Autoren
Prof. Dr. Alfons Kemper
Technische Universität München
Institut für Informatik
Boltzmannstr. 3
D-85748 Garching bei München
E-Mail: alfons.kemper@tum.de

Dr. André Eickler
40468 Düsseldorf

ISBN 978-3-11-044375-2

Library of Congress Cataloging-in-Publication Data
A CIP catalog record for this book has been applied for at the Library of Congress.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

© 2015 Walter de Gruyter GmbH, Berlin/Boston
Einbandabbildung: whilerests/iStock/thinkstock
Druck und Bindung: Hubert und Co. GmbH & Co. KG, Göttingen
♾ Gedruckt auf säurefreiem Papier
Printed in Germany

www.degruyter.com

Inhaltsverzeichnis

Vorwort

Wir drohen derzeit von einer wahren Informationsflut (Stichwort *Big Data*) „überrollt" zu werden und sind auf dem besten Weg in die Informationsgesellschaft. Datenbanksysteme spielen eine immer größere Rolle in Unternehmen, Behörden und anderen Organisationen. Ihre Bedeutung wird durch die zunehmende weltweite Vernetzung – Internet und World Wide Web – noch stärker wachsen. Gleichzeitig wird der systematische Einsatz von Datenbanksystemen wegen der zunehmenden Informationsmenge, der Verteilung der Information auf ein Netz von Datenbankservern, der steigenden Komplexität der Anwendungen und der erhöhten Leistungsanforderungen immer schwieriger – auch wenn sich die Datenbanksystemprodukte weiterentwickeln.

In diesem Buch zur Einführung in Datenbanksysteme haben wir die Lehrinhalte zusammengestellt, die nach unserer Meinung für alle Informatik-nahen Studiengänge an Universitäten oder Fachhochschulen – wie z.B. Informatik, Software-Engineering, Wirtschafts-Informatik, Bio-Informatik etc. – essenziell sind.

Im Vergleich zu anderen Datenbank-Lehrbüchern setzten wir folgende Akzente:

- Es wurde ein durchgängiges Beispiel aus dem Hochschulbereich gewählt, das den Datenbankeinsatz gut illustriert. Dieses Beispiel haben wir bewusst einfach gehalten, damit man es sich gut einprägen kann. Für SQL-Übungen stellen wir auch eine Webschnittstelle unseres an der TUM entwickelten Datenbanksystems HyPer zur Verfügung: `www.hyper-db.de`.

- Das Buch eignet sich auch zum Selbststudium, da wir uns bemüht haben, alle Konzepte an gut verständlichen Beispielen zu veranschaulichen. Eine ideale Ergänzung bietet darüber hinaus das neue *Übungsbuch Datenbanksysteme* von Kemper und Wimmer (2012), das Lösungsvorschläge für die Übungsaufgaben und weitergehende (teilweise multimediale) Lernhilfen enthält.

- Das Buch behandelt nur „moderne" Datenbanksysteme. Sehr ausführlich gehen wir auf das relationale Modell ein, da es derzeit die marktbeherrschende Rolle spielt. Es werden aber auch neuere Entwicklungen, wie Hauptspeicher-Datenbanken, Big Data-Technologien und -Anwendungen, XML und Multi-Tenancy für Cloud-Datenbanken behandelt. Ältere Datenmodelle (die sogenannten satzorientierten Modelle, zu denen das Netzwerkmodell und das hierarchische Modell zählen) haben wir ausgeklammert, da diese Datenbanksysteme in absehbarer Zeit wohl nur noch „historische" Bedeutung haben werden.

- Das Buch behandelt auch Implementierungsaspekte – wie z.B. physische Strukturen für die Datenverwaltung, Realisierungskonzepte für die Mehrbenutzersynchronisation und die Recovery, Optimierungsmethoden zur Anfrageauswertung etc. Auch wenn die wenigsten Informatiker später ein Datenbanksystem „bauen" werden, so meinen wir doch, dass ein tiefgehendes Wissen unabdingbar

ist, um ein Datenbanksystem in der „harten" industriellen Praxis systematisch einsetzen und optimieren zu können.

