5.14: Intervalle A,B,C,D,E,F über der Zahlengeraden

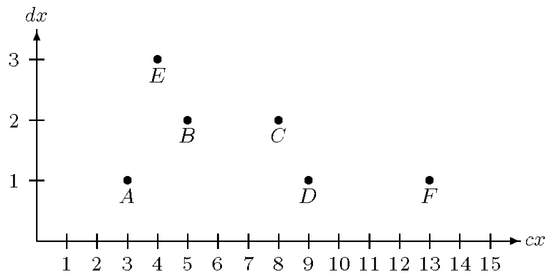


Abbildung 5.15: Repräsentation von Intervallen durch Punkte

Abbildung 5.14 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 5.15 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.

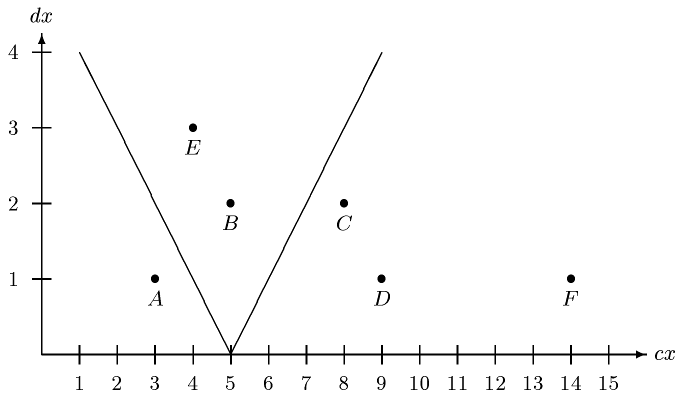


Abbildung 5.16: Abfragekegel zu Punkt p=5

Abbildung 5.16 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 cx und halber Länge dx enhalten ist, wenn gilt: $cx - dx \leq p \leq cx + dx$

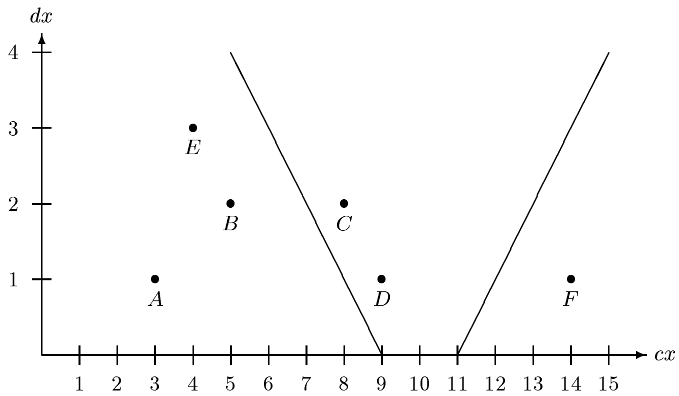


Abbildung 5.17: Abfragekegel zu Intervall mit p=10 und d=1

Abbildung 5.17 zeigt den kegelförmigen Abfragebereich zu dem Query-Intervall mit Mittelpunkt p=10 und halber Länge d=1, in dem alle Intervalle (repäsentiert durch Punkte) liegen, die das Query-Intervall schneiden. Grundlage ist die Überlegung, daß ein Intervall mit Mitte p und halber Länge d genau dann ein Intervall mit Mitte cx und halber Länge dx schneidet, wenn gilt: $cx - dx \leq p + d$ und $cx + dx \geq p - d$

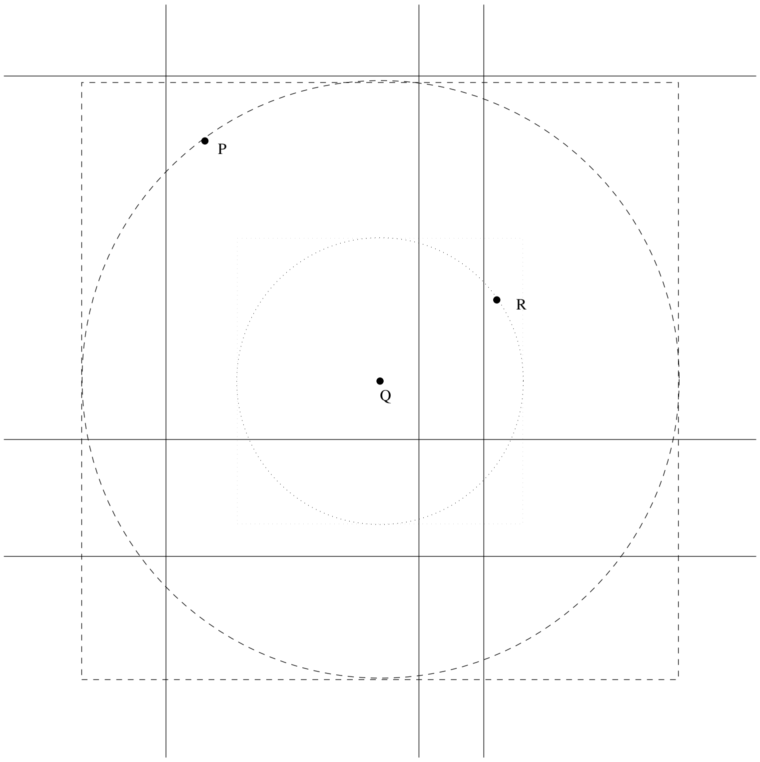


Abbildung 5.18: Nearest-Neighbor-Suche zu Query-Punkt Q

Abbildung 5.18 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.

Kapitel 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 $\mathbf{sch}(R)$ oder mit $\mathcal{R} = A_1, \ldots,\ A_n$ die Menge der Attribute und mit R die aktuelle Ausprägung. Mit $\mathbf{dom}(A)$ 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, <u>TelefonNr : integer</u>] }

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, <u>TelefonNr</u>] }

6.2 Umsetzung in ein relationales Schema

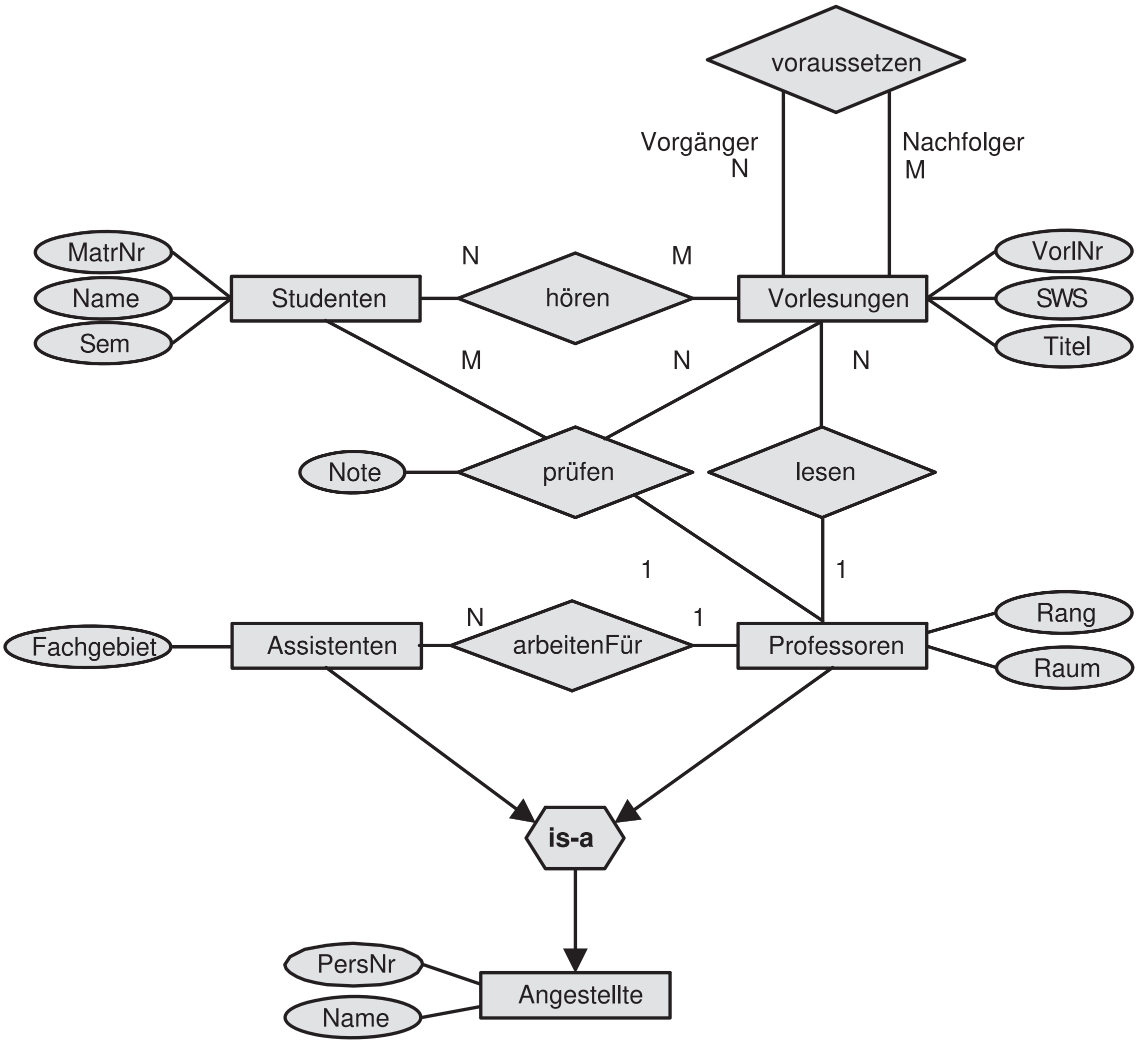


Abbildung 6.1: Konzeptuelles Schema der Universität

Das ER-Modell besitzt zwei grundlegende Strukturierungskonzepte:

- Entity-Typen
- Relationship-Typen

Abbildung 6.1 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:

$$
\begin{array}{rcl}
\text{Studenten} & : & \{[\ \underline{\text{MatrNr : integer}}, \text{Name : string, Semester : integer}]\ \} \\
\text{Vorlesungen} & : & \{[\ \underline{\text{VorlNr : integer}}, \text{Titel : string, SWS : integer}]\ \} \\
\text{Professoren} & : & \{[\ \underline{\text{PersNr : integer}}, \text{Name : string, Rang : string, Raum : integer}]\ \} \\
\text{Assistenten} & : & \{[\ \underline{\text{PersNr : integer}}, \text{Name : string, Fachgebiet : string}]\ \}
\end{array}
$$

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.*

Für das Universitätsschema entstehen aus den Relationships die folgenden Schemata:

$$
\begin{array}{rcl}
\text{hören} & : & \{[\ \underline{\text{MatrNr : integer}}, \underline{\text{VorlNr : integer}}]\ \} \\
\text{lesen} & : & \{[\ \text{PersNr : integer}, \underline{\text{VorlNr : integer}}]\ \} \\
\text{arbeitenFür} & : & \{[\ \underline{\text{AssiPersNr : integer}}, \text{ProfPersNr : integer}]\ \} \\
\text{voraussetzen} & : & \{[\ \underline{\text{Vorgänger : integer}}, \underline{\text{Nachfolger : integer}}]\ \} \\
\text{prüfen} & : & \{[\ \underline{\text{MatrNr : integer}}, \underline{\text{VorlNr : integer}}, \text{PersNr : integer, Note : decimal}]\ \}
\end{array}
$$

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, daß *MatrNr* und *VorlNr* zusammen den Schlüssel bilden.

6.3 Verfeinerung des relationalen Schemas

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

Professoren´				
PersNrN	liestVorl	Name	Rang	Raum
2125	5041	Sokrates	C4	226
2125	5049	Sokrates	C4	226
2125	4052	Sokrates	C4	226

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

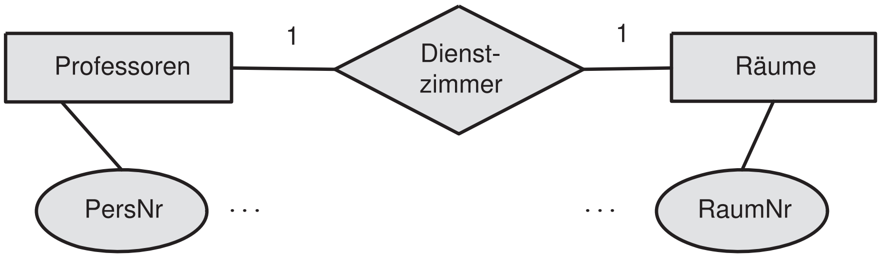


Abbildung 6.2: Beispiel einer 1:1-Beziehung

Abbildung 6.2 zeigt eine mögliche Modellierung für die Unterbringung von Professoren in Räumen als 1 : 1-Beziehung. Die hierzu gehörenden Relationen heißen

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

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

$$\begin{array}{rcl} \text{Professoren} & : & \{[\ \underline{\text{PersNr}}, \text{Name, Rang, Raum}]\ \} \\ \text{Räume} & : & \{[\ \underline{\text{RaumNr}}, \text{Größe, Lage}\ ]\ \} \end{array}$$

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

$$\begin{array}{rcl} \text{Professoren} & : & \{[\ \underline{\text{PersNr}}, \text{Name, Rang}]\ \} \\ \text{Räume} & : & \{[\ \underline{\text{RaumNr}}, \text{Größe, Lage, ProfPersNr}\ ]\ \} \end{array}$$

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

Die in Abbildung 6.1 gezeigte Generalisierung von *Assistenten* und *Professoren* zu *Angestellte* könnte wie folgt durch drei Relationen dargestellt werden:

$$\begin{array}{rcl} \text{Angestellte} & : & \{[\ \underline{\text{PersNr}}, \text{Name}]\ \} \\ \text{Professoren} & : & \{[\ \underline{\text{PersNr}}, \text{Rang, Raum}]\ \} \\ \text{Assistenten} & : & \{[\ \underline{\text{PersNr}}, \text{Fachgebiet}]\ \} \end{array}$$

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).

Tabelle 6.1 zeigt eine Beispiel-Ausprägung der Universitäts-Datenbasis. Das zugrundeliegende Schema enthält folgende Relationen:

$$\begin{array}{rcl} \text{Studenten} & : & \{[\ \underline{\text{MatrNr : integer}}, \text{Name : string, Semester : integer}]\ \} \\ \text{Vorlesungen} & : & \{[\ \underline{\text{VorlNr : integer}}, \text{Titel : string, SWS : integer, gelesenVon : integer}]\ \} \\ \text{Professoren} & : & \{[\ \underline{\text{PersNr : integer}}, \text{Name : string, Rang : string, Raum : integer}]\ \} \\ \text{Assistenten} & : & \{[\ \underline{\text{PersNr : integer}}, \text{Name : string, Fachgebiet : string, Boss : integer}]\ \} \\ \text{hören} & : & \{[\ \underline{\text{MatNr : integer, VorlNr : integer}}]\ \} \\ \text{voraussetzen} & : & \{[\ \underline{\text{Vorgänger : integer, Nachfolger : integer}}]\ \} \\ \text{prüfen} & : & \{[\ \underline{\text{MatrNr : integer, VorlNr : integer}}, \text{PersNr : integer, Note :decimal}]\ \} \end{array}$$

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
28106	5041
28106	5052
28106	5216
28106	5259
29120	5001
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

Abbildung 6.3: Beispielausprägung der Universitäts-Datenbank

Zur Modellierung von schwachen Entity-Typen betrachten wir Abbildung 6.4, 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:

$$\text{Räume} : \{[\ \underline{\text{GebNr, RaumNr}}, \text{Größe}]\ \}$$

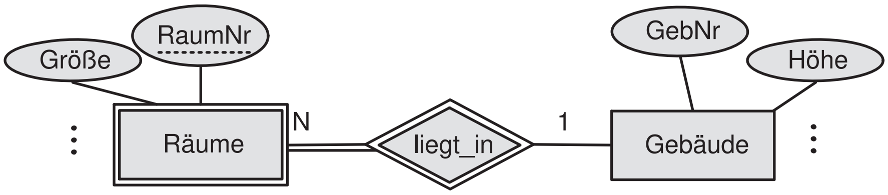


Abbildung 6.4: Schwacher Entity-Typ

Ein 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:

$$\text{bewohnt} : \{[\ \underline{\text{PersNr, GebNr, RaumNr}}]\ \}$$

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 6.1 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 $\wedge, \vee, \neg, \exists, \forall$.
- SQL (kommerziell):
  Stellt eine in Umgangssprache gegossene Mischung aus Relationenalgebra und Relationenkalkül dar.
- Query by Example (für Analphabeten):
  Verlangt vom Anwender das Ausfüllen eines Gerüstes mit Beispiel-Einträgen.

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 Projektion

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

liefert als Ergebnis

$\sigma_{Semester>10}(Studenten)$		
MatNr	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 $\cup$, zusammengefaßt 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 × hören					
Professoren				hören	
PersNr	name	Rang	Raum	MatNr	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 ein kartesisches Produkt der Tabelle mit sich selbst erforderlich, nachdem zuvor die Spalten umbenannt worden sind:

$$\Pi_{V1.Vorgänger}(\sigma_{V2.Nachfolger=5216 \wedge V1.Nachfolger=V2.Vorgänger}$$

$$(\rho_{V1}(voraussetzen) \times \rho_{V2}(voraussetzen)))$$

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, daß 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 läßt 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:

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

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 äußeren 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

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 läßt sich mithilfe der Mengendifferenz bilden:

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

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:

$$\begin{array}{|c|c|}\hline \multicolumn{2}{|c|}{H} \\ \hline M & V \\ \hline m_1 & v_1 \\ m_1 & v_2 \\ m_1 & v_3 \\ m_2 & v_2 \\ m_2 & v_3 \\ \hline \end{array} \;\div\; \begin{array}{|c|}\hline L \\ \hline V \\ \hline v_1 \\ v_2 \\ \hline \end{array} \;=\; \begin{array}{|c|}\hline H \div L \\ \hline M \\ \hline m_1 \\ \hline \end{array}$$

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

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 zuläßt. Es gibt zwei unterschiedliche, aber gleichmächtige Ausprägungen des Relationenkalküls:

- Der relationale Tupelkalkül
- Der relationale Domänenkalkül

6.7 Der relationale Tupelkalkül

Im *relationen 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 muß, damit t in das Ergebnis aufgenommen werden kann. Das Prädikat P wird formuliert unter Verwendung von $\vee, \wedge, \neg, \forall, \exists, \Rightarrow$.

Alle C4-Professoren:

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

Alle Professorennamen zusammen mit den Personalnummern ihrer Assistenten:

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

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:

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, daß das Ergebnis eine endliche Menge war.

Kapitel 7

Relationale Anfragesprachen

7.1 Oracle Datenbank

Stellvertretend für die zahlreichen am Markt befindlichen relationalen Datenbanksysteme wird in diesem Kapitel das System *Oracle, Version 8.05* verwendet. Als Vorbereitungen zum Zugriff sind erforderlich:

Server :

Nach dem Installieren des Oracle Servers muß vom DBA (Data Base Administrator) jeder User mit Passwort, Verbindungsrecht und Arbeitsbereich eingerichtet werden:

- Richte neuen User `erika` mit Passwort `mustermann` ein:
  ```
  create user erika identified by mustermann;
  ```
- Erteile erika das Recht, eine Oracle-Verbindung herzustellen:
  ```
  grant connect to erika;
  ```
- Erteile erika das Recht, in Oracle Objekte anzulegen:
  ```
  grant resource to erika;
  ```
- Weise dem User `erika` den temporären Arbeitsbereich `temp` zu:
  ```
  alter user erika temporary tablespace temp;
  ```
- Liste Angaben zu allen Benutzern:
  ```
  select * from all_users;
  ```
- Erteile `erika` das Leserecht für die Tabelle `Vorlesungen`:
  ```
  grant select on Vorlesungen to erika;
  ```
- Entziehe `erika` das Recht auf Verbindung:
  ```
  revoke connect from erika;
  ```
- Entferne `erika` aus der Datenbank:
  ```
  drop user erika;
  ```

Client :
In jeder Windows-NT-Station wird die Klienten-Software *Oracle SQL*PLUS* installiert. Diese stellt eine ASCII-Schnittstelle zur Verfügung, auf der SQL-Statements abgesetzt werden können.

Verbindung :
Mit dem Modul *Oracle Net8 Easy Config* wird pro Station ein sogenannter Dienst mit frei wählbarem Namen, z.B. `dbs99` eingerichtet, welcher über TCP/IP eine Verbindung zu einer im Server angelegten Datenbank, z. B. `ora1`, herstellt. Erforderlich ist die Angabe des Hostnamens (`dbserv.rz.uni-osnabrueck.de`) und der Portnummer (`1521`). Zum Testen ist der beim Server angelegte Benutzername (`erika`) erforderlich.

ODBC-Treiber :
In der Windows-NT-Systemsteuerung wird der von Oracle mitgelieferte ODBC (Open Data Base Connectivity) Treiber eingerichtet. Hierzu wird eine User Data Source hinzugefügt unter Verwendung des Oracle ODBC Treibers, die einen frei wählbaren Namen erhält, z. B. `dbs99.dsn` und Bezug nimmt auf den eingetragenen Dienst `dbs99`.

MS-Access :
MS-Access kann als Frontend zu Oracle verwendet werden. Hierzu wird in MS-Access nach dem Anlegen einer Datenbank eine Verknüpfung hergestellt zu dem Dateityp *ODBC Datenbanken*. Dort wird der eingetragene Dienst, z. B. `dbs99`, ausgewählt und nach dem Prompt Benutzername und Passwort eingegeben. Danach werden die gewünschten Tabellen ausgewählt.

7.2 SQL

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 SQL2 genannt. Er wird weitgehend vom relationalen Datenbanksystem *Oracle* unterstützt.

7.3 Datentypen in Oracle

Die von Oracle verwendeten Datentypen lauten:

Typ	Wertebereich
character(n)	String fester Länge mit n Zeichen
varchar2(n)	String variabler Länge mit bis zu n Zeichen
integer	ganze Zahl
number(n, m)	Festkommazahl mit n Stellen, davon m nach dem Komma
float	Gleitkommazahlen
date	Zeitangabe (für Daten zwischen 4712 v. Chr bis 4712 n. Chr.)
long	Zeichenkette bis zu 2 GByte
raw	Binärstrings bis zu 255 Bytes
long raw	Binärobjekte bis zu 2 GByte (Soundfiles, Videos, ...)

Nicht besetzte Attributwerte werden durch das Schlüsselwort `NULL` gekennzeichnet.

7.4 Schemadefinition

SQL-Statements zum Anlegen, Ändern und Entfernen einer Tabelle:

1. Tabelle anlegen:

```
create table Professoren (
  PersNr  integer       not null,
  Name    varchar2(10)  not null,
  Rang    character(2) );
```

2. Tabelle erweitern:

```
alter table Professoren
add (Raum integer);
```

3. Tabelle ändern:

```
alter table Professoren
modify (Name varchar2(30));
```

4. Tabelle verkürzen (nicht in Oracle):

```
alter table Professoren
drop column Gebdatum;
```

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 *Date* 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, die eine Semesterangabe haben:

```
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 groß geschrieben:

```
select *
from Vorlesungen
where upper(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 to_char(GebDatum,'month DD, YYYY') AS Geburtstag
from studenten;
```

10. Liste das Alter der Studenten in Jahren:

```
select (sysdate - GebDatum) / 365 as Alter_in_Jahren
from studenten;
```

11. Liste die Wochentage der Geburtsdaten der Studenten:

```
select to_char(GebDatum,'day')
from studenten;
```

12. Liste die Uhrzeiten der Geburtsdaten der Studenten:

```
select to_char(GebDatum,'hh:mi:ss')
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, s.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, max(GebDatum)
from Professoren
group by rang;
```

18. Liste Summe der SWS pro Professor:

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

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

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

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

```
select gelesenVon, Name, sum(SWS)
from Vorlesungen, Professoren
where gelesenVon = PersNr and Rang = 'C4'
group by gelesenVon, Name
      having avg(SWS) > 3;
```

21. Liste alle Prüfungen, die als Ergebnis die Durchschnittsnote haben:

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

22. Liste alle Professoren zusammen mit ihrer Lehrbelastung (nicht Oracle):

```
select PersNr, Name, (select sum(SWS) as Lehrbelastung
                      from Vorlesungen
                      where gelesenVon = PersNr)
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 p.GebDatum > a.GebDatum;
```

25. Liste alle Studenten mit der Zahl ihrer Vorlesungen, sofern diese Zahl größer 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, tmp.maximum
from  Professoren p,
      (select Rang, max(GebDatum) as maximum
       from Professoren
       group by Rang) tmp
where p.Rang = tmp.Rang and p.GebDat = 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:

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

30. Liste den Durchschnitt von Professoren- und Assistenten-Namen:

```
( 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 größter Semesterzahl:

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

33. Liste Studenten, die nicht die größte 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
    )
  );
```

35. Natürlicher Verbund (nur in SQL-92):

```
select *
from Studenten
natural join hoeren;
```

36. Berechnung der transitiven Hülle einer rekursiven Relation (nur in Oracle):
Liste alle Voraussetzungen für die Vorlesung 'Der Wiener Kreis':

```
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'
        )
);
```

7.7 SQL-Queries zum Einfügen, Modifizieren und Löschen

1. Füge Student mit Matrikelnummer und Name ein:

```
insert into Studenten (MatrNr, Name)
            values (28121, 'Archimedes');
```

2. Alle Studenten sollen die Vorlesung 'Selber Atmen' hören:

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

3. Alle Studenten um 10 Tage älter machen:

```
update studenten
set GebDatum = GebDatum + 10;
```

4. Alle Studenten mit Semesterzahlen größer als 13 löschen:

```
delete from Studenten
   where Semester > 13;
```

5. Niemand soll mehr die Vorlesung 'Selber Atmen' hören:

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

7.8 SQL-Queries 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 läßt

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. Lege Untertyp als Verbund von Obertyp und Erweiterung an:

```
create table Angestellte   (PersNr      integer not null,
                            Name        varchar2(30) not null);
create table ProfDaten     (PersNr      integer not null,
                            Rang        character(2),
                            Raum        integer);
create table AssiDaten      (PersNr     integer not null,
                            Fachgebiet  varchar2(30),
                            Boss        integer);

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

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

5. Lege Obertyp als Vereinigung von Untertypen an:

```
create table Professoren       (PersNr      integer not null,
                                Name        varchar2(30) not null,
                                Rang        character(2),
                                Raum        integer);

create table Assistenten       (PersNr      integer not null,
                                Name        varchar2(30) not null,
                                Fachgebiet  varchar2(30),
                                Boss        integer);

create table AndereAngestellte (PersNr      integer not null,
                                Name        varchar2(30) not null);

create view Angestellte as

  (select PersNr, Name from Professoren) union
  (select PersNr, Name from Assistenten) union
  (select PersNr, Name from AndereAngestellte);
```

7.9 Query by Example

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

Liste alle Vorlesungen von Sokrates mit mehr als 3 SWS:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
		p._t	> 3	`Sokrates`

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

Im Domänenkalkül lautet diese Anfrage

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

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.** , die Aggregatfunktionen heißen wie gewohnt **sum.**, **avg.**, **min.**, **max.** und **cnt.**. Die Duplikateliminierung wird durch **all.** erreicht:

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	

Kapitel 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 statische und dynamische Integritätsbedingungen. Eine statische Bedingung muß 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, daß 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 heißt *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 muß also beim Einfügen, Löschen und Ändern immer sichergestellt sein, daß gilt

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

Erlaubte Änderungen sind daher:

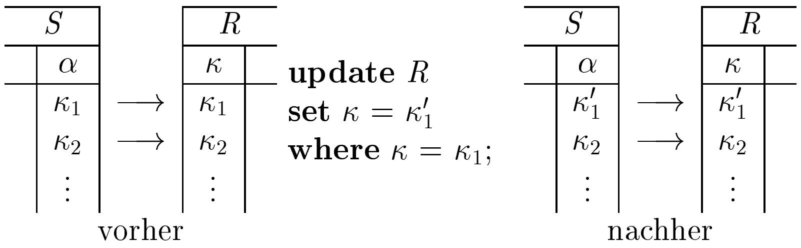
- Einfügen eines Tupels in S verlangt, daß der Fremdschlüssel auf ein existierendes Tupel in R verweist.
- Ändern eines Tupels in S verlangt, daß der neue Fremdschlüssel auf ein existierendes Tupel in R verweist.
- Ändern eines Primärschlüssels in R verlangt, daß kein Tupel aus S auf ihn verwiesen hat.
- 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 heißt `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, daß 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 (Abbildung 8.1a).
- Durch die Klausel `on delete cascade` zieht das Löschen eines Tupels in R das Entfernen des auf ihn verweisenden Tupels in S nach sich (Abbildung 8.1b).
- 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 8.2).

(a) **create table** S (…, α **integer references** R **on update cascade**);



(b) **create table** S (…, α **integer references** R **on delete cascade**);

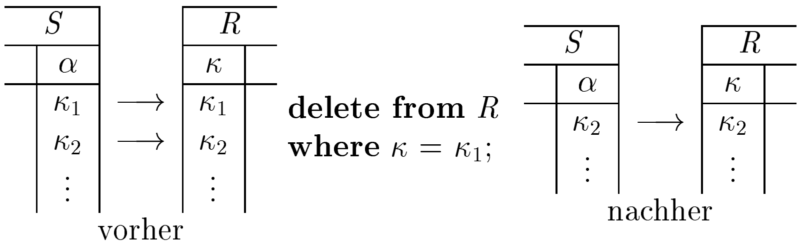
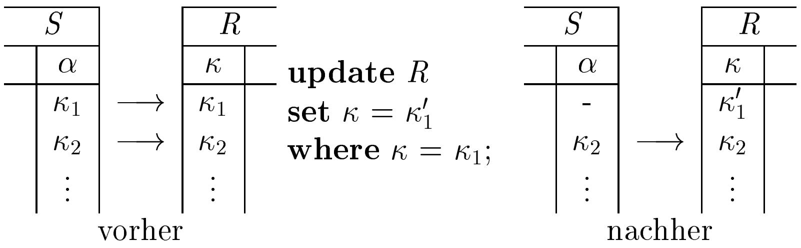


Abbildung 8.1: Referentielle Integrität durch Kaskadieren

a) **create table** S (…, α **integer references** R **on update set null**);



(b) **create table** S (…, α **integer references** R **on delete set null**);

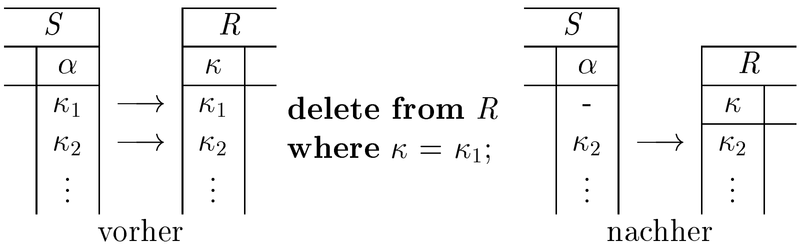
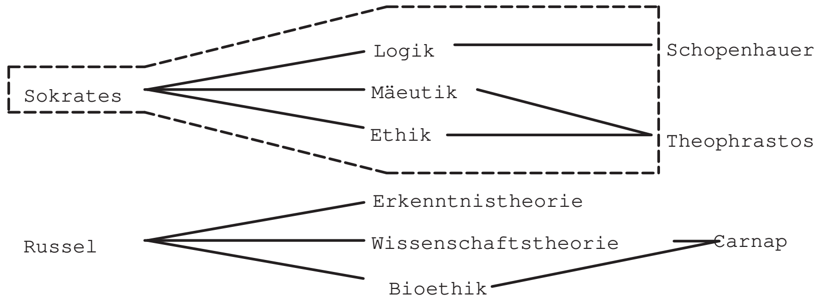


Abbildung 8.2: Referentielle Integrität durch Nullsetzen

Kaskadierendes Löschen kann ggf. eine Kettenreaktion nach sich ziehen. In Abbildung 8.3 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);
```

Abbildung 8.3: 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.

```
CREATE TABLE Studenten
        (MatrNr         INTEGER PRIMARY KEY,
         Name           VARCHAR2(20) NOT NULL,
         Semester       INTEGER,
         GebDatum       DATE);

CREATE TABLE Professoren
        (PersNr         INTEGER PRIMARY KEY,
         Name           VARCHAR2(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           VARCHAR2(20) NOT NULL,
         Fachgebiet     VARCHAR2(20),
         Boss           INTEGER,
         FOREIGN KEY    (Boss) REFERENCES Professoren,
         GebDatum       DATE);

CREATE TABLE Vorlesungen
        (VorlNr         INTEGER PRIMARY KEY,
         Titel          VARCHAR2(20),
         SWS            INTEGER,
         gelesenVon     INTEGER REFERENCES Professoren);

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

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

CREATE TABLE pruefen
        (MatrNr         INTEGER REFERENCES Studenten ON DELETE CASCADE,
         VorlNr         INTEGER REFERENCES Vorlesungen,
         PersNr         INTEGER REFERENCES Professoren,
         Note           NUMERIC(2,1) CHECK (Note between 0.7 and 5.0),
         PRIMARY KEY    (MatrNr, VorlNr));
```

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. SQL-92 sieht diesen Mechanismus noch nicht vor, wohl aber Oracle.

Listing 8.2 zeigt einen Trigger für die Tabelle Professoren, der vor jedem Update die Tupel mit nichtleerem Rang daraufhin untersucht, ob sie eine Degradierung verursachen würden.

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

Listing 8.2: Trigger zur Verhinderung einer Degradierung

Listing 8.3 zeigt einen Trigger, der immer nach dem Einfügen eines Tupels in die Tabelle *hoeren* einen Professor sucht, der jetzt mehr als 10 Hörer hat und ihn dann nach C4 befördert.

```
create or replace trigger befoerderung
after update on hoeren
declare
begin
UPDATE professoren SET rang = 'C4'
WHERE persnr in
  (SELECT persnr
   FROM professoren, vorlesungen v, hoeren h
   WHERE persnr = v.gelesenvon
   and v.vorlnr = h.vorlnr
   GROUP BY persnr
   having count(*) > 10);
end;
```

Listing 8.3: Trigger zum Auslösen einer Beförderung

Das Schlüsselwort `drop` entfernt einen Trigger:

```
drop trigger befoerderung;
```

Kapitel 9

Datenbankapplikationen

9.1 MS-Access

MS-Access ist ein relationales Datenbanksystem der Firma *Microsoft*, welches als Einzel- und Mehrplatzsystem unter Windows 95/98 und Windows NT 4.0 läuft. Seine wichtigsten Eigenschaften:

- Der **Schemaentwurf** geschieht menugesteuert (Abbildung 9.2))
- Referenzen zwischen den Tabellen werden in Form von **Beziehungen** visualisiert.
- **Queries** können per SQL oder menugesteuert abgesetzt werden (Abbildung 9.1).
- **Formulare** definieren Eingabemasken, die das Erfassen und Updaten von Tabellendaten vereinfachen (Abbildung 9.3)
- **Berichte** fassen Tabelleninhalte und Query-Antworten in formatierter Form zusammen und können als Rich-Text-Format exportiert werden (Listing 9.1 + Abbildung 9.4).
- Als **Frontend** eignet es sich für relationale Datenbanksysteme, die per ODBC-Schnittstelle angesprochen werden können.

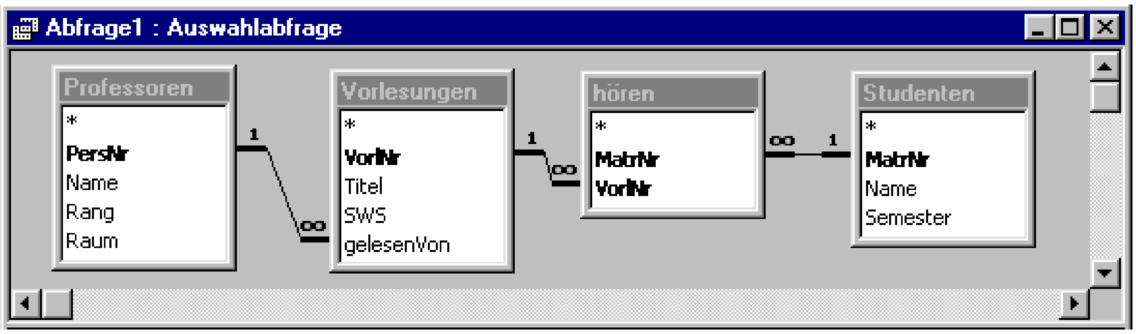


Abbildung 9.1: in MS-Access formulierte Abfrage

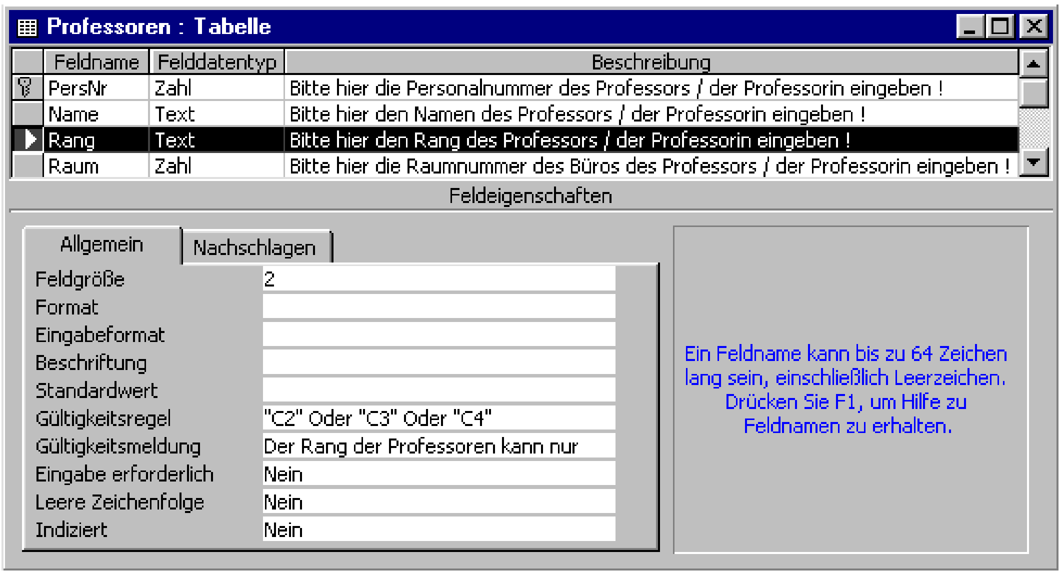


Abbildung 9.2: Schemadefinition in MS-Access

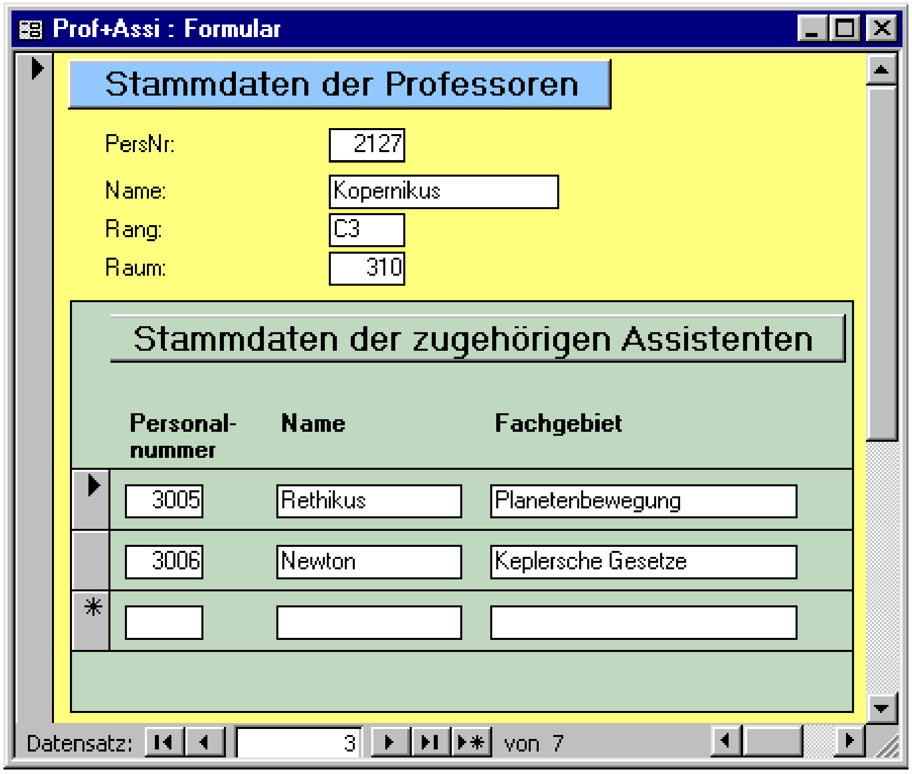


Abbildung 9.3: durch MS-Access-Formular erzeugte Eingabemaske

Listing 9.1 zeigt die Formulierung einer SQL-Abfrage, die zu jedem Professor seine Studenten ermittelt. Aus den Treffern dieser Query wird der Bericht in Abbildung 9.4 generiert.

```
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;
```

Listing 9.1: Abfrage für Bericht in Abbildung 9.4

Dozenten und ihre Hörer

Professor	***Student***
Augustinus	Feuerbach
	Jonas
Kant	Feuerbach
	Fichte
	Schopenhauer
	Theophrastos
Popper	Carnap
Russel	Carnap
Sokrates	Carnap
	Schopenhauer
	Theophrastos

Abbildung 9.4: Word-Ausgabe eines MS-Access-Berichts, basierend auf Listing 9.1

9.2 PL/SQL

Das Oracle-Datenbanksystem bietet eine prozedurale Erweiterung von SQL an, genannt *PL/SQL*. Hiermit können SQL-Statements zu namenlosen Blöcken, 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);
```

Listing 9.1 zeigt einen namenlosen PL/SQL-Block, der 50 Konten mit durchlaufender Numerierung einrichtet und alle Kontostände mit 0 initialisiert.

```
declare
  i int;
begin
  for i in 1..50 loop
    insert into konto
    values (i, 0);
  end loop;
end;
```

Listing 9.1: Namenloser PL/SQL-Block

Listing 9.2 zeigt eine benannte PL/SQL-Prozedur, welche versucht, innerhalb der Tabelle *konto* eine Überweisung durchzuführen und danach das Ergebnis in zwei Tabellen

```
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);
```

in Form einer Erfolgs- bzw. Mißerfolgsmeldung festhält.

```
create or replace procedure ueberweisen     -- lege Prozedur an
( x int,                                    -- Konto-Nr. zum Belasten
  y int,                                    -- Konto-Nr. fuer Gutschrift
  betrag int)                               -- Ueberweisungsbetrag
IS
s konto.stand%TYPE;                         -- lokale Variable vom Typ konto.stand
begin
  SELECT stand INTO s FROM konto            -- hole Kontostand nach s
  WHERE nr = x FOR UPDATE OF stand;         -- von Konto-Nr. x und sperre Konto
  IF s < betrag THEN                        -- falls Konto ueberzogen wuerde
    INSERT INTO abgelehnt                   -- notiere den Fehlschlag
    VALUES (SYSDATE, x, y, betrag);         -- in der Tabelle abgelehnt
  ELSE
    UPDATE konto                            -- setze in der Tabelle konto
    SET stand = stand-betrag WHERE nr = x;  -- neuen Kontostand bei Konto x ein
    UPDATE konto                            -- setze in der Tabelle konto
    SET stand = stand+betrag WHERE nr = y;  -- neuen Kontostand bei Konto y ein
    INSERT INTO gebucht                     -- notiere die Ueberweisung
    VALUES (SYSDATE, x, y, betrag);         -- in der Tabelle gebucht
  END IF;
  COMMIT;
end;
```

Listing 9.2: PL/SQL-Prozedur

Im Gegensatz zu einem konventionellen Benutzerprogramm wird eine PL/SQL-Prozedur in der Datenbank gespeichert. Sie wird aufgerufen und (später) wieder entfernt durch

```
execute ueberweisung (42,37,50);
drop procedure ueberweisung;
```

In Listing 9.3 wird eine Funktion `f2c` definiert, die eine übergebene Zahl als Temperatur in Fahrenheit auffaßt und den Wert nach Celsius umrechnet.

```
create or replace function f2c                -- definiere eine Funktion f2c
  (fahrenheit IN number)                      -- Eingangsparameter vom Typ number
  return number                               -- Ausgangsparameter vom Typ number
is
  celsius number;                             -- lokale Variable
  begin                                       -- Beginn des Funktionsrumpfes
    celsius := (5.0/9.0)*(fahrenheit-32);     -- Umrechnung nach Celsius
    return celsius;                           -- Rueckgabe des Funktionswertes
  end f2c;                                    -- Ende der Funktion
```

Listing 9.3: PL/SQL-Funktion

Der Aufruf der Funktion erfolgt innerhalb einer SQL-Abfrage:

```
select nr, f2c(nr) from daten;
```

9.3 Embedded SQL

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

Oracle unterstützt Embedded SQL im Zusammenspiel mit den Programmiersprachen C und C++ durch den *Pre-Compiler Pro*C/C++*. Abbildung 9.5 zeigt den prinzipiellen Ablauf: Das mit eingebetteten SQL-Statements formulierte Host-Programm `hallo.pc` wird zunächst durch den Pre-Compiler unter Verwendung von SQL-spezifischen Include-Dateien in ein ANSI-C-Programm `hallo.c` überführt. Diese Datei übersetzt ein konventioneller C-Compiler unter Verwendung der üblichen C-Include-Files in ein Objekt-File `hallo.o`. Unter Verwendung der Oracle Runtime Library wird daraus das ausführbare Programm `hallo.exe` gebunden.

Eingebettete SQL-Statements werden durch ein vorangestelltes `EXEC SQL` gekennzeichnet und ähneln ansonsten ihrem interaktiven Gegenstück. Zum Beispiel werden durch

```
EXEC SQL COMMIT;
```

die seit Transaktionsbeginn durchgeführten Änderungen permanent gemacht.

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

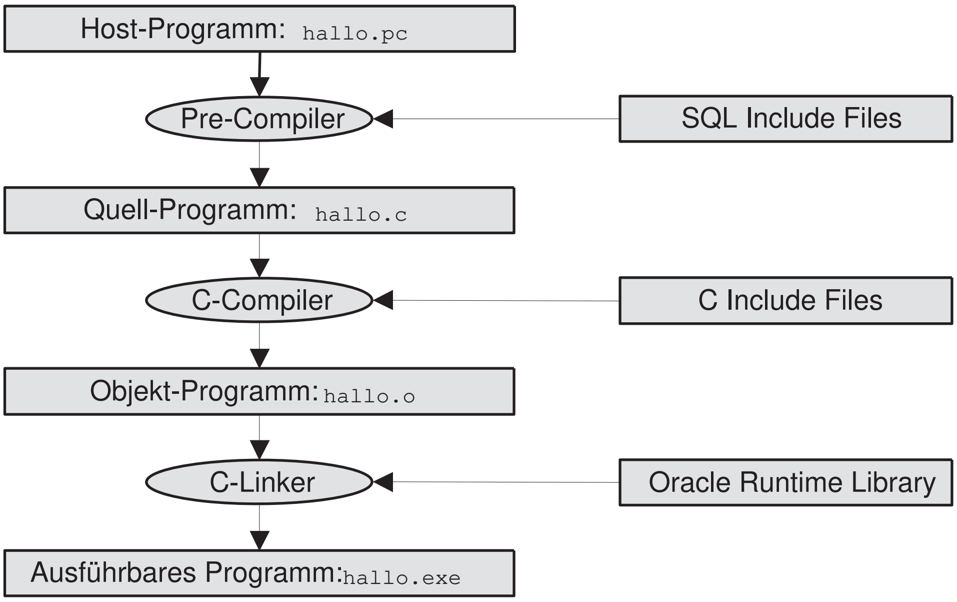


Abbildung 9.5: Vom Hostprogramm zum EXE-File

Datenbankwerte und Statusinformationen an das Host-Programm. Hostvariablen werden innerhalb von SQL-Statements durch einen vorangestellten Doppelpunkt (:) gekennzeichnet.