- Das Buch betont die praktischen Aspekte des Datenbankbereichs – ohne jedoch die theoretischen Grundlagen zu vernachlässigen. Die zugrundeliegende Theorie wird eingeführt, auf Beweise haben wir aber bewusst verzichtet.
- UML wird als objekt-orientierte Datenmodellierungs-Alternative zum ER-Modell eingeführt und die objekt-orientierten und objekt-relationalen Datenbankkonzepte werden detailliert diskutiert.
- Der Einsatz von Datenbanken als Data Warehouse für Decision Support-Anfragen sowie für das Data Mining wird beschrieben.
- Die XML-Datenbanktechnologien werden ausführlich behandelt: XML-Datenmodell, XPath und XQuery als Anfragesprachen sowie XML-basierte Web Services. Weiterhin wird die XML-Unterstützung der kommerziellen relationalen Datenbanksysteme ausführlich diskutiert.
- In dieser zehnten Auflage wurden die Ausführungen aktualisiert und neuere Entwicklungen aufgegriffen. In einigen Kapiteln (insbesondere in den Kapiteln über die relationale Anfragesprache SQL und deren logische Optimierung) wurde die Darstellung vertieft, um neuesten Entwicklungen gerecht zu werden. Insbesondere wurden in Kapitel 17 die SQL Window-Funktionen (oft auch SQL OLAP Funktionen genannt) sowie das Data Mining vertiefend dargestellt.
- Die neue, insbesondere auch von der SAP propagierte Entwicklung der **Hauptspeicher-Datenbanken** wurde in dem dedizierten Kapitel 18 vertiefend behandelt. Diese Systeme machen sich die neuesten Hardwareentwicklungen in Bezug auf Multi-Core-Parallelisierung und auf TeraByte-Level skalierte Hauptspeichergrößen zunutze, um mit einer neuen Datenbank-Architektur dramatische Leistungssteigerungen im Vergleich zu traditionellen Sekundärspeicher-Datenbanken zu erzielen.
- In dem Kapitel **Big Data** wurden die Techniken für die Beherrschbarkeit der Informationsflut des Webs, wie NoSQL Key-Value-Speicher, RDF/SPARQL als Grundlage des *Semantic Web*, allgemeine Graph-Datenrepräsentationen sowie Graphexploration, Information Retrieval und Suchmaschinen-Grundlagen (u.a. PageRank), hochgradig verteilte Datenverarbeitung (MapReduce), Datenströme, und Cloud/Multi-Tenancy Datenbanken aktualisiert und vertieft.
- Zusätzliche Unterlagen zu diesem Buch findet man über unseren Webserver (`http://www-db.in.tum.de`).

Wir haben uns bemüht, die inhaltlichen Abhängigkeiten zwischen den Kapiteln gering zu halten. Deshalb ist es problemlos möglich (und wird von uns an der TU München auch praktiziert), eine schon im Grundstudium enthaltene Einführung in Datenbanksysteme – in der beispielsweise die Grundlagen der konzeptuellen Datenmodellierung, der physischen Datenorganisation, des relationalen Datenmodells und der Anfragesprache SQL, der Datenbanktheorie und der Transaktionsverwaltung

vermittelt werden – aus diesem Buch zu „extrahieren", um das Themengebiet dann im Hauptstudium mit den übrigen Kapiteln zu vervollständigen. Es ist auch möglich, einige der weiterführenden Kapitel in einer Vorlesung zur Datenbankimplementierung oder in einer projektorientierten Datenbankeinsatz-Vorlesung zu „verwerten".