Für Hostvariablen, die Datenbankattributen vom Typ VARCHAR2 entsprechen, empfiehlt sich eine Definition nach folgendem Beispiel:

```
VARCHAR fachgebiet[20];
```

Daraus generiert der Pre-Compiler eine Struktur mit einem Character-Array und einer Längen-Komponente:

```
struct
{
    unsigned short len;
    unsigned char  arr[20];
} fachgebiet;
```

Bevor diese Struktur als Null-terminierter String genutzt werden kann, muß das Null-Byte explizit gesetzt werden:

```
fachgebiet.arr[fachgebiet.len] = '\0';
```

Mit einer Hostvariable kann eine optionale *Indikator-Variable* assoziiert sein, welche Null-Werte überträgt oder entdeckt. Folgende Zeilen definieren unter Verwendung des Oracle-spezifischen Datentyps `VARCHAR` eine Struktur `prof_rec` zum Aufnehmen eines Datensatzes der Tabelle *Professoren* sowie eine Indikator-Struktur `prof_ind` zum Aufnehmen von Status-Information.

```
struct{
  int     persnr;
  VARCHAR name [15];
  char    rang [2];
  int     raum;
} prof_rec;

struct {
  short persnr_ind;
  short name_ind;
  short rang_ind;
  short raum_ind;
} prof_ind;
```

Folgende Zeilen transferieren einen einzelnen Professoren-Datensatz in die Hostvariable `prof_rec` und überprüfen mit Hilfe der Indikator-Variable `prof_ind`, ob eine Raumangabe vorhanden war.

```
EXEC SQL SELECT PersNr, Name, Rang, Raum
         INTO   :prof_rec INDICATOR :prof_ind
         FROM   Professoren
         WHERE  PersNr = 2125;

if (prof_ind.raum_ind == -1)
   printf("Personalnummer %d hat keine Raumngabe \n", prof_rec.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 prof_cursor CURSOR FOR
  SELECT PersNr, Name, Rang, Raum
  FROM   Professoren
  WHERE  Rang='C4';

EXEC SQL OPEN prof_cursor;
EXEC SQL WHENEVER NOT FOUND DO break;

for (;;)
{
  EXEC SQL FETCH prof_cursor INTO :prof_rec;
  printf("Verarbeite Personalnummer %d\n", prof_rec.persnr);
}

EXEC SQL CLOSE prof_cursor;
```

Listing 9.4. zeigt ein Embedded-SQL-Programm, die davon erzeugte Ausgabe zeigt Abb. 9.6.

```
#include <stdio.h>                                  /* C-Header fuer I/O         */
#include <sqlca.h>                                  /* SQL-Communication Area    */

EXEC SQL BEGIN DECLARE SECTION;                     /* Beginn der Deklarationen  */

  char * username = "erika";                        /* Benutzername              */
  char * passwort = "mustermann";                   /* Benutzerpasswort          */
  char * dbdienst = "dbs99";                        /* Datenbankdienst           */


  struct{                                           /* Daten-Record              */
    int      persnr;                                /* Personalnummer            */
    VARCHAR  name[15];                              /* Name                      */
    char     rang[2];                               /* Rang                      */
    int      raum;                                  /* Raum                      */
  } prof_rec;

  struct {                                          /* Indikator-Record          */
    short persnr_ind;                               /* Indikator fuer PersNr     */
    short name_ind;                                 /* Indikator fuer Name       */
    short rang_ind;                                 /* Indikator fuer Rang       */
    short raum_ind;                                 /* Indikator fuer Raum       */
  } prof_ind;

  char eingaberang[3];                              /* lokale Variable           */

EXEC SQL END DECLARE SECTION;                       /* Ende der Deklarationen    */

void main()
{
    EXEC SQL WHENEVER SQLERROR GOTO fehler;         /* ggf. zur Fehlermarke      */
    EXEC SQL DECLARE dbname DATABASE;               /* deklariere Datenbankname  */

    EXEC SQL CONNECT :username                      /* mit Benutzername          */
    IDENTIFIED BY    :passwort                      /* mit Passwort              */
    AT dbname USING  :dbdienst;                     /* mit Datenbankdienst       */

    printf("Bitte Rang eingeben: ");                /* Aufforderung zur Eingabe  */
    scanf("%s", eingaberang);                       /* Einlesen der Eingabe      */
    printf("Mit Rang %s gespeichert:\n",eingaberang); /* Ausgabeankuendigung     */

    EXEC SQL at dbname DECLARE prof_cursor CURSOR FOR /* oeffne Cursor           */
    SELECT PersNr, Name, Rang, Raum                 /* fuer alle Attibute von    */
    FROM   oliver.Professoren                       /* oliver's Professoren      */
    WHERE  Rang = :eingaberang;                     /* mit vorgegebenem Rang     */

    EXEC SQL OPEN prof_cursor;                      /* oeffne den Cursor         */
    EXEC SQL WHENEVER NOT FOUND DO break;           /* falls keine Records:      */
                                                    /* hinter die Schleife       */
    for (;;)
    {
      EXEC SQL FETCH prof_cursor                    /* arbeite Cursor ab         */
               INTO :prof_rec INDICATOR :prof_ind;  /* incl. Indikatorvariable   */
```

```
      prof_rec.name.arr[prof_rec.name.len]='\0';        /* setze Nullbyte           */
      printf("%d %-10s %s",                             /* gib formatiert aus       */
             prof_rec.persnr,                           /* die Personalnummer       */
             prof_rec.name.arr,                         /* den Professorennamen     */
             prof_rec.rang);                            /* den Professorenrang      */
      if (prof_ind.raum_ind == -1) printf(" ???\n");    /* falls kein Null-Wert     */
                 else printf(" %d\n",prof_rec.raum);    /* gib Raumnummer aus       */
    }

    EXEC SQL CLOSE prof_cursor;                         /* schliesse den Cursor     */

    EXEC SQL at dbname COMMIT RELEASE; return;          /* beende Verbindung        */

    fehler:
       EXEC SQL WHENEVER SQLERROR CONTINUE;             /* im Falle eines Fehlers   */
       printf("Fehler: %70s\n",sqlca.sqlerrm.sqlerrmc); /* gib Fehlermeldung aus    */
       EXEC SQL at dbname ROLLBACK RELEASE;             /* Rollback und beenden     */
       return;
}
```

Listing 9.4: Quelltext von Embedded-SQL-Programm

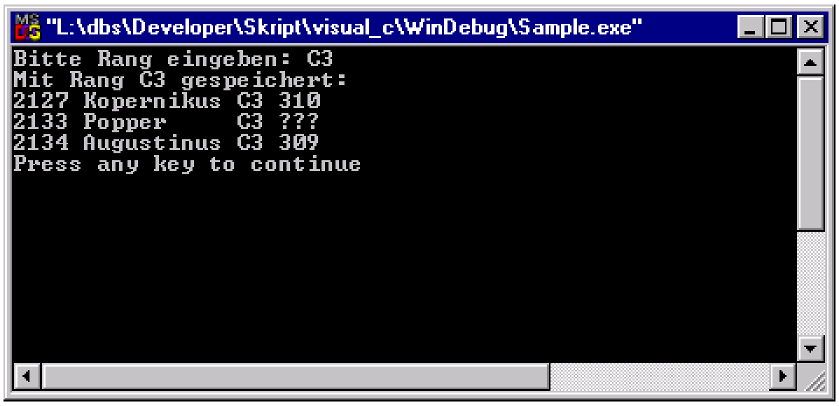


Abbildung 9.6: Ausgabe des Embedded-SQL-Programms von Listing 9.4

9.4 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.

JDBC ermöglicht drei Dinge:

1. eine Verbindung zur Datenbank aufzubauen,
2. SQL-Anweisungen abzusenden,
3. die Ergebnisse zu verarbeiten.

Der folgende Quelltext zeigt ein einfaches Beispiel für diese drei Schritte:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");        // Treiber laden
Connection con  = DriverManager.getConnection         // Verbindung herst.
                  ("jdbc:odbc:dbserv.dsn","erika","Mustermann");
Statement  stmt = con.createStatement();
ResultSet  rs   = stmt.executeQuery("select persnr, name from Professoren");
while (rs.next()){
  int x    = rs.getInt("persnr");
  String s = rs.getString("name");
  System.out.println("Professor "+s" hat die Personalnummer "+x);
}
```

Abbildung 9.7 zeigt die von Listing 9.5 erzeugte Ausgabe einer Java-Applikation auf der Konsole.

Abbildung 9.7: Ausgabe einer Java-Applikation

```
/* ShowJdbc.java (c) Stefan Rauch 1999 */

import java.sql.*;                              // Import der SQL-Klassen

public class ShowJdbc {

  public static void main(String args[]) {

    String url = "jdbc:odbc:dbserv.dsn";         // Treiber-url f. Verbindung
    Connection con;                             // Verbindungs-Objekt
    Statement stmt;                             // Instanz der Verbindung
                                                // sendet query und liefert
                                                // Ergebnis (ResultSet)
    String user   = "Erika";
    String passwd = "Mustermann";
    String query  = "select * from Professoren";

    try {                                       // Laden des jdbc-odbc-Brueckentreiber
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }catch(java.lang.ClassNotFoundException e) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(e.getMessage());
    }

    try {
      con = DriverManager.getConnection(url, user, passwd); // Erst. der Verbindung
      stmt = con.createStatement();             // Instantiierung des Statement
      ResultSet rs = stmt.executeQuery(query);  // Ergebnis in ResultSet

      System.out.println("Ausgabe der Professoren mit jeweiligem Rang: \n");

      while(rs.next()) {                        // Zeilenweise durch
        System.out.print("Der Professor ");     // Ergebnismenge laufen
        System.out.print(rs.getString("Name")); // dabei Namen und Rang
        System.out.print(" hat den Rang ");     // formatiert ausgeben
        System.out.println(rs.getString("Rang"));
      }
      stmt.close();                             // Schliessen des Statements
      con.close();                              // Schliessen der Verbindung

    }catch(SQLException ex) {
      System.err.println("SQLException: " + ex.getMessage());
    }
  }
}
```

Listing 9.5: Quelltext der Java-Applikation ShowJdbc.java

Listing 9.6 zeigt den Quelltext einer HTML-Seite mit dem Aufruf eines Java-Applets. Abbildung 9.8 zeigt die Oberfläche des Applets. Listing 9.7 zeigt den Quelltext des Applets.

```html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<!--last modified on Thursday, June 03, 1999 11:37 PM -->
<HTML>
  <HEAD>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html;CHARSET=iso-8859-1">
    <META NAME="Author" Content="Stefan Rauch">
    <TITLE>JDBC Test-Applet </TITLE>
  </HEAD>
  <BODY BGCOLOR="#d3e2cf">
  <P ALIGN="CENTER"><FONT SIZE="5">
  <B>Demo-Applet f&uuml;r JDBC-Datenbankzugriff </B></FONT></P>
  <CENTER>
  <P> <FONT SIZE="4">
    <B>SQL-Abfrage an dbserv als user Erika mit Passwort Mustermann</B> </FONT>
  </P>
  <P><APPLET CODEBASE="../skript/Applets" CODE="JdbcApplet" WIDTH="700" HEIGHT="400" ALIGN="BOTTOM">
  </APPLET>
  </CENTER>
  </BODY>
</HTML>
```

Listing 9.6: Quelltext einer HTML-Seite zum Aufruf eines Applets

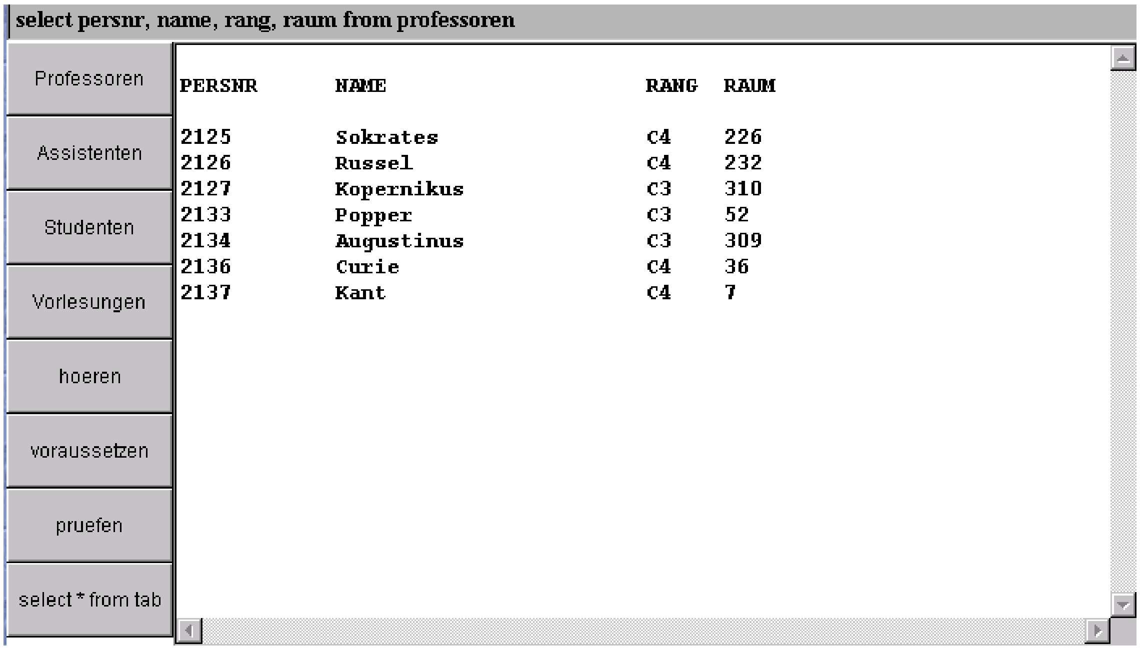


Abbildung 9.8: Java-Applet mit JDBC-Zugriff auf Oracle-Datenbank

```
/* JdbcApplet (c) Stefan Rauch 1999          */
/* Applet, das den Umgang mit dem JDBC zeigt */


import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.sql.*;


public class JdbcApplet extends Applet {

  BorderLayout layout = new BorderLayout();
  TextArea outputArea = new TextArea();
  TextField inputField = new TextField();
  Panel p;

/* Erstellen der Buttons, die den Inhalt der benannten Tabellen komplett ausgeben */
  Button qu1 = new Button("Professoren");
  Button qu2 = new Button("Assistenten");
  Button qu3 = new Button("Studenten");
  Button qu4 = new Button("Vorlesungen");
  Button qu5 = new Button("hoeren");
  Button qu6 = new Button("voraussetzen");
  Button qu7 = new Button("pruefen");
  Button qu8 = new Button("select * from tab");

/* Verbindungsobjekt um die Verbindung zum DBMS aufzubauen */
  Connection con;

/* Statement-Objekt zur Kommunikation mit DBMS */
  Statement stmt;


  public JdbcApplet() {
  }

/* Das Applet initialisieren */

  public void init() {
    try {
      this.setLayout(layout);
      this.setSize(700,400);
      inputField.setBackground(Color.gray);
      inputField.setFont(new Font("Serif",1,14));

/* ActionListener fuer das Eingabefeld         */
/* gibt den Text an Methode execQuery() weiter */
      inputField.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          String q = inputField.getText();
          execQuery(q);
        }
```

```
      });
      outputArea.setBackground(Color.white);
      outputArea.setEditable(false);
      outputArea.setFont(new Font("Monospaced",1,14));

/* ActionListener fuer jeweiligen Button      */
/* gibt Abfrage an Methode execQuery() weiter */
      qu1.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select persnr,name,rang,raum," +
                    "to_char(gebdatum,'dd:mm:YYYY') as Geburtsdatum from Professoren");
        }
      });
      qu2.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select persnr, name, fachgebiet, boss," +
                    "to_char(gebdatum,'dd:mm:YYYY') as Geburtsdatum from Assistenten");
        }
      });
      qu3.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select matrnr, name, semester," +
                    "to_char(gebdatum,'dd:mm:YYYY') as Geburtsdatum from Studenten");
        }
      });
      qu4.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select * from Vorlesungen");
        }
      });
      qu5.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select * from hoeren");
        }
      });
      qu6.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select * from voraussetzen");
        }
      });
      qu7.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select * from pruefen");
        }
      });
      qu8.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
          execQuery("select * from tab");
        }
      });

/* Hinzufuegen und Anordnen der einzelnen Elemente des Applets */
      this.add(outputArea, BorderLayout.CENTER);
```

```
      this.add(inputField, BorderLayout.NORTH);
      this.add(p= new Panel(new GridLayout(8,1)), BorderLayout.WEST);
      p.add(qu1);
      p.add(qu2);
      p.add(qu3);
      p.add(qu4);
      p.add(qu5);
      p.add(qu6);
      p.add(qu7);
      p.add(qu8);
    }
    catch (Exception e) {
      e.printStackTrace();
    }

/* Verbindungsaufbau mit dem DBMS */
    String url    = "jdbc:odbc:dbserv.dsn";
    String user   = "Erika";
    String passwd = "Mustermann";

/* JDBC-ODBC Brueckentreiber wird genutzt und vom DriverManager registriert */
    try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    }
    catch(java.lang.ClassNotFoundException ex) {
      System.err.print("ClassNotFoundException: ");
      System.err.println(ex.getMessage());
    }

/* Verbindung zum DBMS wird geoeffnet */
    try {
      con = DriverManager.getConnection(url, user, passwd);
    }
    catch(SQLException ex) {
      outputArea.setText("SQLException: " + ex.getMessage());
    }
  }


/* Methode, die die jeweiligen SQL-Querys an das DBMS weiterleitet und die */
/* in einem ResultSet empfangenen Ergebnisse in der TextArea darstellt     */

  void execQuery(String query) {

    try {

/* Fuer jede Abfrage muss ein Statement-Objekt instantiiert werden */
      stmt = con.createStatement();

/* Ausfuehren der Abfrage und Speichern der Ergebnisse im ResultSet rs */
      ResultSet rs = stmt.executeQuery(query);

/* z := Anzahl der Spalten des Ergebnisses */
```

```
      int z = rs.getMetaData().getColumnCount();
      outputArea.setText("\n");
      outputArea.setVisible(false);

/* Fuer jede Spalte wird zunaechst der Spaltenname formatiert ausgegeben */
      for (int i=1;i<=z;i++) {
        String lab=rs.getMetaData().getColumnLabel(i);
        outputArea.append(lab);
        int y = rs.getMetaData().getColumnDisplaySize(i)+4;
        for (int j=0;j<(y-lab.length());j++)
          outputArea.append(" ");
        }
        outputArea.append("\n");

/* Der Inhalt der Ergebnismenge wird zeilenweise formatiert in der TextArea ausgegeben */
        String arg;
        while(rs.next()) {
          outputArea.append("\n");
          for (int i=1;i<=z;i++) {
            arg=rs.getString(i);
            int len;
            if (arg != null) {
              len=arg.length();
              outputArea.append(arg);
            }
            else {
              len=4;
              outputArea.append("null");
            }
            int y = rs.getMetaData().getColumnDisplaySize(i)+4;
            for (int j=0;j<(y-len);j++)
              outputArea.append(" ");
          }
        }

        outputArea.setVisible(true);
        stmt.close();

/* Abfangen von etwaigen SQL-Fehlermeldungen und Ausgabe derer in der TextArea */
      }catch(SQLException ex) {
        outputArea.setText("SQLException: " + ex.getMessage());
      }

  }

}
```

Listing 9.7: Quelltext von Java-Applet

9.5 Cold Fusion

Cold Fusion ist ein Anwendungsentwicklungssystem der Firma Allaire für dynamische Web-Seiten. Eine ColdFusion-Anwendung besteht aus einer Sammlung von CFML-Seiten, die in der *Cold Fusion Markup Language* geschrieben sind. Die Syntax von CFML ist an HTML angelehnt und beschreibt die Anwendungslogik. In Abbilung 9-9 ist der grundsätzliche Ablauf dargestellt:

1. Wenn ein Benutzer eine Seite in einer Cold Fusion - Anwendung anfordert, sendet der Web-Browser des Benutzers eine HTTP-Anforderung an den Web-Server.
2. Der Web-Server übergibt die vom Clienten übermittelten Daten an den Cold Fusion Application Server.
3. Cold Fusion liest die Daten vom Clienten und verarbeitet den auf der Seite verwendeten CFML-Code und führt die damit angeforderte Anwendungslogik aus.
4. Cold Fusion erzeugt dynamisch eine HTML-Seite und gibt sie an den Web-Server zurück.
5. Der Web-Server gibt die Seite an den Web-Browser zurück.

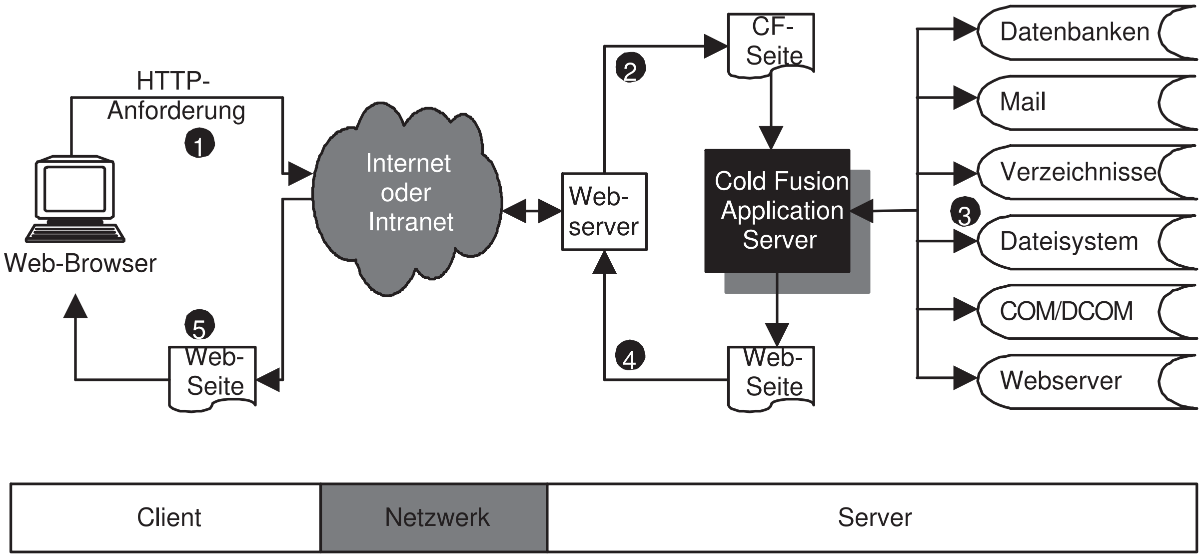


Abbildung 9.9: Arbeitsweise von Coldfusion

Von den zahlreichen Servertechnologien, mit denen Cold Fusion zusammenarbeiten kann, interessiert uns hier nur die Anbindung per ODBC an eine relationale Datenbank.

CF-Vorlesungsverzeichnis: `http://iuk-www.vdv.uni-osnabrueck.de/vpv/sommer99/index.cfm`

CF-Online-Dokumentation: `http://cfserv.rz.uni-osnabrueck.de/cfdocs` .

Listing 9.8 zeigt eine unformatierte Ausgabe einer SQL-Query.

```
<CFQUERY NAME       = "Studentenliste"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">
  SELECT matrnr, name from studenten
</CFQUERY>

<HTML>
  <HEAD>
    <TITLE> Studentenliste </TITLE>
  </HEAD>
  <BODY>
    <H2> Studentenliste (unformatiert)</H2>
    <CFOUTPUT QUERY="Studentenliste">
      #name# #matrnr# <BR>
    </CFOUTPUT>
  </BODY>
</HTML>
```

Listing 9.8: Quelltext von studliste.cfm

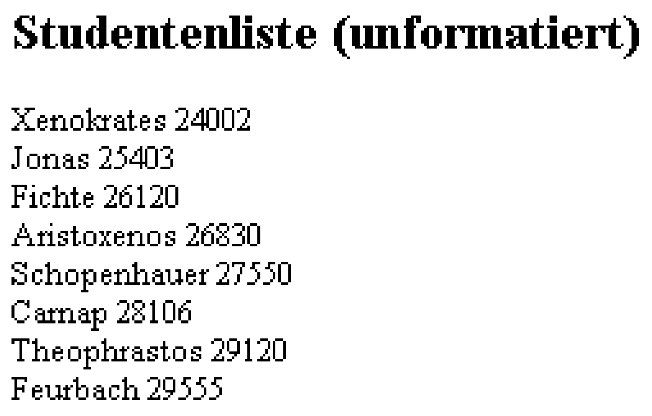


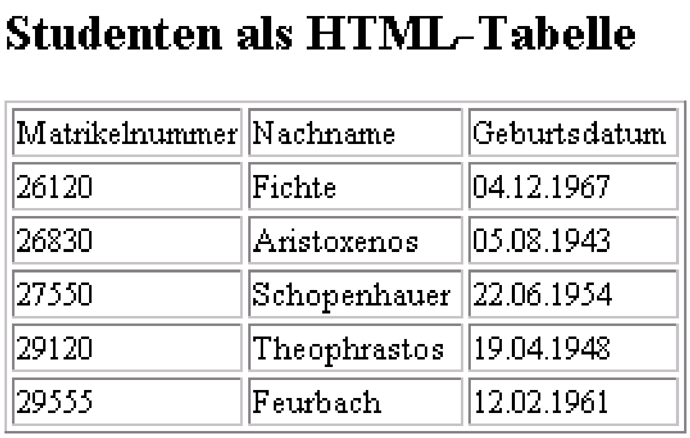
Abbildung 9.10: Screenshot von studliste.cfm

Listing 9.9 zeigt die formatierte Ausgabe einer SQL-Query unter Verwendung einer HTML-Tabelle.