Danksagung Dr. Reinhard Braunandl, Dr. Christian Wiesner, Dr. Jens Claußen, Dr. Carsten Gerlhof, Prof. Donald Kossmann, Dr. Natalija Krivokapić, Dr. Klaus Peithner und Dr. Michael Steinbrunn danken wir für ihre Hilfe bei früheren Auflagen. Dr. Stefan Seltzsam, Dr. Richard Kuntschke und Dr. Martin Wimmer haben wesentlich bei der Ausarbeitung zu den Web-Datenbankschnittstellen geholfen. Dr. Martina-Cezara Albutiu und Herr Stefan Kinauer haben bei der Korrektur geholfen. Meinem Kollegen, Prof. Thomas Neumann, sowie den Doktoranden am Lehrstuhl, die im HyPer-Projekt forschen bzw. geforscht haben (Dr. Martina-Cezara Albutiu, Dr. Stefan Aulbach, Robert Brunel, Jan Finis, Dr. Florian Funke, Harald Lang, Viktor Leis, Dr. Henrik Mühe, Tobias Mühlbauer, Dr. Angelika Reiser, Wolf Rödiger, Dr. Michael Seibold, Manuel Then) danke ich für die Zusammenarbeit – sie finden einige ihrer Forschungsergebnisse in den neueren Kapiteln dieser zehnten Auflage wieder.

Wir haben von etlichen „externen" Lesern Anregungen bekommen. Besonders hilfreich waren die Hinweise von Prof. Stefan Brass, Prof. Sven Helmer, Prof. Volker Linnemann, Prof. Guido Moerkotte, Prof. Reinhard Pichler, Prof. Erhard Rahm, Prof. Stefanie Scherzinger, Frau Katrin Seyr, Prof. Bernhard Thalheim und Prof. Rainer Weber. Dr. Michael Ley danken wir weiterhin für seinen phantastischen Bibliographie-Server `http://dblp.uni-trier.de/`.

München, im August 2015 *Alfons Kemper*

1. Einleitung und Übersicht

Der Zugriff auf und die Verwaltung von *Information* spielt eine immer wichtiger werdende Rolle in der heutigen Gesellschaft – sei es für Unternehmen, Politiker, Wissenschaftler, Verwaltungen, etc. Es wird geschätzt, dass sich die „Informationsmenge" derzeit alle 5 Jahre verdoppelt – zumindest trifft dies für die in Büchern abgelegte Information nach Statistiken der amerikanischen Library of Congress zu. Während in früheren Zeiten der Großteil der Information auf Papier abgelegt war, werden wir heute von einer elektronischen Informationsflut „überrollt". Deshalb gewinnen *Datenbankverwaltungssysteme* (engl. *database management systems*, abgek. *DBMS*) eine immer größere Bedeutung. Heute findet sich kaum noch eine größere Organisation oder ein größeres Unternehmen, das nicht ein DBMS für die Informationsverwaltung einsetzt. Man denke etwa an Banken, Versicherungen, Flugunternehmen und Universitätsverwaltungen (um unsere Beispielanwendung dieses Buchs schon mal zu erwähnen).

Ein Datenbankverwaltungssystem besteht aus einer Menge von *Daten* und den zur Datenverarbeitung notwendigen Programmen:

- Die gespeicherten Daten werden oft als *Datenbasis* bezeichnet. Die Datenbasis enthält die miteinander in Beziehung stehenden Informationseinheiten, die zur Kontrolle und Steuerung eines Aufgabenbereichs (evtl. eines ganzen Unternehmens) notwendig sind.
- Die Gesamtheit der Programme zum Zugriff auf die Datenbasis, zur Kontrolle der Konsistenz und zur Modifikation der Daten wird als *Datenbankverwaltungssystem* bezeichnet.

Oft werden diese Komponenten aber auch weniger scharf getrennt, so dass man mit Datenbankverwaltungssystemen (oder kürzer Datenbanksystemen) sowohl die Datenbasis als auch die Verwaltungsprozesse der Datenbasis meint.