```
<CFQUERY NAME       = "Studententabelle"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">
  SELECT matrnr, name, to_char(gebdatum,'DD.MM.YYYY') as geburt from studenten
  WHERE  (sysdate-gebdatum)/365 > 30
</CFQUERY>

<HTML>
  <HEAD>
    <TITLE> Studententabelle </TITLE>
  </HEAD>
  <BODY>
    <H2> Studenten als HTML-Tabelle</H2>
    <TABLE BORDER>
    <TD>Matrikelnummer</TD> <TD> Nachname </TD> <TD>Geburtsdatum </TD></TR>
    <CFOUTPUT QUERY="Studententabelle">
      <TR><TD> #matrnr# </TD> <TD> #name# </TD> <TD> #geburt# </TR>
    </CFOUTPUT>
    </TABLE>
  </BODY>
</HTML>
```

Listing 9.9: Quelltext von studtabelle.cfm



Studenten als HTML-Tabelle

Matrikelnummer	Nachname	Geburtsdatum
26120	Fichte	04.12.1967
26830	Aristoxenos	05.08.1943
27550	Schopenhauer	22.06.1954
29120	Theophrastos	19.04.1948
29555	Feurbach	12.02.1961

Abbildung 9.11: Screenshot von studtabelle.cfm

Listing 9.10 zeigt die Verwendung eines Formulars zum Eingeben eines Dozentennamens, der eine Suche anstößt.

```
<HTML>
  <HEAD>
    <TITLE> Studentenformular </TITLE>
  </HEAD>
  <BODY>
    <FORM ACTION="studsuche.cfm" METHOD="POST">
    Sie k&ouml;nnen nach den H&ouml;rern eines Dozenten suchen: <BR>
    Professorenname: <INPUT TYPE="text"   NAME="Profname">
    <INPUT TYPE="Submit" VALUE="Suchen">
    </FORM>
  </BODY>
</HTML>
```

Listing 9.10: Quelltext von studformular.cfm

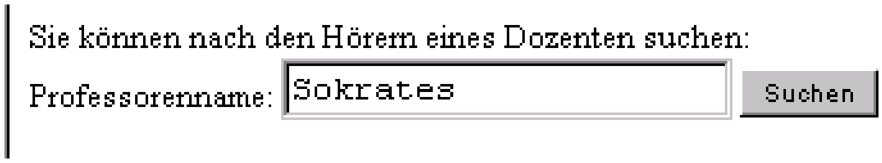


Abbildung 9.12: Screenshot von studformular.cfm

Der vom Formular `studformular.cfm` erfaßte Name wird übergeben an die Datei `studsuche.cfm`, welche im Listing 9.11 gezeigt wird.

```
<CFQUERY NAME       = "Studentensuche"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC" >
  SELECT unique s.matrnr, s.name
  FROM   professoren p, vorlesungen v, hoeren h, studenten s
  WHERE  s.matrnr     = h.matrnr
  AND    h.vorlnr     = v.vorlnr
  AND    v.gelesenvon = p.persnr
  AND    p.name       = '#FORM.Profname#'
</CFQUERY>

<HTML>
  <HEAD>
    <TITLE> Studenten eines Professors </TITLE>
  </HEAD>
  <BODY>
    <CFOUTPUT>
      Professor #FORM.Profname# hat folgende H&ouml;rer: <P>
    </CFOUTPUT>
    <CFOUTPUT QUERY="Studentensuche">
      #matrnr# #name#<BR>
    </CFOUTPUT>
  </BODY>
</HTML>
```

Listing 9.11: Quelltext von studsuche.cfm

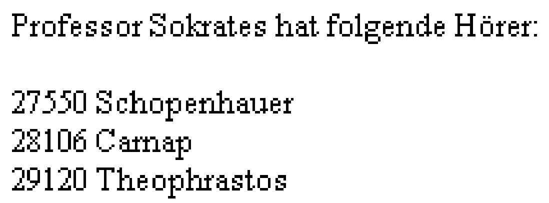


Abbildung 9.13: Screenshot von studsuche.cfm

Listing 9.12 zeigt eine HTML-Tabelle mit sensitiven Links für die Professoren.

```
<CFQUERY NAME       = "Vorlesungstabelle"
         USERNAME   = "erika"      PASSWORD = "mustermann"
         DATASOURCE = "dbserv.dsn" DBTYPE   = "ODBC">
  SELECT vorlnr, titel, name, persnr FROM vorlesungen, professoren
  where gelesenvon = persnr
</CFQUERY>
<HTML>
  <HEAD> <TITLE> Vorlesungstabelle </TITLE> </HEAD>
  <BODY>
    <H2> Vorlesungen mit sensitiven Links </H2>
    <TABLE BORDER>
    <TD>Vorlesungsnr</TD> <TD> Titel </TD> <TD>Dozent</TD> </TR>
    <CFOUTPUT QUERY="Vorlesungstabelle">
      <TR><TD>#vorlnr#</TD><TD>#Titel#</TD>
      <TD><A HREF="profinfo.cfm?profid=#persnr#">#name#</A></TD></TR>
    </CFOUTPUT>
    </TABLE>
  </BODY>
</HTML>
```

Listing 9.12: Quelltext von vorltabelle.cfm

Vorlesungen mit sensitiven Links

Vorlesungsnr	Titel	Dozent
5001	Grundzuge	Kant
5041	Ethik	Sokrates
5043	Erkenntnistheorie	Russel
5049	Maeutik	Sokrates
4052	Logik	Sokrates
5052	Wissenschaftstheorie	Russel
5216	Bioethik	Russel
5259	Der Wiener Kreis	Popper
5022	Glaube und Wissen	Augustinus
4630	Die 3 Kritiken	Kant

Abbildung 9.14: Screenshot von vorltabelle.cfm

Die in Listing 9.12 ermittelte Personalnummer eines Professors wird in Form einer URL an die in Listing 9.13 gezeigte Datei `profinfo.cfm` übergeben und dort in einer Select-Anweisung verwendet. Die gefundenen Angaben zum Dozenten werden anschließend ausgegeben.

```
<CFQUERY NAME       = "Profinfo"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">
  SELECT * from Professoren
  WHERE persnr=#URL.profid#
</CFQUERY>

<HTML>
  <HEAD>
    <TITLE> Professoreninfo: </TITLE>
  </HEAD>
  <BODY>
   <H2> Professoren-Info</H2>
    <CFOUTPUT QUERY="Profinfo">
      Professor #name# hat die Personalnummer #persnr#. <BR>
      Er wird nach Rang #rang# besoldet. <BR>
      Sein Dienstzimmer ist Raum #Raum#.
    </CFOUTPUT>
    </TABLE>
  </BODY>
</HTML>
```

Listing 9.13: Quelltext von profinfo.cfm

Professoren-Info

Professor Sokrates hat die Personalnummer 2125.
Er wird nach Rang C4 besoldet.
Sein Dienstzimmer ist Raum 226.

Abbildung 9.15: Screenshot von profinfo.cfm

Listing 9.14 zeigt ein Formular zum Einfügen eines Professors.

```
<HTML>
  <HEAD> <TITLE> Professoreneinf&uuml;geformular </TITLE> </HEAD>
  <BODY>
    <H2> Professoreneinf&uuml;geformular </H2>
    <FORM ACTION="profinsert.cfm" METHOD="POST"> <PRE>
    PersNr:   <INPUT SIZE= 4 TYPE="text" NAME="ProfPersnr">
              <INPUT TYPE="hidden" NAME="ProfPersnr_required"
              VALUE="PersNr erforderlich !">
              <INPUT TYPE="hidden" NAME="ProfPersnr_integer"
              VALUE="Personalnummer muss ganzzahlig sein !">
    Name:     <INPUT SIZE=15 TYPE="text"       NAME="ProfName">
              <INPUT TYPE="hidden" NAME="ProfName_required"
              VALUE="Name erforderlich !">
    Rang:     <SELECT NAME="ProfRang"> <OPTION>C2 <OPTION>C3 <OPTION>C4 </SELECT>
    Raum:     <INPUT SIZE=4 TYPE="text" NAME="ProfRaum">
              <INPUT TYPE="Submit" VALUE="Einf&uuml;gen">
    </PRE></FORM>
  </BODY>
</HTML>
```

Listing 9.14: Quelltext von profinsertform.cfm

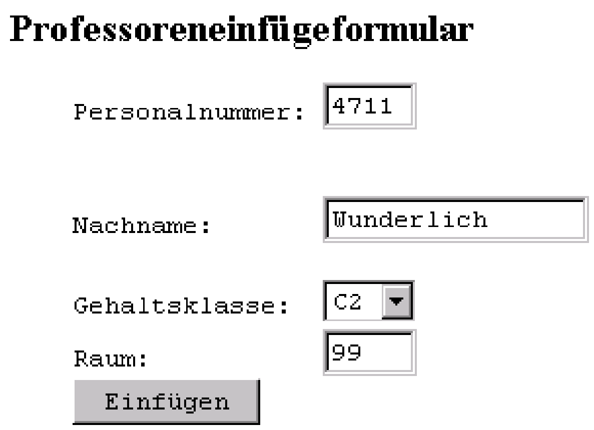


Abbildung 9.16: Screenshot von profinsertform.cfm

Die von Listing 9.14 übermittelten Daten werden in Listing 9.15 zum Einfügen verwendet. Anschließend erfolgt eine Bestätigung.

```
<CFQUERY NAME       = "Profinsert"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">

  INSERT INTO Professoren (PersNr, Name, Rang, Raum)
  VALUES ('#FORM.ProfPersnr#','#FORM.ProfName#','#FORM.ProfRang#','#FORM.ProfRaum#')

</CFQUERY>

<HTML>

  <HEAD>
    <TITLE> Professoreneinf&uuml;gen </TITLE>
  </HEAD>

  <BODY>

    In die Tabelle der Professoren wurde eingef&uuml;gt: <P>
    <CFOUTPUT>
       <PRE>
       Persnr: #FORM.ProfPersnr#
       Name:   #FORM.ProfName#
       Rang:   #FORM.ProfRang#
       Raum:   #ProfRaum#
       </PRE>
    </CFOUTPUT>

  </BODY>

</HTML>
```

Listing 9.15: Quelltext von profinsert.cfm

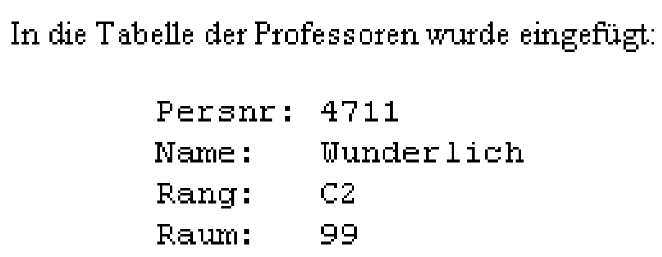


Abbildung 9.17: Screenshot von profinsert.cfm

Listing 9.16 zeigt eine Tabelle mit einer Form zum Löschen eines Professors.

```
<CFQUERY NAME       = "Professorentabelle"
         USERNAME   = "erika" PASSWORD    = "mustermann"
         DATASOURCE = "dbserv.dsn" DBTYPE = "ODBC">
  SELECT * from professoren
</CFQUERY>
<HTML>
  <HEAD>
    <TITLE> Professorenformular zum L&oum;schen </TITLE>
  </HEAD>
  <BODY>
    <H2> Professorenformular zum L&ouml;schen</H2>
    <TABLE BORDER>
    <TD>PersNr</TD><TD>Name</TD><TD>Rang</TD><TD>Raum</TD></TR>
    <CFOUTPUT QUERY="Professorentabelle">
    <TR><TD>#persnr#</TD><TD>#name#</TD><TD>#rang#</TD><TD>#raum#</TD></TR>
    </CFOUTPUT>
    </TABLE>
    <FORM ACTION="profdelete.cfm" METHOD="POST">
    Personalnummer: <INPUT SIZE=4  TYPE="text" NAME="Persnr">
    <INPUT TYPE="Submit" VALUE="Datensatz l&ouml;schen">
    </FORM>
  </BODY>
</HTML>
```

Listing 9.16: Quelltext von profdeleteform.cfm

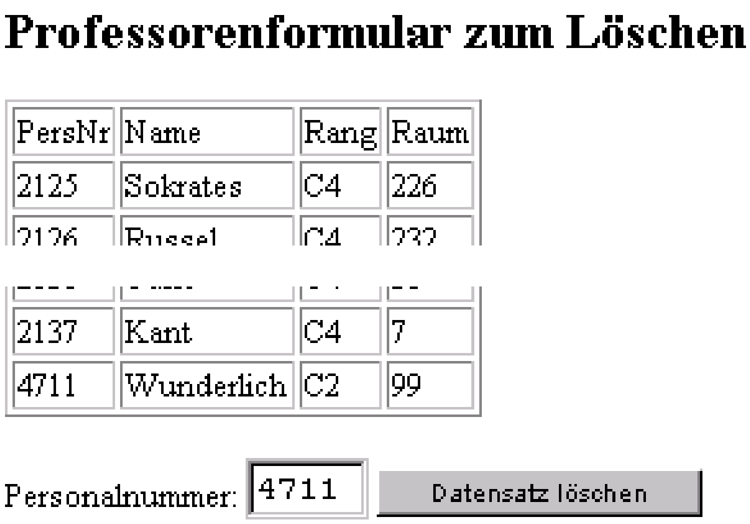


Abbildung 9.18: Screenshot von profdeleteform.cfm

Die in Listing 9.16 ermittelte Personalnummer eines Professors wird in Listing 9.17 zum Löschen verwendet. Anschließend erfolgt eine Bestätigung.

```
<CFQUERY NAME       = "Profdelete"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">
  delete from professoren
  where persnr = #FORM.persnr#
</CFQUERY>

<HTML>
  <HEAD> <TITLE> Professoren l&ouml;schen </TITLE> </HEAD>
  <BODY>
    <CFOUTPUT>
      Der Professor mit Personalnummer #FORM.persnr# wurde gel&ouml;scht
    </CFOUTPUT>
  </BODY>
</HTML>
```

Listing 9.17: Quelltext von profdelete.cfm

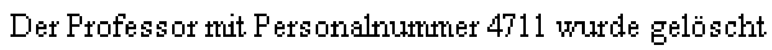


Abbildung 9.19: Screenshot von profdelete.cfm

Listing 9.18 zeigt ein Formular zum Suchen nach einem Professorendatensatz unter Verwendung des Wildcard-Zeichens %.

```
<HTML>
  <HEAD>
    <TITLE> Professorenupdate </TITLE>
  </HEAD>

  <BODY>
    Bitte geben Sie Suchkriterien ein,
    um gezielt einen Professor zum UPDATE zu suchen.<BR>
    <P>
    <FORM ACTION="profupdate.cfm" METHOD="POST">
    <TABLE>
    <TR><TD>Personalnummer:</TD>
             <TD><INPUT TYPE="text" SIZE=4 NAME="ProfPersnr">
                 <INPUT TYPE="HIDDEN"NAME="ProfPersnr_integer"
                      VALUE="Personalnummer muss ganzzahlig sein"></TD></TR>
    <TR><TD> Nachname:</TD>
             <TD><INPUT SIZE=15 TYPE="text" NAME="ProfName">
                      Wildcard <B>%</B> kann genutzt werden.</TD></TR>
    <TR><TD> Gehaltsklasse:</TD>
             <TD><SELECT NAME="ProfRang">
                      <OPTION>
                      <OPTION>C2
                      <OPTION>C3
                      <OPTION>C4
             </SELECT></TD></TR>
    <TR><TD> Raum:</TD>
             <TD><INPUT SIZE=4 TYPE="text" NAME="ProfRaum">
                 <INPUT TYPE="HIDDEN" NAME="ProfRaum_integer"
                        VALUE="Die Raumnummer muss ganzzahlig sein"></TD></TR>

             <!--- Hiddenfield zur spaeteren Steuerung --->
             <INPUT TYPE="HIDDEN" NAME="i" VALUE="1">
    <TR><TD><INPUT TYPE="Submit" VALUE="Prof suchen"></TD><TD></TD></TR>
    </TABLE>
    </FORM>
  </BODY>
</HTML>
```

Listing 9.18: Quelltext von profupdateformular.cfm

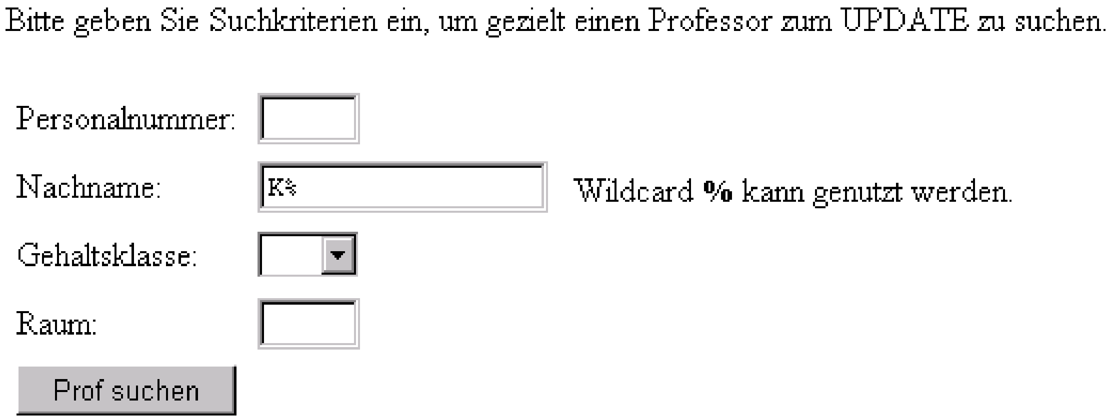


Abbildung 9.20: Screenshot von profupdateformular.cfm

Die in Listing 9.18 gefundenen Treffer können im Listing 9.19 durchlaufen werden und anschließend editiert werden.

```
<!--- erstellt von Ralf Kunze --->

<CFQUERY NAME       = "ProfAbfr"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">

        <!--- Where 0=0, um in jedem Fall eine
        korrekte Abfrage zu erhalten --->
  SELECT * FROM professoren where 0 = 0

        <!--- Weitere Statements gegebenenfalls anhaengen --->
<CFIF #ProfPersnr# is NOT "">
AND PersNr = #ProfPersnr#
</CFIF>

<CFIF #ProfName# is not "">
AND Name LIKE '#ProfName#'
</CFIF>

<CFIF #ProfRang# is not "">
AND Rang = '#ProfRang#'
</CFIF>

<CFIF #ProfRaum# is not "">
AND Raum = '#ProfRaum#'
</CFIF>

</CFQUERY>
```

```
<HTML>

  <HEAD>
    <TITLE> Professorenupdate </TITLE>
  </HEAD>

  <BODY>

    <!--- Falls keine Ergebnisse erzielt wurden, Fehlermeldung geben
          und den Rest der Seite mit CFABORT unterdruecken --->
    <CFIF #ProfAbfr.Recordcount# IS "0">
          Ihre Anfrage lieferte leider keine passenden Records.<BR>
          <A HREF="profupdateformular.cfm">New Search</A>
          <CFABORT>
    </CFIF>
    Bitte geben sie die gew&uuml;nschte &Auml;nderung ein
    bzw. w&auml;hlen sie den entsprechenden Datensatz aus:

    <!--- Ausgabe der Ergebnisse. Bei Record #i# starten
          und nur ein Record liefern --->
    <CFOUTPUT QUERY="ProfAbfr" STARTROW="#i#" MAXROWS="1">
    <FORM ACTION="update.cfm" METHOD="POST">

    <!--- Ausgabe der Werte in ein Formular zum aendern --->
    <TABLE>
      <TR><TD>Personalnummer: </TD>
          <TD><INPUT TYPE="text" SIZE=4 NAME="ProfPersnr" VALUE="#Persnr#">
              <INPUT TYPE="HIDDEN" NAME="ProfPersnr_integer"
                     VALUE="Personalnummer muss ganzzahlig sein"></TD></TR>
      <TR><TD>Nachname:</TD>
          <TD><INPUT SIZE=15 TYPE="text" NAME="ProfName"
                     VALUE="#Name#"></TD></TR>
      <TR><TD>Gehaltsklasse:</TD>
          <TD><SELECT NAME="ProfRang">
                <CFIF #Rang# IS "C2"><OPTION SELECTED><CFELSE><OPTION></CFIF>C2
                <CFIF #Rang# IS "C3"><OPTION SELECTED><CFELSE><OPTION></CFIF>C3
                <CFIF #Rang# IS "C4"><OPTION SELECTED><CFELSE><OPTION></CFIF>C4
      </SELECT></TD></TR>
      <TR><TD> Raum:</TD>
          <TD><INPUT SIZE=4 TYPE="text" NAME="ProfRaum" VALUE="#Raum#">
              <INPUT TYPE="HIDDEN" NAME="ProfRaum_integer"
                     VALUE="Raumnummer muss ganzzahlige sein"></TD></TR>
      <TR><TD><INPUT TYPE="Submit" VALUE="Update"></TD>
          <TD><INPUT TYPE="RESET"></TD></TR>
    </TABLE>
    </FORM>
    </CFOUTPUT>

    <!--- Den Zaehler setzen und entsprechend des
      Wertes weiteren Link anbieten oder nicht --->
    <CFIF #i# IS "1">
      <IMG SRC="Grayleft.gif" ALT="Back">
    <CFELSE>
```

```
      <CFSET iback=#i#-1>
      <CFOUTPUT>
        <A HREF="profupdate.cfm?i=#iback#&ProfPersnr=#ProfPersnr#&Profname=#Profname#
                                             &ProfRang=#ProfRang#&ProfRaum=#ProfRaum#">
        <IMG SRC="redleft.gif" BORDER="0" ALT="back"></A>
      </CFOUTPUT>
    </CFIF>
    <A HREF="profupdateformular.cfm">New Search</A>
    <CFIF #i# LESS THAN #ProfAbfr.RecordCount#>
      <CFSET inext=#i#+1>
      <CFOUTPUT>
        <A HREF="profupdate.cfm?i=#inext#&ProfPersnr=#ProfPersnr#&Profname=#Profname#
                                             &ProfRang=#ProfRang#&ProfRaum=#ProfRaum#">
        <IMG SRC="redright.gif" ALIGN="Next Entry" BORDER="0"></A>
      </CFOUTPUT>
    <CFELSE>
      <IMG SRC="grayright.gif" ALT="Next">
    </CFIF>

    <!--- Ausgabe welcher Datensatz gezeigt wird
          und wieviele insgesamt vorhanden sind --->
    <CFOUTPUT>Eintrag #i# von #ProfAbfr.RecordCount#</CFOUTPUT><BR>
  </BODY>
</HTML>
```

Listing 9.19: Quelltext von profupdate.cfm

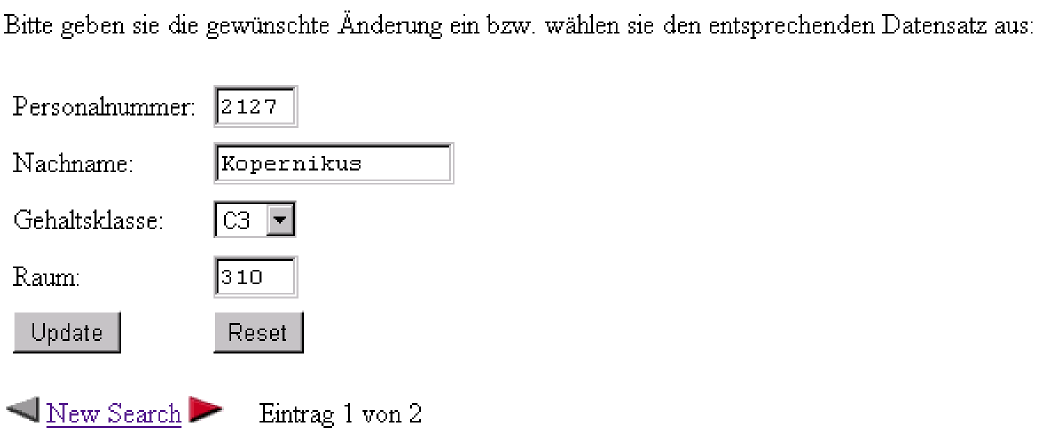


Abbildung 9.21: Screenshot von profupdate.cfm

Listing 9.20 zeigt die Durchführung der Update-Operation auf dem in Listing 9.19 ausgewählten Professorendatensatz.