1.1 Motivation für den Einsatz eines DBMS

Es gibt heutzutage in den meisten Unternehmen und Organisationen keine Alternative mehr zum Einsatz eines DBMS für die Informationsverarbeitung. Wir wollen uns dies anhand der Probleme verdeutlichen, die ohne Nutzung eines einheitlichen, die gesamte Informationsverarbeitung abdeckenden DBMS, auftreten würden. Die im allgemeinen auf vielfältige Art miteinander in Beziehung stehenden Daten müssten dann entweder auf Papier (Karteikästen, Aktenordner, etc.) oder in isolierten Computer-Dateien abgelegt werden. Dies führt zu folgenden schwerwiegenden Problemen:

Redundanz und Inkonsistenz Wenn Daten in isolierten Dateien (oder andersartigen isolierten Archiven) gehalten werden, müssen dieselben Informationen bezüglich eines Anwendungsobjekts oft mehrfach, d.h. redundant, gespeichert werden.

Man denke etwa an die Adressinformation der Studenten einer Universität. Diese Information wird sicherlich in der Studentenverwaltung, aber auch in den jeweiligen Fakultäten benötigt. Bei Änderungen kann es dann zu Inkonsistenzen führen, wenn nur eine Kopie der Daten geändert wird, die andere aber noch im veralteten Zustand beibehalten wird. In einem globalen, integrierten DBMS wird diese Art von unkontrollierter Redundanz vermieden.

Beschränkte Zugriffsmöglichkeiten Es ist schwer, wenn nicht sogar unmöglich, die in isolierten Dateien abgelegten Daten miteinander zu „verknüpfen", d.h. Information aus einer Datei mit anderen logisch verwandten Daten aus einer anderen Datei zu verknüpfen. Bei einem homogenen integrierten DBMS wird die gesamte Information einer Organisation einheitlich modelliert (wir sagen in demselben *Datenmodell*), so dass sich diese Daten sehr flexibel miteinander verknüpfen lassen.

Probleme des Mehrbenutzerbetriebs Die heutigen Dateisysteme bieten entweder gar keine oder nur sehr rudimentäre Kontrollmechanismen für den Mehrbenutzerbetrieb. Daten werden aber i.A. von sehr vielen Anwendern innerhalb und außerhalb der jeweiligen Organisation genutzt – als Beispiel sei ein Flugreservierungssystem genannt. Bei unkontrolliertem Zugriff kann es sehr leicht zu (äußerst) unerwünschten Anomalien kommen. Man denke etwa an das gleichzeitige unkontrollierte Editieren derselben Datei durch zwei Benutzer. Dabei kann es leicht vorkommen, dass die Änderungen des einen Benutzers von dem Benutzer, der die Datei zuletzt zurückschreibt, überschrieben werden. Dieses Phänomen nennt man im Englischen „lost update".

Datenbankverwaltungssysteme bieten eine Mehrbenutzerkontrolle, die solche und noch andere unerwünschte Anomalien des Mehrbenutzerbetriebs ausschließen.

Verlust von Daten Wenn Daten in isolierten Dateien gehalten werden, wird die Wiederherstellung eines konsistenten – d.h. eines gemäß der realen Welt gültigen – Zustands der Gesamtinformationsmenge im Fehlerfall sehr schwierig. Im Allgemeinen bieten Dateisysteme bestenfalls die Möglichkeit einer periodisch durchgeführten Sicherung der Dateien. Datenverluste, die während der Bearbeitung von Dateien oder nach der letzten Sicherungskopie auftreten, sind i.A. nicht auszuschließen.

Datenbankverwaltungssysteme besitzen eine ausgefeilte Recoverykomponente, die den Benutzer für alle vorhersehbaren Fehlerfälle vor Datenverlust schützen soll.

Integritätsverletzung Je nach Anwendungsgebiet gibt es vielfältigste, sich global über mehrere Informationseinheiten erstreckende Integritätsbedingungen. Man denke im Universitätsbereich etwa an die Bedingung, dass Studenten erst die Pflichtseminare abgeschlossen haben müssen, bevor sie zur Prüfung zugelassen werden dürfen. Oder dass dieselbe Prüfung maximal zweimal wiederholt werden darf. Die Einhaltung derartiger Integritätsbedingungen ist bei der isolierten Speicherung der Informationseinheiten in verschiedenen Dateien sehr schwierig, da man zur Kontrolle Daten aus unterschiedlichen Dateien verknüpfen muß. Außerdem will man im Allgemeinen nicht nur die Konsistenzbedingungen überprüfen, sondern die Einhaltung erzwingen, d.h. bestimmte Datenverarbeitungsvorgänge sollen vom System „abgelehnt" werden, falls sie zu einer Verletzung der Integrität führen werden. In

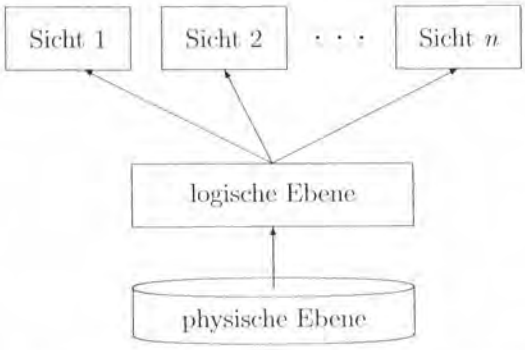


Abbildung 1.1: Drei Abstraktionsebenen eines Datenbanksystems

Datenbankverwaltungssystemen werden Transaktionen (das sind die aus Benutzersicht atomaren Verarbeitungsvorgänge) nur dann vollzogen, wenn sie die Datenbasis in einen konsistenten Zustand überführen.

Sicherheitsprobleme Nicht alle Benutzer sollten Zugriff auf die gesamten gespeicherten Daten haben. Und erst recht sollten nur bestimmte ausgewählte Benutzer das *Privileg* haben, Daten zu ändern. Man denke etwa an die Information, die zu den Professoren einer Universität abgespeichert ist. Dann ist denkbar, dass man relativ vielen Benutzern Zugriff auf die Information bezüglich Rang, Raum, Telefon, gelesene Vorlesungen, etc. gewährt. Die Information bezüglich Gehalt und abgenommener Prüfungen ist andererseits vor dem Zugriff durch die meisten Benutzer zu schützen.

Datenbankverwaltungssysteme bieten die Möglichkeit, die Zugriffsrechte sehr flexibel an einzelne Benutzer bzw. Benutzergruppen zu übertragen. Dabei ist es auch möglich, Informationsteile (z.B. Gehalt) bezüglich bestimmter Objekte (z.B. Professoren) auszublenden, während andere Teile (z.B. Telefonnummer) sichtbar bleiben.

Hohe Entwicklungskosten In vielen Fällen muss für die Entwicklung eines neuen Anwendungsprogrammes praktisch „das Rad neu erfunden" werden. Jedesmal muss sich der Anwendungsprogrammierer zusätzlich zu Fragen der Dateiverwaltung mit zumindest einer Teilmenge der obigen Probleme auseinandersetzen. DBMS stellen eine deutlich komfortablere Schnittstelle dar, die die Entwicklungszeiten (und damit die Kosten) für neue Anwendungen verkürzt und die Fehleranfälligkeit reduziert.

1.2 Datenabstraktion

Man unterscheidet drei Abstraktionsebenen im Datenbanksystem (Abbildung 1.1):

1. *Die physische Ebene*: Auf dieser Ebene wird festgelegt, wie die Daten gespeichert sind. Im Allgemeinen sind die Daten auf dem Hintergrundspeicher (meistens als Plattenspeicher realisiert) abgelegt.

2. *Die logische Ebene*: Auf der logischen Ebene wird in einem sogenannten *Datenbankschema* festgelegt, welche Daten abgespeichert sind.

3. *Die Sichten*: Während das Datenbankschema der logischen Ebene ein integriertes Modell der gesamten Informationsmenge des jeweiligen Anwendungsbereichs (z.B. des gesamten Unternehmens) darstellt, werden in den Sichten Teilmengen der Information bereitgestellt. Die Sichten sind auf die Bedürfnisse der jeweiligen Benutzer bzw. Benutzergruppen zugeschnitten. Mögliche Benutzergruppen in den Universitäten wären etwa die Studenten, die Professoren, die Hausmeister, etc.