```
<!--- erstellt von Ralf Kunze --->
<CFQUERY NAME       = "Profupdate"
         USERNAME   = "erika"
         PASSWORD   = "mustermann"
         DATASOURCE = "dbserv.dsn"
         DBTYPE     = "ODBC">
  UPDATE professoren set
  name ='#FORM.ProfName#',
  rang ='#FORM.ProfRang#',
  raum ='#FORM.ProfRaum#'
  where persnr = #FORM.ProfPersnr#
</CFQUERY>

<HTML>
  <HEAD>
    <TITLE> Professorenupdate </TITLE>
  </HEAD>
  <BODY>
    In der Tabelle der Professoren wurde ein Datensatz modifiziert:
    <CFOUTPUT>
      <PRE>
       Persnr: #FORM.ProfPersnr#
       Name:   #FORM.ProfName#
       Rang:   #FORM.ProfRang#
       Raum:   #Form.ProfRaum#
      </PRE>
    </CFOUTPUT>
    <A HREF="profupdateformular.cfm">New Search</A>
  </BODY>
</HTML>
```

Listing 9.20: Quelltext von update.cfm

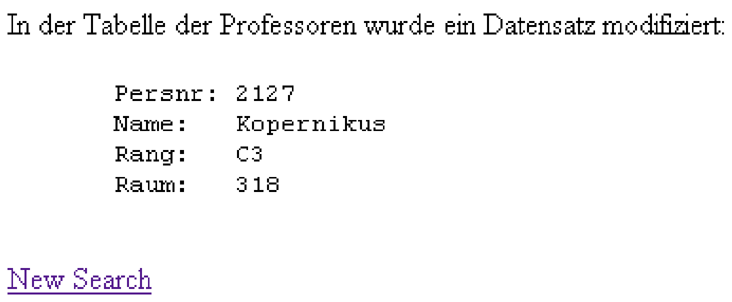


Abbildung 9.22: Screenshot von update.cfm

Kapitel 10

Relationale Entwurfstheorie

10.1 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, läßt 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.

10.2 Schlüssel

In dem Relationenschema $\mathcal{R}$ ist $\alpha \subseteq \mathcal{R}$ ein *Superschlüssel*, falls gilt

$$\alpha \rightarrow \mathcal{R}$$

Der Begriff Superschlüssel besagt, daß 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}

10.3 Bestimmung funktionaler Abhängigkeiten

Wir betrachten folgendes Relationenschema:

ProfessorenAdr : {[PersNr, Name, Rang, Raum,
Ort, Straße, 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 Straße, Städte und Straßen gehen nicht über Bundesgrenzen hinweg.

Folgende Abhängigkeiten gelten:

1.	{PersNr}	$\rightarrow$	{PersNr, Name, Rang, Raum, Ort, Straße, PLZ, Vorwahl, BLand, EW, Landesregierung}
2.	{Ort, BLand}	$\rightarrow$	{Vorwahl}
3.	{PLZ}	$\rightarrow$	{BLand, Ort}
4.	{Ort, BLand, Straße}	$\rightarrow$	{PLZ}
5.	{BLand}	$\rightarrow$	{Landesregierung}
6.	{Raum}	$\rightarrow$	{PersNr}

Hieraus können weitere Abhängigkeiten abgeleitet werden:

7.	{Raum}	$\rightarrow$	{PersNr, Name, Rang, Raum, Ort, Straße, 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, daß nur solche FDs abgeleitet werden, die von jeder Ausprägung erfüllt sind, für die F erfüllt ist. Vollständig bedeutet, daß 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: $\{PLZ\} \rightarrow \{Landesregierung\}$ läßt sich aus den FDs 1-6 für das Relationenschema *ProfessorenAdr* herleiten:

- {PLZ} → {BLand} (Dekomposition von FD Nr.3)
- {BLand} → {Landesregierung} (FD Nr.6)
- {PLZ} → {Landesregierung} (Transitivität)

Oft ist man an der Menge von Attributen α^+ interessiert, die von α gemäß der Menge F von FDs funktional bestimmt werden:

$$\alpha^+ := \{\beta \subseteq U \mid \alpha \rightarrow \beta \in F^+\}$$

Es gilt der Satz:

$$\alpha \rightarrow \beta \text{ folgt aus Armstrongaxiomen genau dann wenn } \beta \in \alpha^+.$$

Die Menge α^+ kann aus einer Menge F von FDs und einer Menge von Attributen α wie folgt bestimmt werden:

$$X^0 := \alpha$$
$$X^{i+1} := X^i \cup \gamma \ \ falls\ \beta \rightarrow \gamma \in F \wedge \beta \subseteq X^i$$

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 :

$$\begin{array}{lll} \text{Sei } U & = & \{A,B,C,D,E,G\} \\ \text{Sei } F & = & \{AB \rightarrow C, C \rightarrow A, BC \rightarrow D, ACD \rightarrow B, \\ & & D \rightarrow EG, BE \rightarrow C, CG \rightarrow BD, CE \rightarrow AG\} \\ \text{Sei } X & = & \{B,D\} \\ X^0 & = & BD \\ X^1 & = & BDEG \\ X^2 & = & BCDEG \\ X^3 & = & ABCDEG = X^4, \text{ Abbruch.} \\ \text{Also: } (BD)^+ & = & ABCDEG \end{array}$$

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 muß 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 :

$$\text{Sei } U = \{ A, B, C, D, E, G\} \qquad \text{Sei } F = \{ \begin{array}{lcll} AB & \rightarrow & C, & D \rightarrow EG \\ C & \rightarrow & A, & BE \rightarrow C, \\ BC & \rightarrow & D, & CG \rightarrow BD, \\ ACD & \rightarrow & B, & CE \rightarrow AG \} \end{array}$$

Aufspalten der rechten Seiten liefert

$$\begin{array}{ccc} AB & \rightarrow & C \\ C & \rightarrow & A \\ BC & \rightarrow & D \\ ACD & \rightarrow & B \\ D & \rightarrow & E \\ D & \rightarrow & G \\ BE & \rightarrow & C \\ CG & \rightarrow & B \\ CG & \rightarrow & D \\ CE & \rightarrow & A \\ CE & \rightarrow & G \end{array}$$

$$\begin{array}{lllccl} \text{Regel} & CE \rightarrow A & \text{ist redundant wegen} & C & \rightarrow & A \\ & CG \rightarrow B & \text{ist redundant wegen} & CG & \rightarrow & D \\ & & & C & \rightarrow & A \\ & & & ACD & \rightarrow & B \\ \text{Regel} & ACD \rightarrow B & \text{kann gekürzt werden zu} & CD & \rightarrow & B, \text{ wegen } C \rightarrow A \end{array}$$

10.4 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).

10.5 Zerlegung von Relationen

Unter *Normalisierung* verstehen wir die Zerlegung eines Relationenschemas $\mathcal{R}$ in die Relationenschemata $\mathcal{R}_1, \mathcal{R}_2, \ldots \mathcal{R}_n$, 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 $\mathcal{R}_2$ heißt *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
Kowalski	Kemper	Pils
Kowalski	Eickler	Hefeweizen
Innsteg	Kemper	Hefeweizen

Besucht	
Kneipe	Gast
Kowalski	Kemper
Kowalski	Eickler
Innsteg	Kemper

Trinkt	
Gast	Bier
Kemper	Pils
Eickler	Hefeweizen
Kemper	Hefeweizen

Besucht $\bowtie$ Trinkt		
Kneipe	Gast	Pils
Kowalski	Kemper	Pils
Kowalski	*Kemper*	*Hefeweizen*
Kowalski	Eickler	Hefeweizen
Innsteg	*Kemper*	*Pils*
Innsteg	Kemper	Hefeweizen

Eine Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1, \ldots, \mathcal{R}_n$ heißt *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**	**Straße**	PLZ
Frankfurt	Hessen	Goethestraße	60313
Frankfurt	Hessen	Galgenstraße	60437
Frankfurt	Brandenburg	Goethestraße	15234

Straßen	
PLZ	**Straße**
15234	Goethestraße
60313	Gorthestraße
60437	Glagenstraße

Orte		
Ort	BLand	**PLZ**
Frankfurt	Hessen	60313
Frankfurt	Hessen	60437
Frankfurt	Brandenburg	15234

Es sollen die folgenden funktionalen Abhängigkeiten gelten:

- {PLZ} → {Ort, BLand}
- {Straße, Ort, BLand} → {PLZ}

Die Zerlegung ist verlustlos, da PLZ das einzige gemeinsame Attribut ist und {PLZ} → {Ort, BLand} gilt.

Die funktionale Abhängigkeit {Straße, Ort, BLand} → {PLZ} ist jedoch keiner der beiden Relationen zuzuordnen, so daß diese Zerlegung nicht abhängigkeitserhaltend ist.

Folgende Auswirkung ergibt sich: Der Schlüssel von *Straßen* ist {PLZ, Straße} und erlaubt das Hinzufügen des Tupels [15235, Goethestraße].

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} → {PLZ} entdeckt.

10.6 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 müßte 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

10.7 Zweite Normalform

Ein Attribut heißt *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 muß für $1 \leq j \leq n$ gelten:

$$\kappa_j \dot{\rightarrow} A \in F^+$$

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 10.1 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.

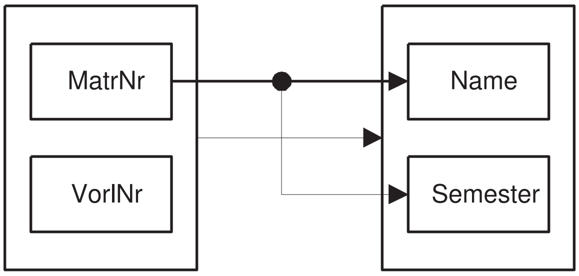


Abbildung 10.1: Graphische Darstellung der funktionalen Abhängigkeiten von StudentenBelegung

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.

10.8 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, daß bei einem Dekanswechsel mehrere Tupel geändert werden müssen.

Seien X, Y, Z Mengen von Attributen eines Relationenschemas $\mathcal{R}$ mit Attributmenge U.
Z heißt *transitiv abhängig* von X, falls gilt

$$\begin{gathered} X \cap Z = \emptyset \\ \exists\, Y \subset U : X \cap Y = \emptyset, Y \cap Z = \emptyset \\ X \rightarrow Y \rightarrow Z, Y \not\rightarrow X \end{gathered}$$

Zum Beispiel ist in der Relation *Student* das Attribut *Dekan* transitiv abhängig von *MatrNr*:

$$\text{MatrNr} \overset{\not\leftarrow}{\rightarrow} \ \textit{Fachbereich} \rightarrow \textit{Dekan}$$

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.

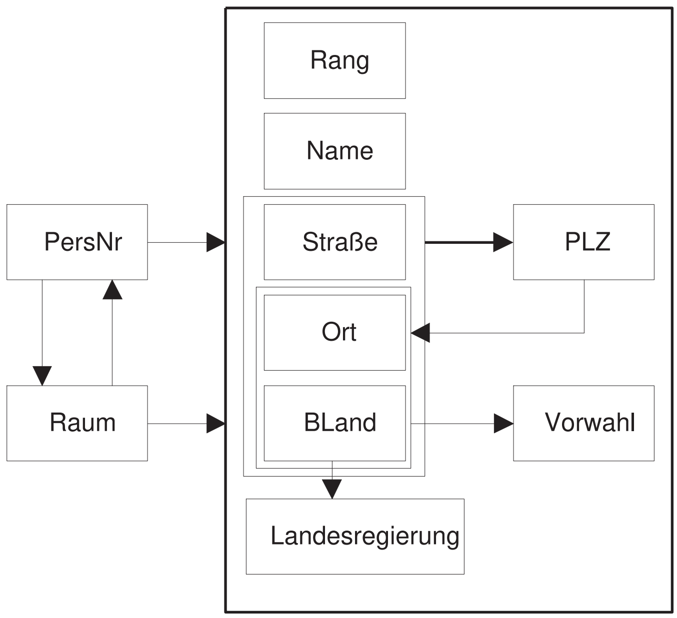


Abbildung 10.2: Graphische Darstellung der funktionalen Abhängigkeiten von ProfessorenAdr

Als Beispiel betrachten wir die bereits bekannte Relation

$$\text{ProfessorenAdr} : \{[\text{PersNr, Name, Rang, Raum, Ort, Straße} \\ \text{PLZ, Vorwahl, BLand, Landesregierung}]\}$$

Abbildung 10.2 zeigt die funktionalen Abhängigkeiten in der graphischen Darstellung. Offenbar ist die Relation nicht in der dritten Normalform, da das Nichtprimärattribut *Vorwahl* nicht-transitiv-abhängig vom Schlüsselkandidaten *PersNr* ist:

$$\textit{PersNr} \overset{\not\leftarrow}{\rightarrow} \ \{\textit{Ort, BLand}\} \rightarrow \textit{Vorwahl}$$

Um Relationen in dritter Normalform zu erhalten, ist häufig eine starke Aufsplittung erforderlich. Dies führt natürlich zu erhöhtem Aufwand bei Queries, da ggf. mehrere Verbundoperationen erforderlich werden.

Kapitel 11

Transaktionsverwaltung

11.1 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,-DM 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.

11.2 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.

11.3 Abschluß einer Transaktion

Der erfolgreiche Abschluß einer Transaktion erfolgt durch eine Sequenz der Form

$$BOT \quad op_1;\ op_2;\ \ldots\ ; op_n; \quad commit$$

Der erfolglose Abschluß einer Transaktion erfolgt entweder durch eine Sequenz der Form

$$BOT \quad op_1;\ op_2;\ \ldots\ ; op_j; \quad abort$$

oder durch das Auftreten eines Fehlers

$$BOT \quad op_1;\ op_2;\ \ldots\ ; op_k; \quad < Fehler >$$

In diesen Fällen muß der Transaktionsverwalter auf den Anfang der Transaktion zurücksetzen.

11.4 Eigenschaften von Transaktionen

Die Eigenschaften des Transaktionskonzepts werden unter der Abkürzung $ACID$ zusammengefaßt:

- **Atomicity:** Eine Transaktion stellt eine nicht weiter zerlegbare Einheit dar mit dem Prinzip *alles-oder-nichts.*
- **Consistency:** Nach Abschluß 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).

11.5 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;
```

11.6 Zustandsübergänge einer Transaktion

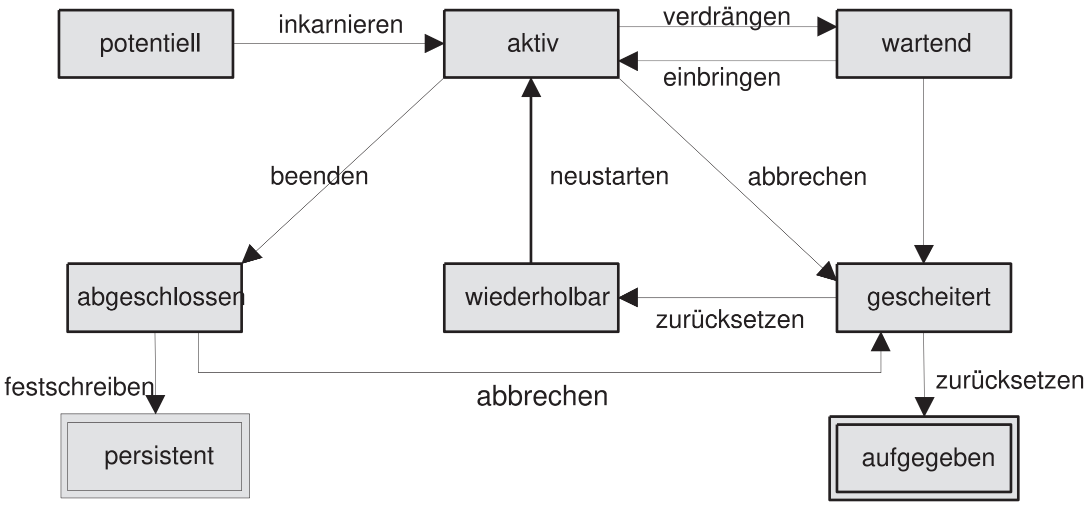


Abbildung 11.1: Zustandsübergangsdiagramm für Transaktionen

Abbildung 11.1 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.

Kapitel 12

Mehrbenutzersynchronisation

12.1 Multiprogramming

Unter *Multiprogramming* versteht man die nebenläufige, verzahnte Ausführung mehrerer Programme. Abbildung 12.1 zeigt exemplarisch die dadurch erreichte bessere CPU-Auslastung.

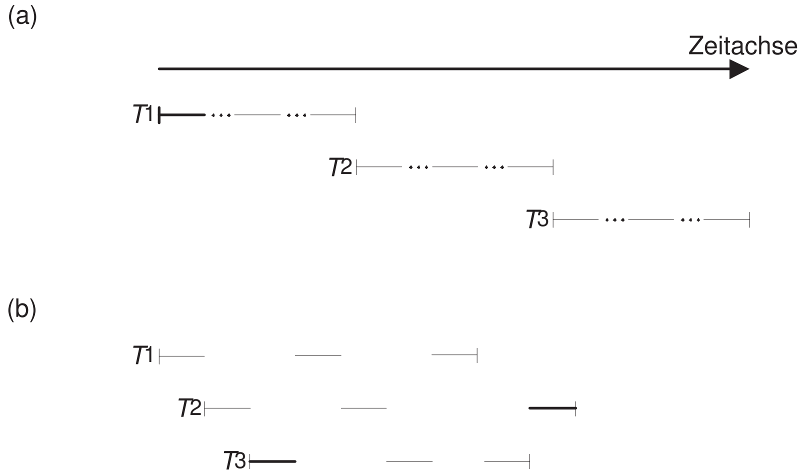


Abbildung 12.1: Einbenutzerbetrieb (a) versus Mehrbenutzerbetrieb (b)

12.2 Fehler bei unkontrolliertem Mehrbenutzerbetrieb

12.2.1 Lost Update

Transaktion T_1 transferiert 300,- DM von Konto A nach Konto B,
Transaktion T_2 schreibt Konto A die 3 % Zinseinkünfte gut.

Den Ablauf zeigt Tabelle 12.1. 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.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1$ - 300	
3.		$\text{read}(A, a_2)$
4.		$a_2 := a_2$ * 1.03
5.		$\text{write}(A, a_2)$
6.	$\text{write}(A, a_1)$	
7.	$\text{read}(B, b_1)$	
8.	$b_1 := b_1 + 300$	
9.	$\text{write}(B, b_1)$	

Tabelle 12.1: Beispiel für Lost Update

12.2.2 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 12.2.

Schritt	T_1	T_2
1.	$\text{read}(A, a_1)$	
2.	$a_1 := a_1 - 300$	
3.	$\text{write}(A, a_1)$	
4.		$\text{read}(A, a_2)$
5.		$a_2 := a_2 * 1.03$
6.		$\text{write}(A, a_2)$
7.	$\text{read}(B, b_1)$	
8.	...	
9.	**abort**	

Tabelle 12.2: Beispiel für Dirty Read

12.2.3 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 12.3.

T_1	T_2
	select sum(KontoStand) **from** Konten;
insert into Konten **values** (C, 1000, ...);	
	select sum(KontoStand) **from** Konten;

Tabelle 12.3: Beispiel für das Phantomproblem

12.3 Serialisierbarkeit

Eine *Historie*, auch genannt *Schedule*, für eine Menge von Transaktionen ist eine Festlegung für die Reihenfolge sämtlicher relevanter Datenbankoperationen. Ein Schedule heißt *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 12.4.

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 12.4: Serialisierbare Historie

Offenbar wird derselbe Effekt verursacht, als wenn zunächst T_1 und dann T_2 ausgeführt worden wäre, wie Tabelle 12.5 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 12.5: Serielle Historie

Wir nennen deshalb das (verzahnte) Schedule *serialisierbar*.

Tabelle 12.6 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 12.6: Nicht-serialisierbares Schedule

Der Grund liegt darin, daß 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 12.7 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 12.7: Zwei verzahnte Überweisungen

In beiden Fällen wird Konto A mit 150,- DM belastet und Konto B werden 150,- DM gutgeschrieben.

Unter einer anderen Semantik würde T_1 einen Betrag von 50,- DM von A nach B überweisen und Transaktion T_2 würde beiden Konten jeweils 3 % Zinsen gutschreiben. Tabelle 12.8 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 12.8: Überweisung verzahnt mit Zinsgutschrift

Offenbar entspricht diese Reihenfolge keiner möglichen seriellen Abarbeitung $T_1 T_3$ oder $T_3 T_1$, denn es fehlen in jedem Falle Zinsen in Höhe von 3 % von 50,- DM = 1,50 DM.

12.4 Theorie der Serialisierbarkeit

Eine *Transaktion* T_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 muß auch $p_i <_{H_2} q_j$ gelten. Die Anordnung der nicht in Konflikt stehenden Operationen ist irrelevant.

12.5 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 $T_i \rightarrow T_j$ 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 läßt sich die äquivalente serielle Historie aus der topologischen Sortierung des Serialisierbarkeitsgraphen bestimmen.

Als Beispiel-Input für diesen Algorithmus verwenden wir die in Tabelle 12.9 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 12.9: 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

$$T_2 \to T_1,$$

$$T_1 \to T_2,$$

$$T_2 \to T_3,$$

$$T_2 \to T_3.$$

Den resultierenden Graph zeigt Abbildung 12.2

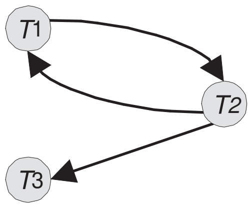


Abbildung 12.2: Der zu Historie H konstruierte Serialisierbarkeitsgraph

Da der konstruierte Graph einen Kreis besitzt, ist die Historie nicht serialisierbar.

12.6 Sperrbasierte Synchronisation

Bei der sperrbasierten Synchronisation wird während des laufenden Betriebs sichergestellt, daß 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 12.10 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 12.10: Kompatibilitätsmatrix

Folgendes Zwei-Phasen-Sperrprotokoll (*two phase locking*, $2PL$) garantiert die Serialisierbarkeit:

1. Jedes Objekt muß vor der Benutzung gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. Eine Transaktion respektiert vorhandene Sperren gemäß 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 muß eine Transaktion alle ihre Sperren zurückgeben.

Abbildung 12.3 visualisiert den Verlauf des 2PL-Protokolls. Tabelle 12.11 zeigt eine Verzahnung zweier Transaktionen nach dem 2PL-Protokoll.

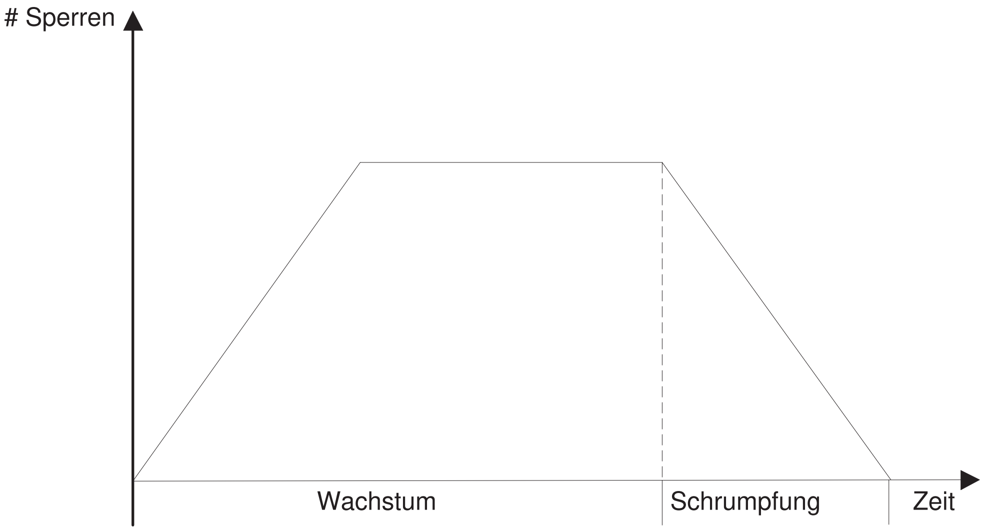


Abbildung 12.3: 2-Phasen-Sperrprotokoll

Schritt	T_1	T_2	Bemerkung
1.	**BOT**		
2.	**lockX**(A)		
3.	read(A)		
4.	write(A)		
5.		**BOT**	
6.		**lockS**(A)	T_2 muß warten
7.	**lockX**(B)		
8.	read(B)		
9.	**unlockX**(A)		T_2 wecken
10.		read(A)	
11.		**lockS**(B)	T_2 muß warten
12.	write(B)		
13.	**unlockX**(B)		T_2 wecken
14.		read(B)	
15.	**commit**		
16.		**unlockS**(A)	
17.		**unlockS**(B)	
18.		**commit**	

Tabelle 12.11: Beispiel für 2PL-Protokoll

12.7 Verklemmungen (Deadlocks)

Ein schwerwiegendes Problem bei sperrbasierten Synchronisationsmethoden ist das Auftreten von Verklemmungen (englisch: deadlocks). Tabelle 12.12 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 muß warten auf T_2
9.		**lockS**(A)	T_2 muß warten auf T_1
10.	...	...	$\Rightarrow$ *Deadlock*

Tabelle 12.12: Ein verklemmter Schedule

Eine Methode zur Erkennung von Deadlocks ist die $Time-out-$Strategie. Falls eine Transaktion innerhalb eines Zeitmaßes (z. B. 1 Sekunde) keinerlei Fortschritt erzielt, wird sie zurückgesetzt. Allerdings ist die Wahl des richtigen Zeitmaßes 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. Bild 12.4 zeigt ein Beispiel.