Die physische Ebene ist für den „normalen" Benutzer eines Datenbanksystems nicht relevant. Auf dieser Ebene werden die Speicherstrukturen und eventuell Indexstrukturen für das schnelle Auffinden von Daten festgelegt. Die Pflege der physischen Ebene eines Datenbanksystems obliegt einem Systemprogrammierer bzw. einer Systemprogrammiererin – im Datenbankjargon als *Datenbankadministrator/in* (DBA) bezeichnet. Änderungen der physischen Speicherstruktur werden nur zum Zweck der Leistungssteigerung des DBMS vollzogen. Will man das Informationsmodell ändern – wie z.B. zusätzliche Daten einbeziehen oder zusätzliche Beziehungen zwischen Daten modellieren – muss die logische Ebene, also das Datenbankschema geändert werden. Das Datenbankschema kann man sich als eine Menge von Typdefinitionen – wie z.B. Record-Typen in Pascal – vorstellen. Diese Typdefinitionen legen die logische Struktur der Dateneinheiten fest. Auf der Ebene der Sichten können die im Datenbankschema (auf der logischen Ebene des DBMS) festgelegten Strukturen auf die besonderen Bedürfnisse bestimmter Anwender(gruppen) zugeschnitten werden. Zum einen wollen Anwender i.A. nur einen Ausschnitt des gesamten Informationsmodells zu sehen bekommen. Zum anderen können bestimmte kritische Informationsteile in den Sichten ausgeblendet werden, um dadurch den Datenschutz zu gewährleisten.

1.3 Datenunabhängigkeit

Die drei Ebenen eines DBMS gewährleisten einen bestimmten Grad der *Datenunabhängigkeit*. Dies ist analog zum Konzept der abstrakten Datentypen (ADTs) in Programmiersprachen. Durch eine wohldefinierte Schnittstelle wird die „darunterliegende" Implementierung verdeckt, so dass man – bei Beibehaltung der Schnittstelle – die Realisierung variieren kann, ohne dass die Benutzer der Schnittstelle davon in Mitleidenschaft gezogen werden. Aufgrund der drei Schichten ergeben sich zwei Stufen der Datenunabhängigkeit im DBMS.

- *Physische Datenunabhängigkeit*: Die Modifikation der physischen Speicherstruktur belässt die logische Ebene (also das Datenbankschema) invariant. Z.B. erlauben fast alle Datenbanksysteme das nachträgliche Anlegen eines Indexes, um die Datenobjekte schneller finden zu können. Dies darf keinen Einfluss auf bereits existierende Anwendungen auf der logischen Ebene haben – außer natürlich hinsichtlich der Effizienz.

- *Logische Datenunabhängigkeit*: In den Anwendungen wird (natürlich) Bezug auf die logische Struktur der Datenbasis genommen: Es werden Mengen von Datenobjekten nach einem Namen angesprochen, die Datenobjekte haben „benannte" Eigenschaften, etc. Man denke etwa an eine Anfrage, in der die Professoren ermittelt werden, die den Rang „C2" haben. In einer solchen Anfrage

wird vorausgesetzt, dass es eine Menge von Professoren gibt und dass die Datenobjekte, die Professoren repräsentieren, eine Eigenschaft (Attribut, Feld) namens *Rang* haben. Bei Änderungen der logischen Ebene (also des Datenbankschemas) könnte z.B. diese Eigenschaften umbenannt werden in, sagen wir, *Gehaltsstufe*. In einer Sichtendefinition kann man solche kleineren Änderungen vor den Anwendern verbergen. Dadurch wird zu einem gewissen Grad eine logische Datenunabhängigkeit erzielt.

Die heutigen Datenbanksysteme erfüllen zumeist die physische Datenunabhängigkeit. Die logische Datenunabhängigkeit kann schon rein konzeptuell nur für einfachste Modifikationen des Datenbankschemas gewährleistet werden.

1.4 Datenmodelle

Datenbankverwaltungssysteme basieren auf einem *Datenmodell*, das sozusagen die Infrastruktur für die Modellierung der realen Welt zur Verfügung stellt. Das Datenmodell legt die Modellierungskonstrukte fest, mittels derer man ein computerisiertes Informationsabbild der realen Welt (bzw. des relevanten Ausschnitts) generieren kann. Es beinhaltet die Möglichkeit zur

- Beschreibung der Datenobjekte und zur
- Festlegung der anwendbaren Operatoren und deren Wirkung.

Das Datenmodell ist somit analog zu einer Programmiersprache: Es legt die generischen Strukturen und Operatoren fest, die man zur Modellierung einer bestimmten Anwendung ausnutzen kann. Eine Programmiersprache legt die Typkonstruktoren und Sprachkonstrukte fest, mit deren Hilfe man spezifische Anwendungsprogramme realisiert.

Das Datenmodell besteht demnach aus zwei Teilsprachen:

1. der *Datendefinitionssprache* (engl. *Data Definition Language*, DDL) und
2. der *Datenmanipulationssprache* (engl. *Data Manipulation Language*, DML).

Die DDL wird benutzt, um die Struktur der abzuspeichernden Datenobjekte zu beschreiben. Dabei werden gleichartige Datenobjekte durch ein gemeinsames Schema (analog zu einem Datentyp in Programmiersprachen) beschrieben. Die Strukturbeschreibung aller Datenobjekte des betrachteten Anwendungsbereichs nennt man das *Datenbankschema*.

Die Datenmanipulationssprache (DML) besteht aus

- der *Anfragesprache* (engl. *Query Language*) und
- der „eigentlichen" Datenmanipulationssprache zur Änderung von abgespeicherten Datenobjekten, zum Einfügen von Daten und zum Löschen von gespeicherten Daten.

Die DML (einschließlich der Anfragesprache) kann in zwei unterschiedlichen Arten genutzt werden:

- *interaktiv*, indem DML-Kommandos direkt am Arbeitsplatzrechner (oder Terminal) eingegeben werden, oder
- in einem Programm einer höheren Programmiersprache, das „eingebettete" DML-Kommandos enthält.

1.5 Datenbankschema und Ausprägung

Man muss sehr klar zwischen *Datenbankschema* und *Datenbankausprägung* unterscheiden: Das Datenbankschema legt die Struktur[1] der abspeicherbaren Datenobjekte fest. Das Schema sagt also noch nichts über die individuellen Datenobjekte aus. Deshalb kann man das Datenbankschema auch als *Metadaten* – also Daten über Daten – verstehen.

Unter der Datenbankausprägung versteht man demgegenüber den momentan gültigen (also abgespeicherten) Zustand der Datenbasis. Die Datenbankausprägung muss also den im Schema festgelegten Strukturbeschreibungen „gehorchen". Manchmal spricht man in diesem Zusammenhang auch von der *intensionalen* (Schema) und der *extensionalen* (Ausprägung) Ebene einer Datenbank.

Im Allgemeinen geht man davon aus, dass sich das einmal festgelegte Datenbankschema sehr selten ändert, wohingegen die Datenbankausprägung einer laufenden Modifikation unterliegt. Man denke etwa an ein Flugbuchungssystem: Jede Reservierung entspricht einer Änderung der Datenbankausprägung. Änderungen am Schema werden oft auch als „Schemaevolution" bezeichnet. Man beachte, dass Schemaänderungen schwerwiegende Folgen haben können: Die bereits abgespeicherten Datenobjekte können nach einer Schemaänderung eine – gemäß dem neuen Datenbankschema – inkonsistente Struktur aufweisen.

1.6 Einordnung der Datenmodelle

In Abbildung 1.2 sind die grundlegendsten Phasen der Datenmodellierung gezeigt in Kapitel 2 werden wir detaillierter darauf eingehen.