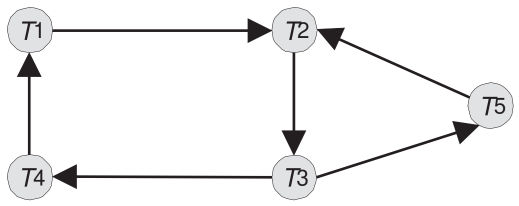


Abbildung 12.4: Wartegraph mit zwei Zyklen

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.

12.8 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 12.5 zeigt die hierarchische Anordnung der möglichen Sperrgranulate.

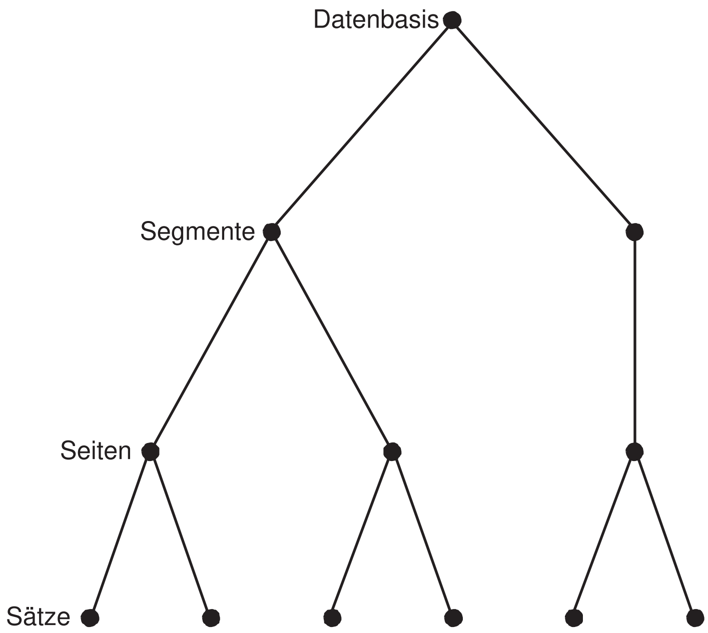


Abbildung 12.5: Hierarchie der Sperrgranulate

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 großer 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 12.13 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 12.13: Kompatibilitätsmatrix beim Multiple-Granularity-Locking

Die Sperrung eines Datenobjekts muß so durchgeführt werden, daß 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.

Abbildung 12.6 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.

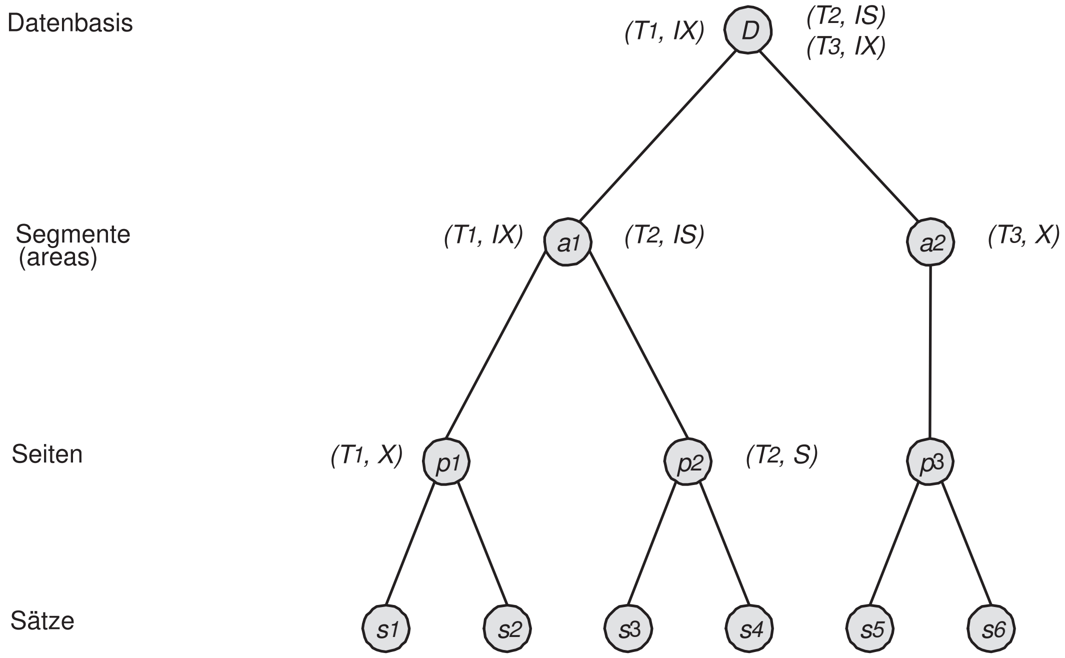


Abbildung 12.6: Datenbasis-Hierarchie mit Sperren

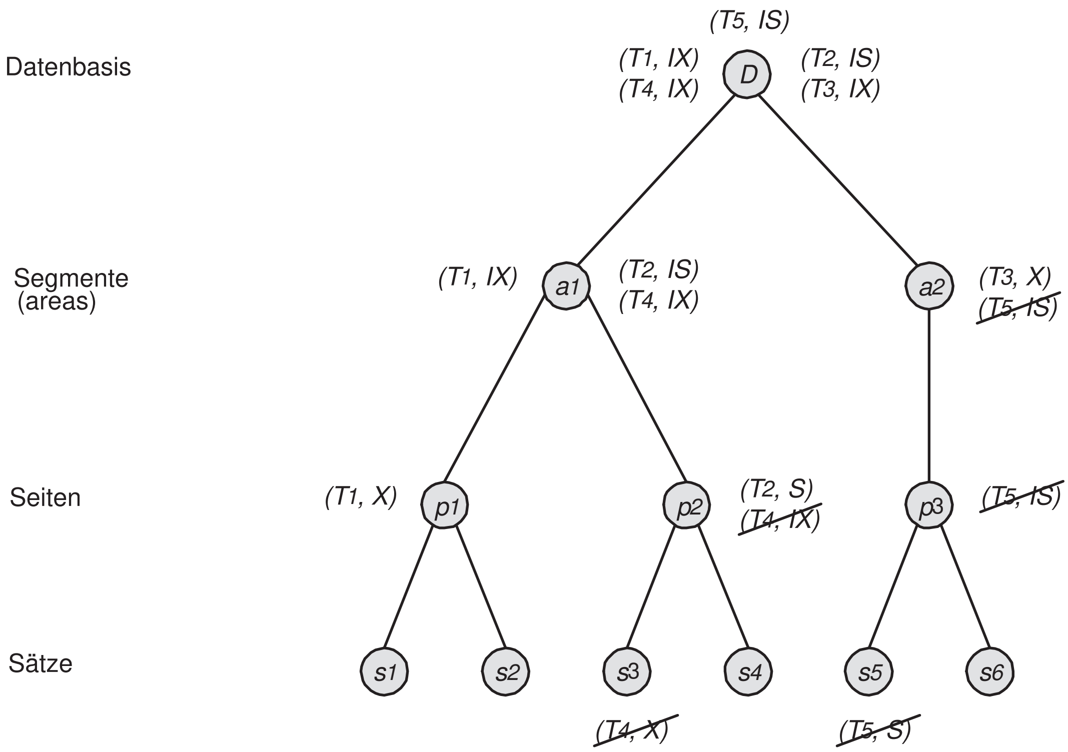


Abbildung 12.7: Datenbasis-Hierarchie mit zwei blockierten Transaktionen

- 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.

Bild 12.7 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.

12.9 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 gemäß 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).

Bei Eintreten von Fall 1 und 2 muß die Transaktion zurückgesetzt zu werden.

Bei den beiden anderen Fällen brauchen die Transaktionen nicht zurückgesetzt zu werden:

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.

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 ≤ t < t_w) then tue nichts
else {(X = read and t < t_w) or (X = write and t < t_r)}
   setze Transaktion zurück
```

Tabelle 12.14 und 12.15 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 12.14: Beispiel für Zeitstempelverfahren

In Tabelle 12.14 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 12.15 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 12.15: Beispiel für Zeitstempelverfahren

Kapitel 13

Recovery

Aufgabe der Recovery-Komponente des Datenbanksystems ist es, nach einem Fehler den jüngsten konsistenten Datenbankzustand wiederherzustellen.

13.1 Fehlerklassen

Wir unterscheiden drei Fehlerklassen:

1. lokaler Fehler in einer noch nicht festgeschriebenen Transaktion,
2. Fehler mit Hauptspeicherverlust,
3. Fehler mit Hintergrundspeicherverlust.

13.1.1 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.

13.1.2 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, daß die im Puffer durchgeführten Änderungen erst mit dem Zurückschreiben in die materialisierte Datenbasis permanent werden. Abbildung 13.1 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.

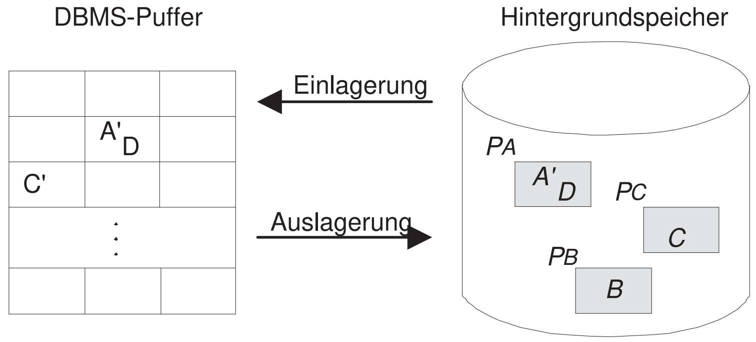


Abbildung 13.1: Schematische Darstellung der zweistufigen Speicherhierarchie

Bei einem Verlust des Hauptspeicherinhalts verlangt das Transaktionsparadigma, daß

- 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.

13.1.3 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.

13.2 Die Speicherhierarchie

13.2.1 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 Abschluß 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.

13.2.2 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
¬steal	• kein Redo • kein Undo	• Redo • kein Undo
steal	• kein Redo • Undo	• Redo • 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*).

13.2.3 Einbringstrategie

Es gibt zwei Strategien zur Organisation des Zurückschreibens:

- *update-in-place*: Jeder ausgelagerten Seite im Datenbankpuffer entspricht eine Seite im Hintergrundspeicher, auf die sie kopiert wird im Falle einer Modifikation.
- Twin-Block-Verfahren: Jeder ausgelagerten 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, sodaß bei einem Fehler während des Zurückschreibens der letzte Stand noch verfügbar ist.

13.3 Protokollierung der Änderungsoperationen

Wir gehen im weiteren von folgender Systemkonfiguration aus:

- *steal* : Nicht fixierte Seiten können jederzeit ersetzt werden.
- $\neg$*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.

13.3.1 Struktur der Log-Einträge

Für jede Änderungsoperation, die von einer Transaktion durchgeführt wird, werden 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).

13.3.2 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* muß A um 50 erniedrigt werden.
- Für ein *Undo* muß 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

13.3.3 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.

Um zu erkennen, ob das *Before-Image* oder *After-Image* in der materialisierten Datenbasis enthalten ist, 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 läßt sich erkennen, ob für einen bestimmten Log-Eintrag das *Before-Image* oder das *After-Image* in der Seite steht:

- 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 größer oder gleich der LSN des Log-Eintrags, dann wurde bereits das *After-Image* auf den Hintergrundspeicher propagiert.

13.3.4 Schreiben der Log-Information

Bevor eine Änderungsoperation ausgeführt wird, muß der zugehörige Log-Eintrag angelegt werden. Die Log-Einträge werden im *Log-Puffer* im Hauptspeicher zwischengelagert. Abbildung 13.2 zeigt das Wechselspiel zwischen den beteiligten Sicherungskomponenten.

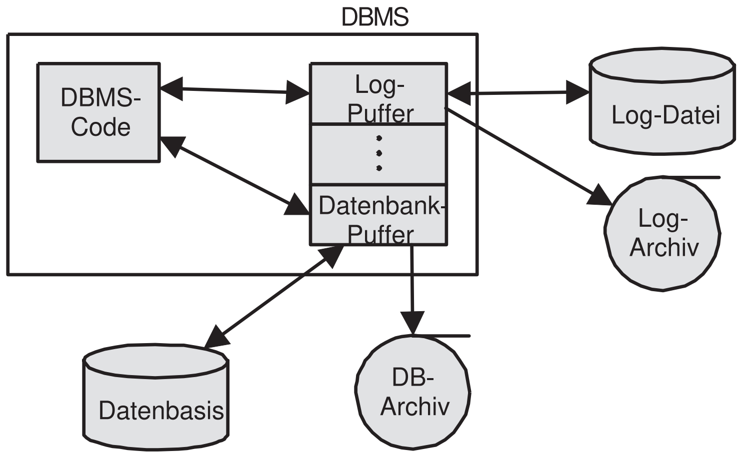


Abbildung 13.2: Speicherhierarchie zur Datensicherung

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 13.3). Die Log-Einträge werden gleichzeitig auf das temporäre Log (Platte) und auf das Log-Archiv (Magnetband) geschrieben.

13.3.5 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

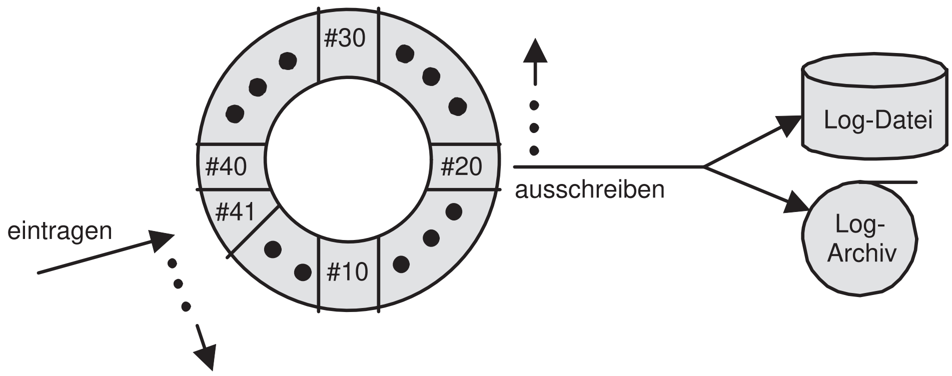


Abbildung 13.3: Log-Ringpuffer

Fehlerfall die Änderungen nicht abgeschlossener Transaktionen aus den modifizierten Seiten der materialisierten Datenbasis entfernen zu können (*undo*).

13.4 Wiederanlauf nach einem Fehler

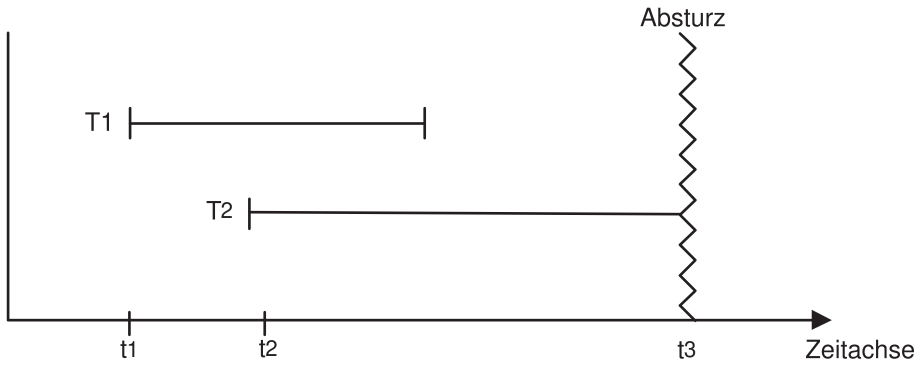


Abbildung 13.4: Zwei Transaktionstypen bei Systemabsturz

Abbildung 13.4 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*.

Der Wiederanlauf geschieht in drei Phasen (Abbildung 13.5):

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.

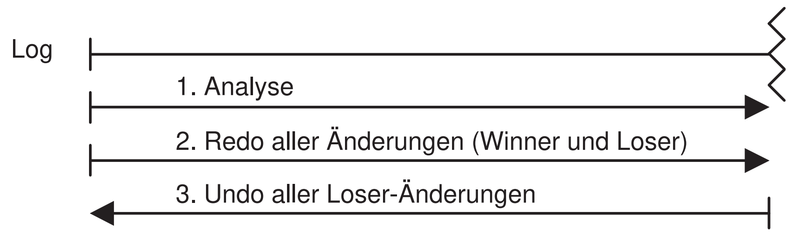


Abbildung 13.5: Wiederanlauf in drei Phasen

2. *Redo*: Es werden alle protokollierten Änderungen 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 größer 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, daß die *Redo*- und *Undo*-Phasen *idempotent* sind, d.h. sie müssen auch nach mehrmaliger Ausführung (hintereinander) immer wieder dasselbe Ergebnis liefern:

$$undo(undo(\ldots(undo(a))\ldots)) = undo(a)$$
$$redo(redo(\ldots(redo(a))\ldots)) = redo(a)$$

13.5 Lokales Zurücksetzen einer Transaktion

Die zu einer zurückzusetzenden Transaktion gehörenden Log-Einträge werden mit Hilfe des *PrevLSN*-Eintrags in umgekehrter Reihenfolge abgearbeitet. Jede Änderung wird durch eine *Undo*-Operation rückgängig gemacht.

Wichig in diesem Zusammenhang ist die Verwendung von *rücksetzbaren Historien*, die auf den Schreib/Leseabhängigkeiten basieren.

Wir sagen, daß 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 heißt *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.

13.6 Sicherungspunkte

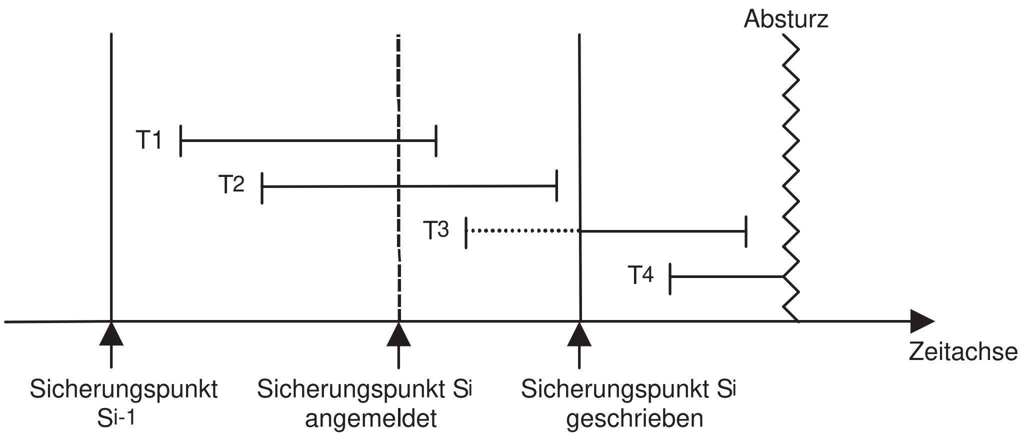


Abbildung 13.6: Transaktionsausführung relativ zu einem Sicherungspunkt

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 muß.

Abbildung 13.6 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.

13.7 Verlust der materialisierten Datenbasis

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 13.7 faßt die zwei möglichen Recoveryarten nach einem Systemabsturz zusammen:

- Der obere (schnellere) Weg wird bei intaktem Hintergrundspeicher beschritten.
- Der untere (langsamere) Weg wird bei zerstörtem Hintergrundspeichert beschritten.

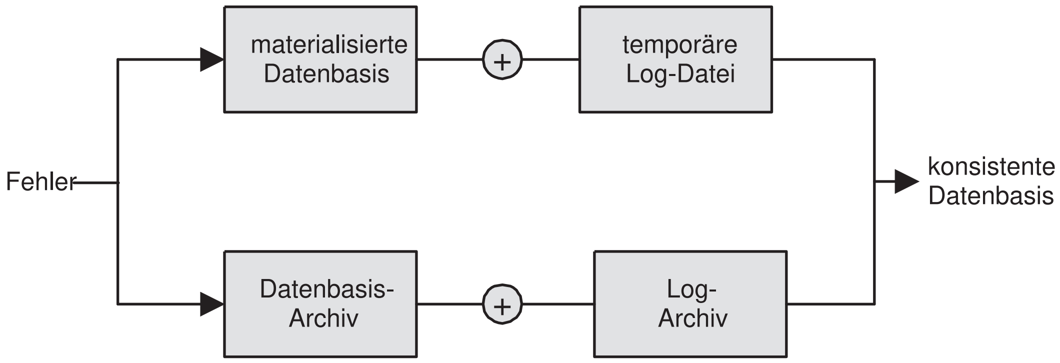


Abbildung 13.7: Zwei Recovery-Arten

Kapitel 14

Sicherheit

In diesem Kapitel geht es um den Schutz gegen absichtliche Beschädigung oder Enthüllung von sensiblen Daten. Abbildung 14.1 zeigt die hierarchische Kapselung verschiedenster Maßnahmen.

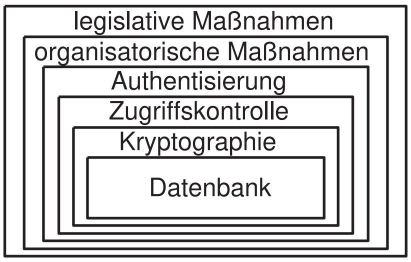


Abbildung 14.1: Ebenen des Datenschutzes

14.1 Legislative Maßnahmen

Im *Gesetz zum Schutz vor Mißbrauch personenbezogener Daten bei der Datenverarbeitung* ist festgelegt, welche Daten in welchem Umfang schutzbedürftig sind.

14.2 Organisatorische Maßnahmen

Darunter fallen Maßnahmen, um den persönlichen Zugang zum Computer zu regeln:

- bauliche Maßnahmen
- Pförtner
- Ausweiskontrolle
- Diebstahlsicherung

- Alarmanlage

14.3 Authentisierung

Darunter fallen Maßnahmen zur Überprüfung der Identität eines Benutzers:

- Magnetkarte
- Stimmanalyse/Fingerabdruck
- Paßwort: w ohne Echo eintippen,
  System überprüft, ob $f(w)$ eingetragen ist, f^{-1} aus f nicht rekonstruierbar
- dynamisches Paßwort: vereinbare Algorithmus, der aus Zufallsstring gewisse Buchstaben heraussucht

Paßwortverfahren sollten mit Überwachungsmaßnahmen kombiniert werden (Ort, Zeit, Fehleingabe notieren)

14.4 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:

$$\text{Zugriff } (A, \text{Gehalt}): \quad \begin{array}{lllr} R: & \text{Gehalt} & < & 10.000 \\ W: & \text{Gehalt} & < & 5.000 \end{array}$$

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 groß ist.

- **Beispiel:**
  Es habe Manager X als einziger eine bestimmte Eigenschaft, z. B. habe er das höchste Gehalt. Dann läßt 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(4) 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. Außerdem 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, E verliert keine Rechte. Falls aber vorher A Rechte an B gab, z.B. durch

```
A: grant all on angest to B with grant option
```

  dann müssten D und E ihr *update*-Recht verlieren.

14.5 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.

14.6 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 14.2 skizziert: Der Klartext x dient als Eingabe für ein Verschlüsselungsverfahren *encode*, welches über einen Schlüssel e parametrisiert ist. Das heißt, das grundsätzliche Verfahren der Verschlüsselung ist allen Beteiligten bekannt, mit Hilfe des Schlüssels e kann der Vorgang jeweils individuell beeinflußt werden. Auf der Gegenseite wird mit dem Verfahren *decode* und seinem Schlüssel d der Vorgang umgekehrt und somit der Klartext rekonstruiert.

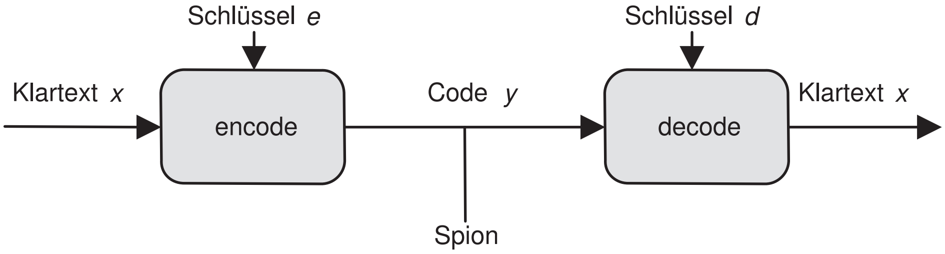


Abbildung 14.2: Ablauf beim Übertragen einer Nachricht

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 ⊕ Schlüssel
Schlüssel	`111010011`	
Klartext	`010111001`	= Chiffre ⊕ 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 groß 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, daß 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.

14.6.1 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 läßt 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 $x := dec_B(y)$.

14.6.2 Das RSA-Verfahren

Im Jahre 1978 schlugen Rivest, Shamir, Adleman folgendes Verfahren vor:

geheim:	Wähle zwei große 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 $e \cdot d \equiv 1 \bmod \varphi(n)$
öffentlich:	enc(x):= $x^e \bmod n$
geheim:	dec(y):= $y^d \bmod n$

- **Beispiel:**
  $p = 11,\ q = 13,\ d = 23\ \Rightarrow$
  $n = 143,\ e = 47$
  $enc(x) := x^{47} \bmod 143$
  $dec(y) := y^{23} \bmod 143$

14.6.3 Korrektheit des RSA-Verfahrens

Die Korrektheit stützt sich auf den **Satz von Fermat/Euler:**

$$x \text{ rel. prim zu } n \Rightarrow x^{\varphi(n)} \equiv 1 \bmod n$$

14.6.4 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:

$$x^e := \begin{cases} (x^{e/2})^2 & \text{falls } e \text{ gerade} \\ (x^{\lfloor e/2 \rfloor})^2 \cdot x & \text{falls } e \text{ ungerade} \end{cases}$$

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 großen 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 Größe $n = 2^{500}$ 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:

$$x \notin L \Rightarrow \text{ Antwort: nein}$$
$$x \in L \Rightarrow \text{ Antwort: } \underbrace{\text{ja}}_{>1-\varepsilon} \text{ oder } \underbrace{\text{nein}}_{<=\varepsilon}$$

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) $x^q \equiv 1 \bmod n$ oder
2) $x^{q \cdot 2^i} \equiv -1 \bmod n$ 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 $x < n$ heißt Zeuge für die Zusammengesetztheit von n

1) $ggt(x, n) \;\neq 1$ oder
2) $x^q \not\equiv 1 \bmod n$ 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: $(\frac{1}{4})^{50} \sim 10^{-30}$

14.6.5 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.

14.6.6 Implementation des RSA-Verfahrens

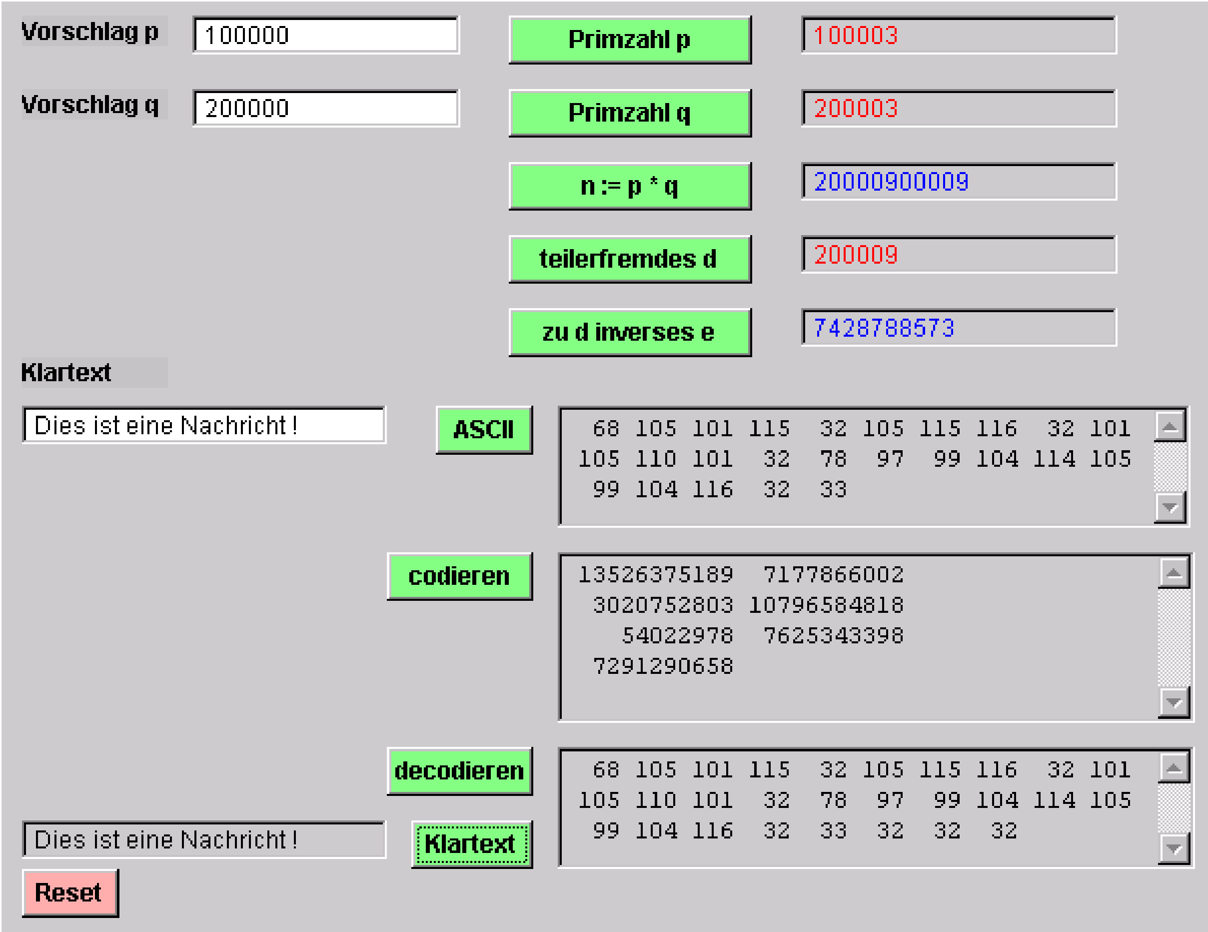


Abbildung 14.3: Java-Applet mit RSA-Algorithmus

14.6.7 Anwendungen des RSA-Verfahrens

Verschlüsseln :

Alice schickt $y := enc_B(x)$ an Bob.
Bob bildet $x := dec_B(y)$.

Unterschreiben einer geheimen Nachricht :

Alice schickt $y := enc_B(dec_A(x))$ an Bob.
Bob bildet $x := enc_A(dec_B(y))$.

Unterschreiben einer öffentlichen Nachricht :

Sei f eine unumkehrbare Hashfunktion.
Alice bildet $z := dec_A(f(x))$.
Alice schickt $< x\ , z >$ an Bob.
Bob vergleicht $f(x)$ mit $enc_A(z)$.

Anonymer Zahlungsverkehr mit blinder Unterschrift :

Alice würfelt Schecknummer x und Ausblendfaktor r.
Alice schickt $s := x \cdot enc_{Bank}(r)$ zur Bank.
Bank belastet Alice's Konto mit 1,-DM und schickt $z = dec_{Bank}(s)$ zurück.
Alice erhält also $(x \cdot r^e)^d \bmod n = (x^d \cdot r) \bmod n$.
Alice bildet z/r und verfügt nun über $y := x^d \bmod n = dec_{Bank}(x)$.
Alice präsentiert y dem Kaufmann.
Der Kaufmann verifiziert y durch $enc_{Bank}(y)$.
Der Kaufmann schickt y an die Bank.
Die Bank trägt $enc_{Bank}(y)$ in die Liste der verbrauchten Schecks ein.
Die Bank schreibt dem Kaufmann 1,- DM gut.
Der Kaufmann schickt die Ware an Alice.

Zertifizierungscenter :

Bob erzeugt selbst sein Schlüsselpaar enc_B und dec_B.
Bob besorgt sich persönlich von einem Zertifizierungscenter Z
dessen öffentlichen Schlüssel enc_Z und ein Zertifikat $z := dec_Z(enc_B)$.
Bob schickt z an Alice.
Alice bildet $enc_Z(z)$ und erhält somit enc_B.
Alice kann nun sicher sein, Bob's öffentlichen Schlüssel vor sich zu haben.

Kapitel 15

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 *verhaltensmäßigen* Information integriert.

15.1 Schwächen relationaler Systeme

Die Relation

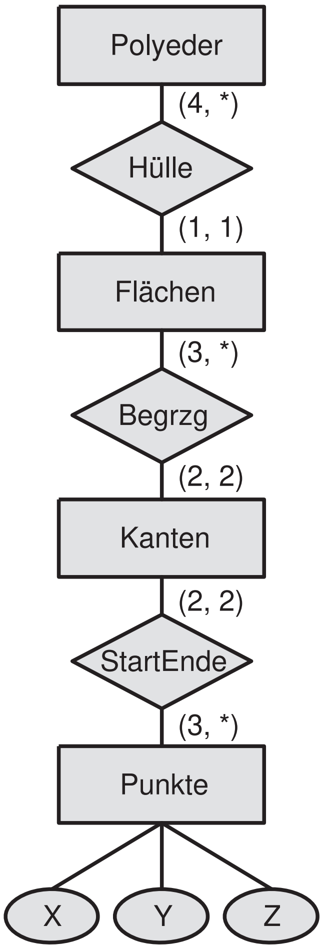
$$\text{Buch} \quad : \quad \{[\ \underline{\text{ISBN}}, \text{Verlag, Titel, Autor, Version, Stichwort}]\}$$

erfordert bei 2 Autoren, 5 Versionen, 6 Stichworten für jedes Buch $2 \times 5 \times 6 = 60$ Einträge.

Eine Aufsplittung auf mehrere Tabellen ergibt

$$\begin{array}{rcl}
\text{Buch} & : & \{[\ \underline{\text{ISBN}}, \text{Titel, Verlag}\ ]\} \\
\text{Autor} & : & \{[\ \underline{\text{ISBN}}, \text{Name, Vorname}\ ]\} \\
\text{Version} & : & \{[\ \underline{\text{ISBN}}, \text{Auflage, Jahr}\ ]\} \\
\text{Stichwort} & : & \{[\ \underline{\text{ISBN}}, \text{Stichwort}\ ]\}
\end{array}$$

Nun sind die Informationen zu einem Buch auf vier Tabellen verteilt. Beim Einfügen eines neuen Buches muß mehrmals dieselbe ISBN eingegeben werden. Die referentielle Integrität



(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
...	...	...	...

(b) Relationales Schema

Abbildung 15.1: Modellierung von Polyedern

muß selbst überwacht werden. Eine Query der Form "*Liste Bücher mit den Autoren Meier und Schmidt*" ist nur sehr umständlich zu formulieren.

Abbildung 15.1a 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 15.1b 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.

15.2 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 muß.

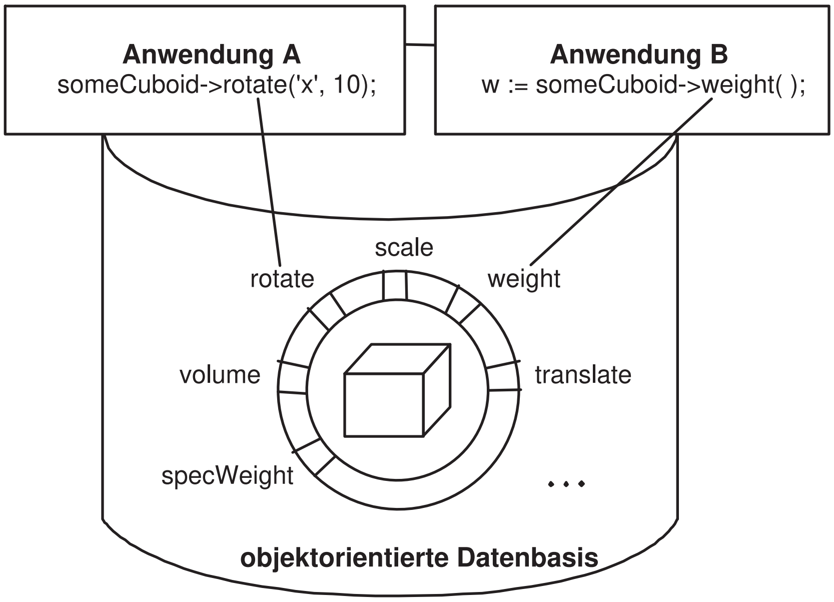


Abbildung 15.2: Visualisierung der Vorteile der objektorientierten Datenmodellierung

Abbildung 15.2 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.

15.3 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 umfaßt das objektorientierte Datenbanksystem und eine einheitliche Anbindung an bestehende Programmiersprachen. Bisher wurden Schnittstellen für C++ und Smalltalk vorgesehen. Außerdem wurde eine an SQL angelehnte deklarative Abfragesprache namens *OQL* (Object Query Language) entworfen.

15.4 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 heißen *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.

Abbildung 15.3 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.

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.

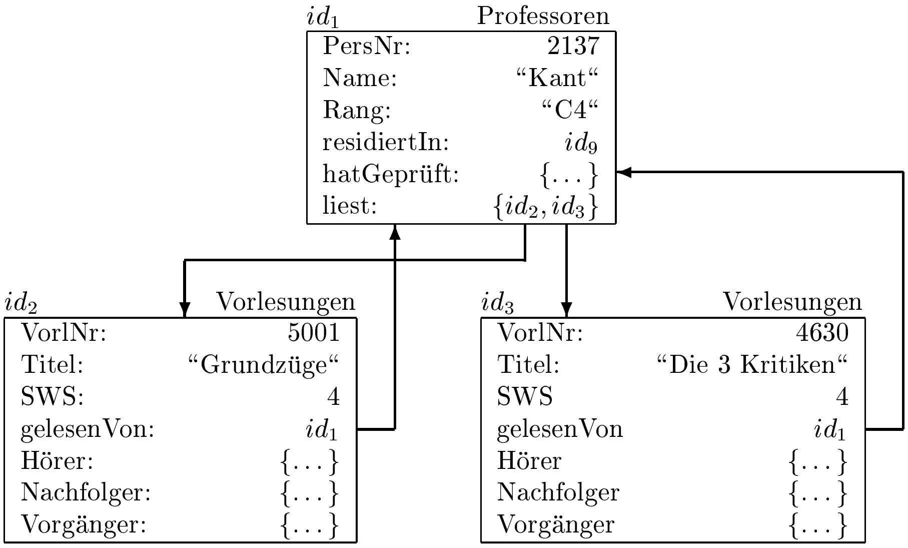


Abbildung 15.3: Einige Objekte aus der Universitätswelt

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.

15.5 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;
};
```

15.6 Definition von Beziehungen

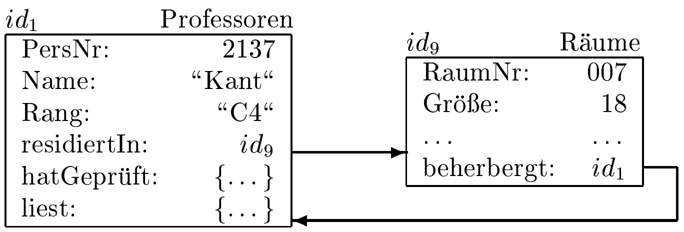


Abbildung 15.4: Ausprägung einer 1:1-Beziehung

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;
};
```

Abbildung 15.4 zeigt eine mögliche Ausprägung der Beziehungen *residiertIn* und *beherbergt*.

Allerdings wird durch die gegebene Klassenspezifikation weder die Symmetrie noch die 1:1-Einschränkung garantiert. Abbildung 15.5 zeigt einen inkonsistenten Zustand des Beziehungspaars *residiertIn* und *beherbergt*.

Um Inkonsistenzen dieser Art zu vermeiden, wurde im ODMG-Objektmodell das **inverse**-Konstrukt integriert:

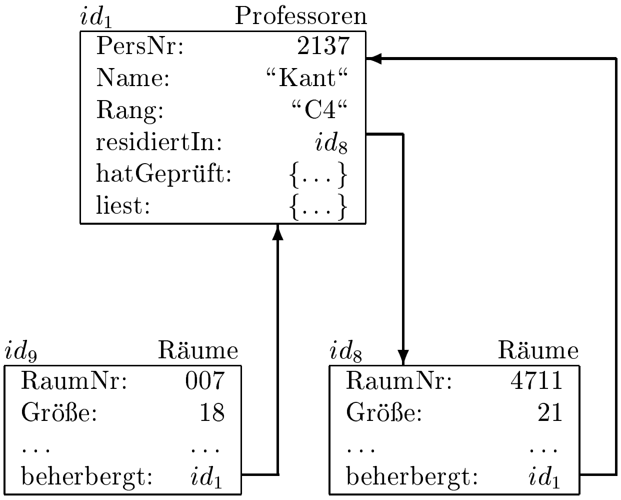


Abbildung 15.5: Inkonsistenter Zustand einer Beziehung

```
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, daß immer gilt:

$$p = r.beherbergt \quad \Leftrightarrow \quad r = p.residiertIn$$

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, daß 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, daß gilt:

$$s \in v.Hoerer \quad \Leftrightarrow \quad v \in s.hoert$$

Analog lassen sich rekursive $N : M$ - Beziehungen beschreiben:

```
class Vorlesungen {
  ...
  relationship set (Vorlesungn) Vorgaenger inverse Vorlesungen::Nachfolger;
  relationship set (Vorlesungn) Nachfolger inverse Vorlesungen::Vorgaenger;
};
```

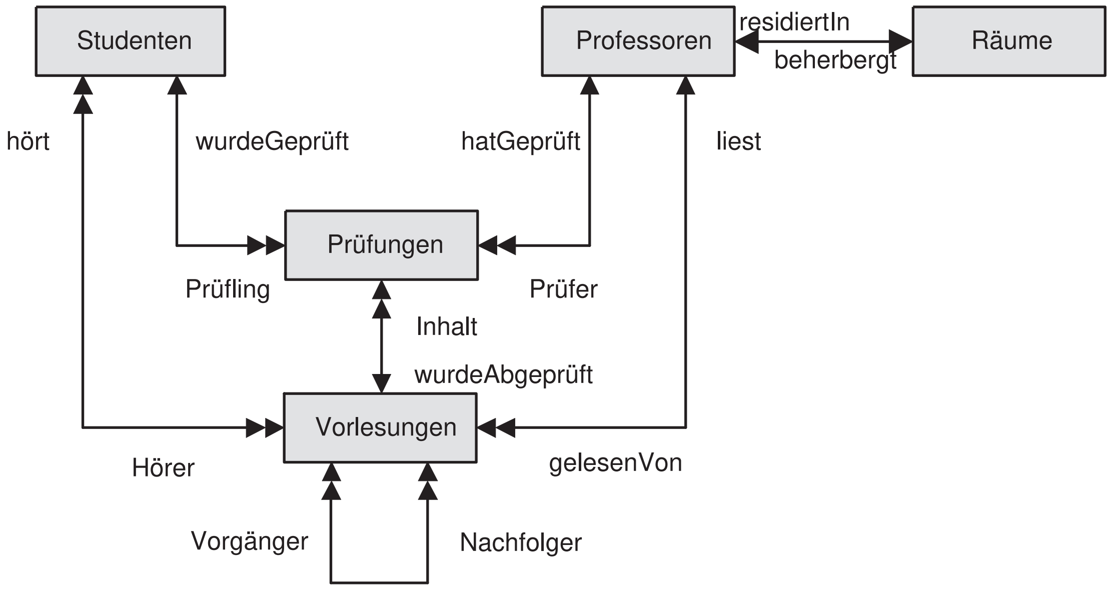


Abbildung 15.6: Modellierungen von Beziehungen im Objektmodell

Ternäre oder $n \geq 3$ stellige Beziehungen benötigen einen eigenständigen Objekttyp, der die Beziehung repräsentiert. Zum Beispiel wird die ternäre Beziehung

$$\text{pruefen} \quad : \quad \{[\ \text{MatrNr, VorlNr, PersNr, Note}\ ]\}$$

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 15.6 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
$\leftarrow\!\!\!\!\leftrightarrow$	bezeichnet eine N : 1-Beziehung	$\twoheadleftarrow\!\!\!\twoheadrightarrow$	bezeichnet eine N : M-Beziehung

15.7 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;
};
```

15.8 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 stößt die Ausnahmebehandlung *hatNochNichtGeprueft* an, wenn keine Prüfungen vorliegen.
- Der Mutator *befoerdert* erhöht den Rang um eine Stufe und stößt 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 (siehe Abschnitt 15.13) ist der Aufruf wahlweise mit Pfeil (`->`) oder mit einem Punkt (`.`)durchzuführen:

```
select p.wieHartAlsPruefer()
from p in AlleProfessoren
where p.Name = "Kant";
```

15.9 Vererbung

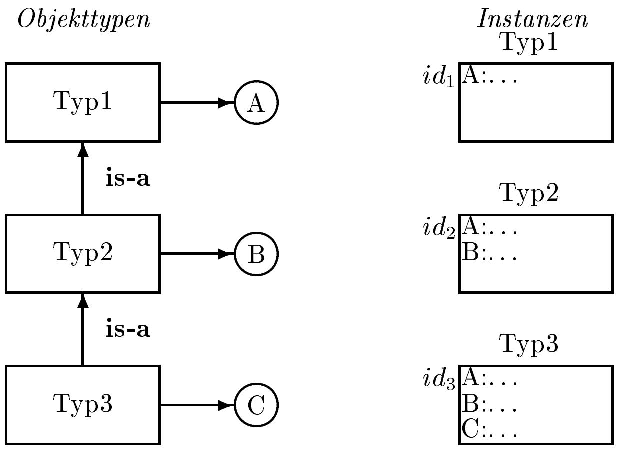


Abbildung 15.7: Schematische Darstellung einer abstrakten Typhierarchie

Das in Kapitel 2 eingeführte Konzept der Generalisierung bzw. Spezialisierung läßt sich bei objektorientierten Datenbanksystemen mit Hilfe der *Vererbung* lösen. Hierbei erbt der Untertyp nicht nur die Struktur, sondern auch das Verhalten des Obertyps. Außerdem sind

Instanzen des Untertyps überall dort einsetzbar (*substituierbar*), wo Instanzen des Obertyps erforderlich sind.

Abbildung 15.7 zeigt eine Typhierarchie, bei der die Untertypen *Typ2* und *Typ3* jeweils ein weiteres Attribut, nämlich B bzw. C aufweisen. Operationen sind hier gänzlich außer Acht gelassen. Instanzen des Typs *Typ3* gehören auch zur Extension von Typ *Typ2* und von Typ *Typ1*.

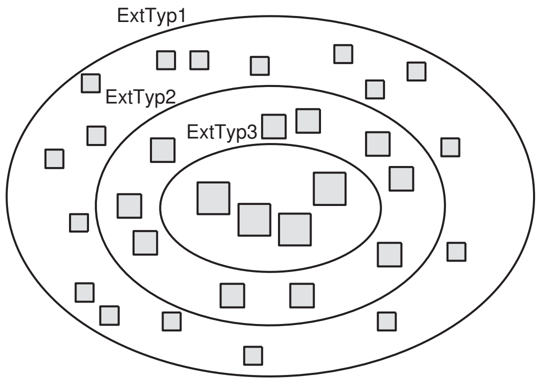


Abbildung 15.8: Darstellung der Subtypisierung

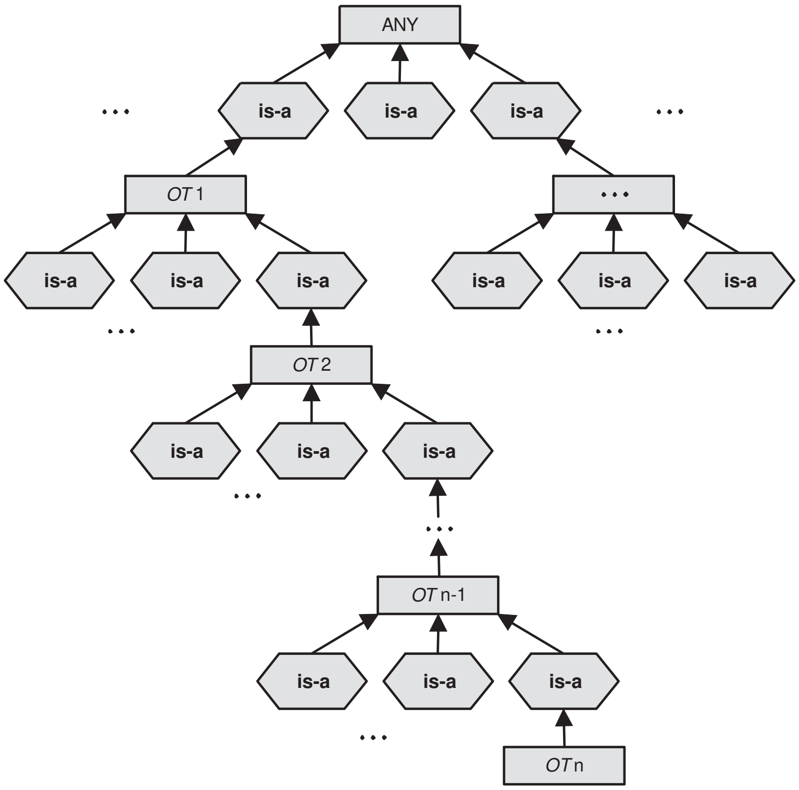


Abbildung 15.9: Abstrakte Typhierachie bei einfacher Vererbung

Abbildung 15.8 zeigt die geschachtelte Anordung der drei Extensionen der Typen *Typ1*, *Typ2* und *Typ3*. Durch die unterschiedliche Kästchengröße soll angedeutet werden, daß 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.

Abbildung 15.9 zeigt eine abstrakte Typhierarchie mit Einfachvererbung. Der Vorteil der Einfachvererbung gegenüber der Mehrfachvererbung besteht darin, daß 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 heißt er *d_Object*.

15.10 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 15.10 zeigt die drei Objekttypen *Angestellte*, *Professoren* und *Assistenten*, wobei die geerbten Eigenschaften in den gepunkteten Ovalen angegeben ist.

Abbildung 15.11 zeigt schematisch die aus der Ober-/Untertyp-Beziehung resultierende Inklusion der Extensionen *AlleProfessoren* und *AlleAssistenten* in der Extension *AlleAngestellte*.

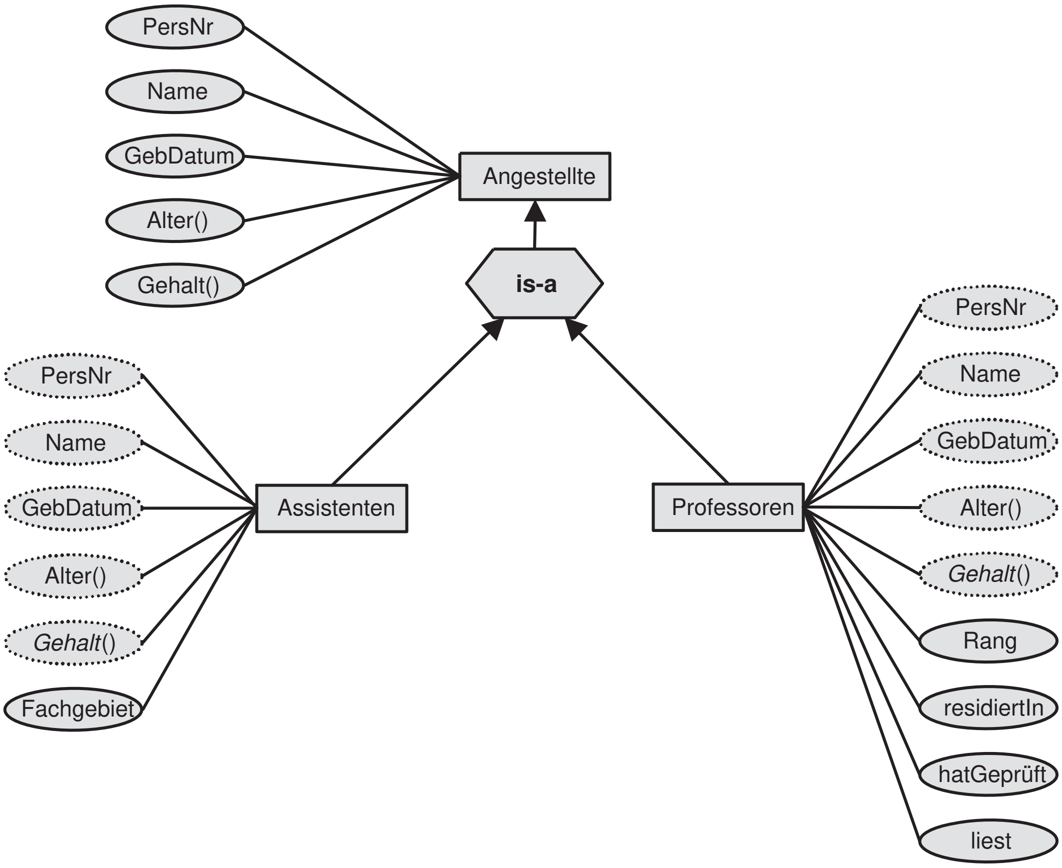


Abbildung 15.10: Vererbung von Eigenschaften

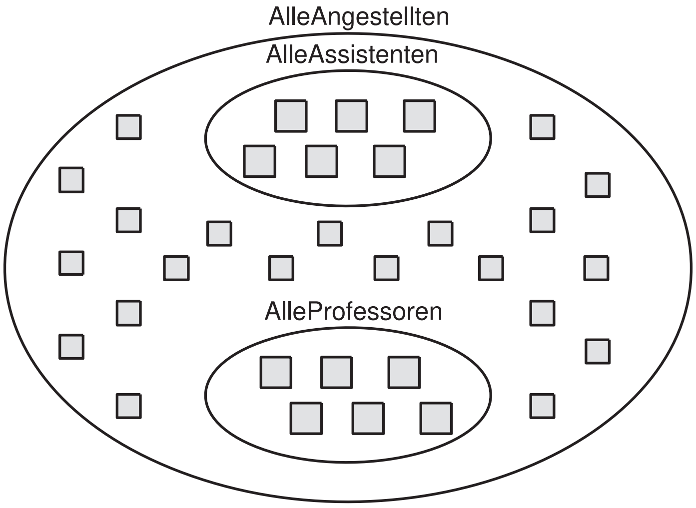


Abbildung 15.11: Visualisierung der Extensionen

15.11 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 DM,
- Assistenten erhalten 2500 + (Alter() - 21) * 125 DM,
- Professoren erhalten 3000 + (Alter() - 21) * 150 DM.

In Abbildung 15.10 ist dies durch den kursiven Schrifttyp der geerbten *Gehalt*-Eigenschaft gekennzeichnet.

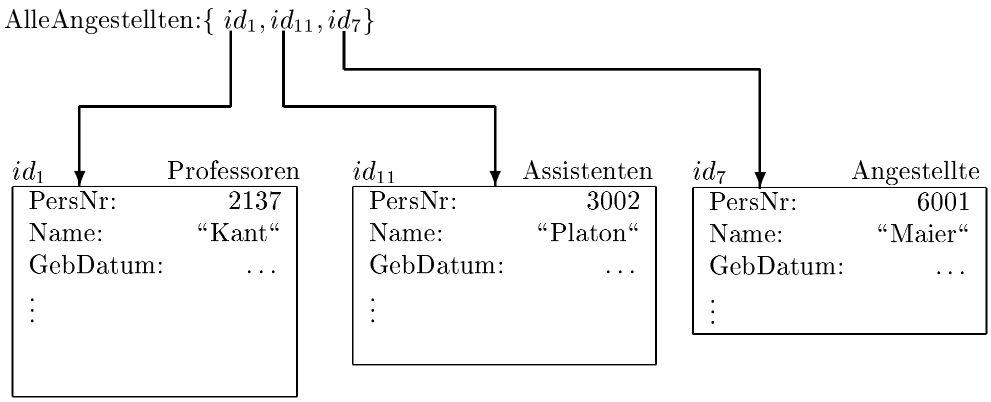


Abbildung 15.12: Die Extension AlleAngestellten mit drei Objekten

Abbildung 15.12 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*).

15.12 Mehrfachvererbung

Bei der Mehrfachvererbung erbt ein Objekttyp die Eigenschaften von mehreren Obertypen. Abbildung 15.13 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.*

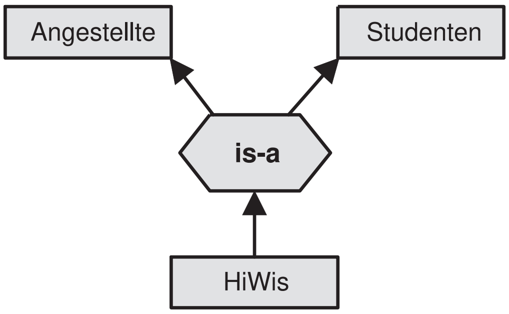


Abbildung 15.13: Beispiel für Mehrfachvererbung

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 muß 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 müßte 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, daß Ableitungen, bei denen solche Konflikte auftreten würden, verboten sind.

15.13 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 = 'C4';
```

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 DM:

```
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 15.14 zeigt die graphische Darstellung des Pfadausdrucks von *Studenten* über *Vorlesungen* zu *Professoren*.

Abbildung 15.14: 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:

```
Vorlesungen (vorlNr: 4711, Titel: "Selber Atmen", SWS: 4, gelesenVon : (
  select p
  from p in AlleProfessoren
  where p.Name = "Sokrates"));
```

15.14 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**

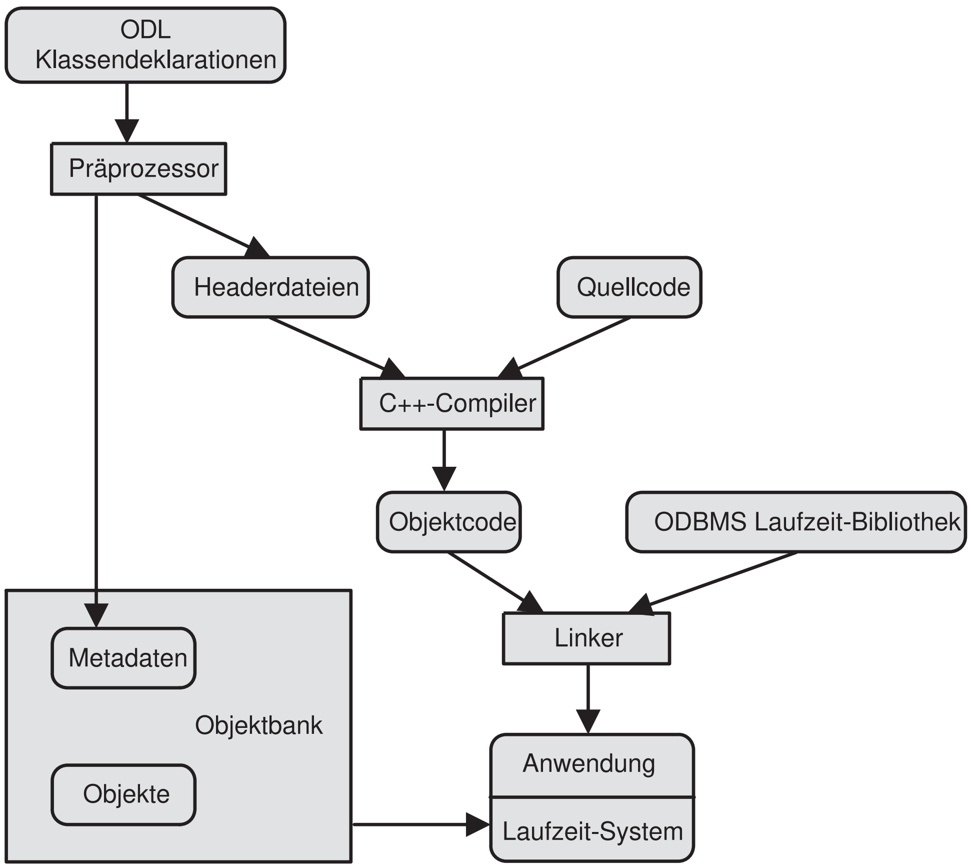


Abbildung 15.15: C++-Einbindung

Die von der ODMG gewählte Einbindung entspricht dem dritten Ansatz. Ihre Realisierung ist in Abbildung 15.15 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, daß 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.

Kapitel 16

Data Warehouse

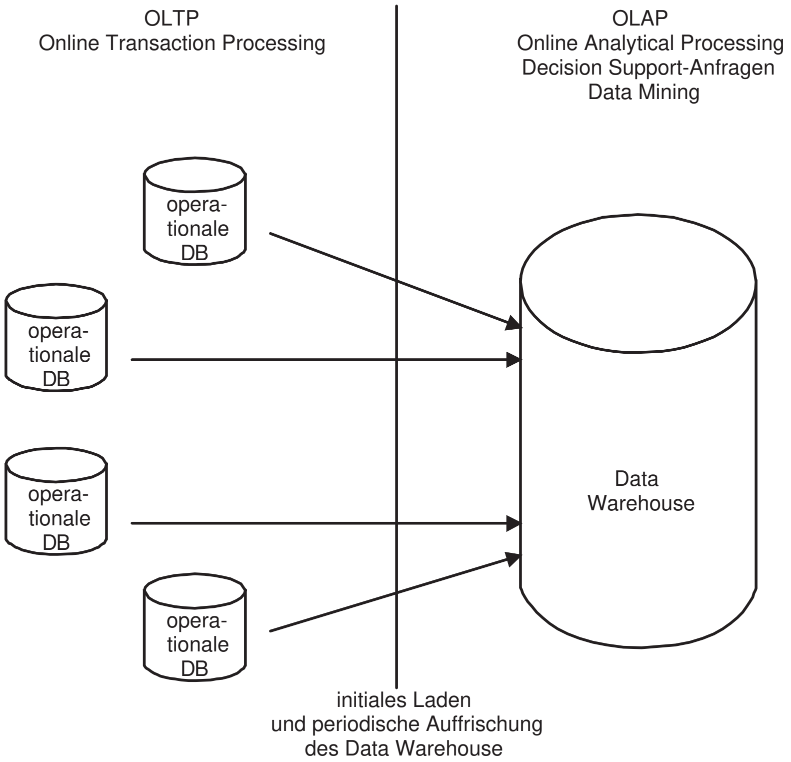


Abbildung 16.1: Zusammenspiel zwischen OLTP und OLAP

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 große 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 16.1 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.

16.1 Datenbankentwurf für Data Warehouse

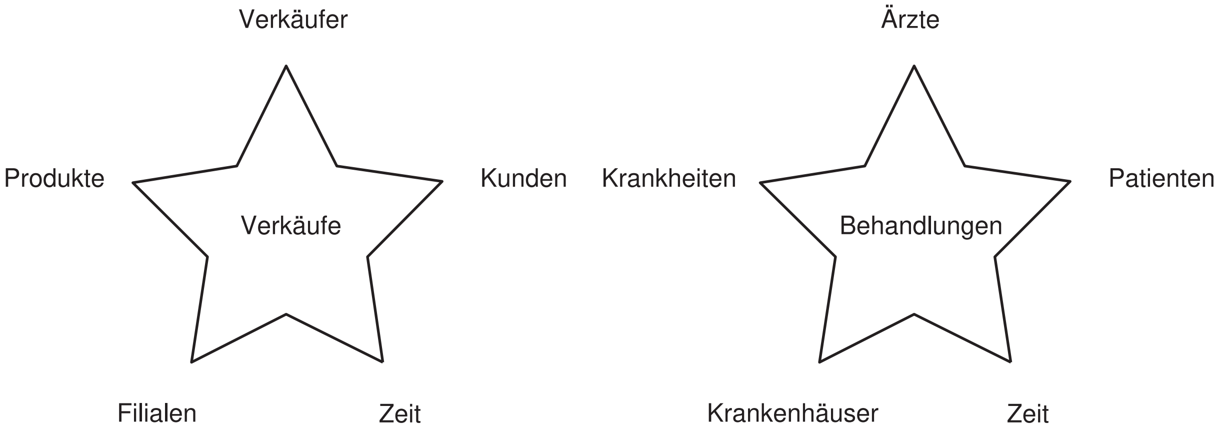


Abbildung 16.2: Sternschemata für Handelsunternehmen und Gesundheitswesen

Als Datenbankschema für Data Warehouse-Anwendungen hat sich das sogenannte *Sternschema* (engl.: *star scheme*) durchgesetzt. Dieses Schema besteht aus einer *Faktentabelle* und mehreren *Dimensionstabellen*. Abbildung 16.2 zeigt die Sternschemata für zwei Beispielanwendungen in einem Handelsunternehmen und im Gesundheitswesen.

Bei dem Handelsunternehmen können in der Faktentabelle *Verkäufe* mehrere Millionen Tupel sein, während die Dimensionstabelle *Produkte* vielleicht 10.000 Einträge und die Dimensions-

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	...
...	...	...	...	...	...

Abbildung 16.3: Ausprägung des Sternschemas in einem Handelsunternehmen

tabelle *Zeit* vielleicht 1.000 Einträge (für die letzen drei Jahre) aufweist. Abbildung 16.3 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 großen (normalisierten) Faktentabelle nicht so sehr ins Gewicht fällt.

16.2 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;
```

16.3 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 Hersteller, Jahr, sum(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;
```

Das Ergebnis wird in der linken Tabelle von Abbildung 16.4 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 Jahr, sum(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;
```

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
Nokai	1995	1.000
Nokai	1996	1.500
Nokai	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
Nokai	4.500

Abbildung 16.4: Analyse der Handy-Verkäufe nach unterschiedlichen Dimensionen

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 Hersteller, sum(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(Anzahl)
from   Verkäufe v, Produkte p
where  v.Produkt = p.ProduktNr
and    p.Produkttyp = 'Handy';
```

Durch eine sogenannte *cross tabulation* können die Ergebnisse in einem n-dimensionalen Spreadsheet zusammengefaßt werden. Abbildung 16.5 zeigt die Ergebisse aller drei Abfragen zu Abbildung 16.4 in einem 2-dimensionalen Datenwürfel *data cube*.

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
Nokai	1.000	1.500	2.000	4.500
Σ	4.500	6.500	8.500	19.500

Abbildung 16.5: Handy-Verkäufe nach Jahr und Hersteller

16.4 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 Abbildung 16.6 erzeugen. Mit dem **null**-Wert wird markiert, daß 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, to_number(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, to_number(null), sum(v.Anzahl)
from  Verkäufe v, Produkte p
where v.Produkt = p.ProduktNr and p.Produkttyp = 'Handy');
```

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. Außerdem 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.

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
Nokai	1995	1.000
Nokai	1996	1.500
Nokai	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

Abbildung 16.6: Materialisierung von Aggregaten in einer Relation

16.5 Der Cube-Operator

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 Abbildung 16.7 gezeigten 3D-Quader; die relationale Repräsentation ist in der rechten Tabelle von Abbildung 16.6 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 große) *Verkäufe*-Relation nur einmal eingelesen werden muß.

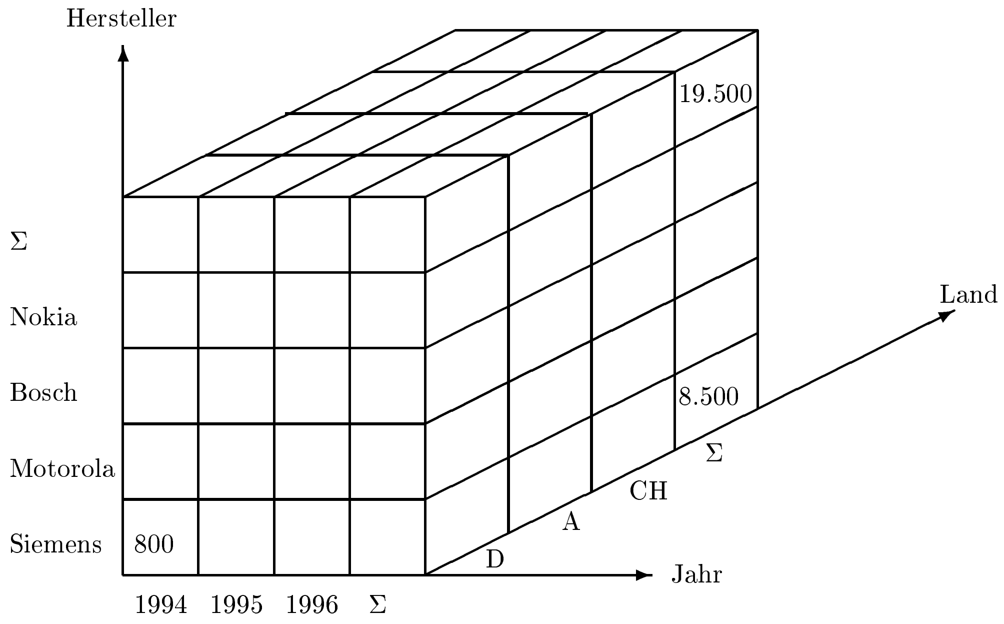


Abbildung 16.7: Würfeldarstellung der Handyverkaufszahlen nach Jahr, Hersteller und Land

16.6 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 maßgeschneiderter Datenstrukturen realisiert. Das heißt, die Daten werden nicht als Tupel in Tabellen gehalten, sondern als Einträge in mehrdimensionalen Arrays. Probleme bereiten dabei dünn besetzte Dimensionen.

16.7 Data Mining

Beim *Data Mining* geht es darum, große Datenmengen nach (bisher unbekannten) Zusammenhängen zu durchsuchen. Man unterscheidet zwei Zielsetzungen bei der Auswertung der Suche:

- Klassifikation von Objekten,
- Finden von Assoziativregeln.

Bei der Klassifikation von Objekten (z. B: Menschen, Aktienkursen, etc.) geht es darum, Vorhersagen über das zukünftige Verhalten auf Basis bekannter Attributwerte zu machen. Abbildung 16.8 zeigt ein Beispiel aus der Versicherungswirtschaft. Für die Risikoabschätzung

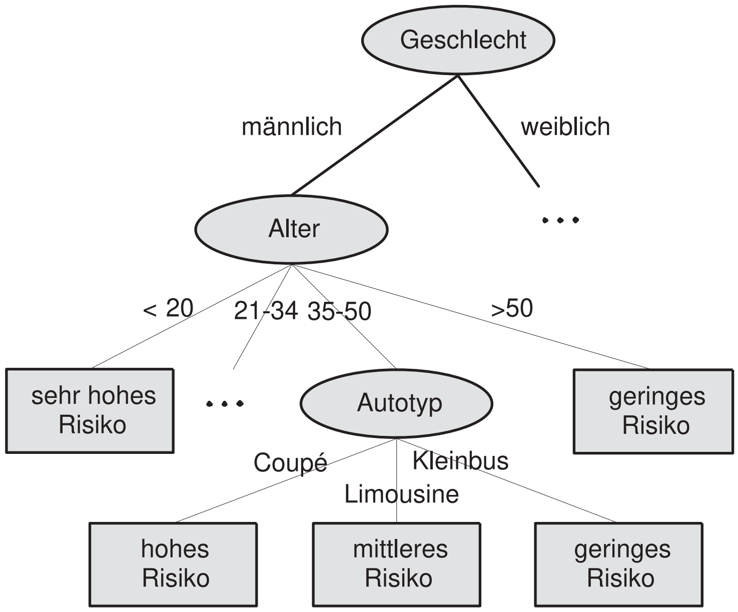


Abbildung 16.8: Klassifikation für Haftpflicht-Risikoabschätzung

könnte man beispielsweise vermuten, daß 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".

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 Kenngrößen:

- **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, daß 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.