1.6.1 Modelle des konzeptuellen Entwurfs

Man beginnt beim Datenbankentwurf mit der Abgrenzung eines Teils der „realen Welt", um den Ausschnitt (die sogenannte *Miniwelt*) zu bestimmen, der in der Datenbank modelliert werden soll. Diese Miniwelt wird dann konzeptuell modelliert. Für die konzeptuelle Modellierung gibt es mehrere mögliche Datenmodelle:

- Entity-Relationship-Modell, auch Gegenstand-Beziehungs-Modell genannt,
- semantisches Datenmodell und
- objektorientierte Entwurfsmodelle, wie UML (siehe Kapitel 2 und 13).

[1] Im objektorientierten Datenmodell legt das Schema zusätzlich auch das Verhalten (also die Operationen) der Datenobjekte fest.

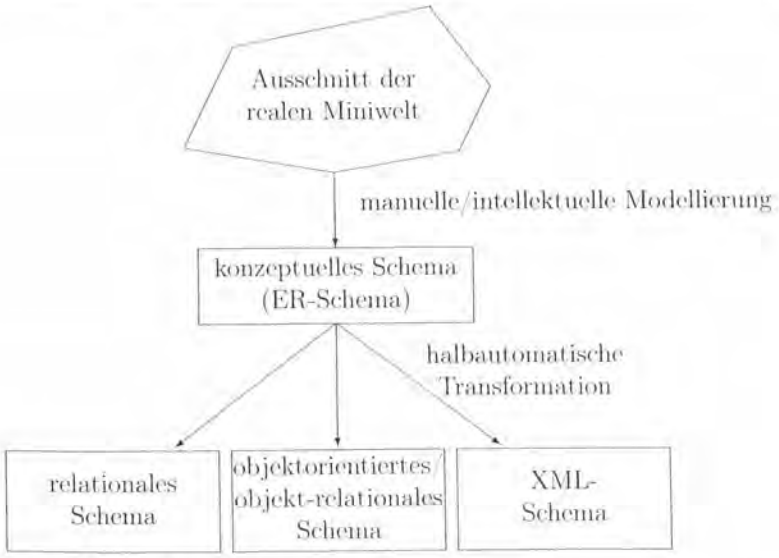


Abbildung 1.2: Übersicht der Datenmodellierung

Das mit Abstand am häufigsten benutzte Modell für den konzeptuellen Entwurf ist das Entity-Relationship-Modell. Folglich werden wir uns im nächsten Kapitel detaillierter damit beschäftigen. In der konzeptuellen Entwurfsphase werden die in der realen Welt vorkommenden Konzepte in Gegenstandsmengen und Beziehungen zwischen diesen Gegenstandsmengen strukturiert.

Abbildung 1.3 zeigt diese „intellektuelle" Aufgabe für einen ganz kleinen Ausschnitt der Universitätswelt. Es werden die Gegenstandsmengen *Studenten*, *Professoren* und *Vorlesungen* ermittelt. Weiterhin werden die Beziehungen *hören* (zwischen *Vorlesungen* und *Studenten*) und *lesen* (zwischen *Vorlesungen* und *Professoren*) bestimmt. Der untere Teil der Abbildung stellt jetzt schon ein (stark vereinfachtes) konzeptuelles Schema in der graphischen Beschreibungssprache des Entity-Relationship-Modells dar.

Die konzeptuellen Datenmodelle verfügen im Allgemeinen nur über eine DDL und haben keine Datenmanipulationssprache, da sie nur die Struktur der Daten beschreiben. Sie verzichten auf die Abbildung von individuellen Datenobjekten, d.h. es werden keine Datenbankausprägungen erzeugt. Deshalb benötigen sie natürlich auch keine Datenmodifikationssprache (DML).

1.6.2 Logische (Implementations-)Datenmodelle

Das konzeptuelle Schema ist aber in dieser Form meist nicht als Implementationsschema geeignet. Die Datenmodelle für den konzeptuellen Entwurf sind i.A. reine Beschreibungsmodelle mit graphischer Notation und sehr reichhaltigen Modellierungskonstrukten, um die Gesetzmäßigkeiten der realen Welt möglichst anschaulich abbilden zu können.

Die logische Ebene eines Datenbankverwaltungssystems wird von einem der folgenden Datenmodelle gebildet:

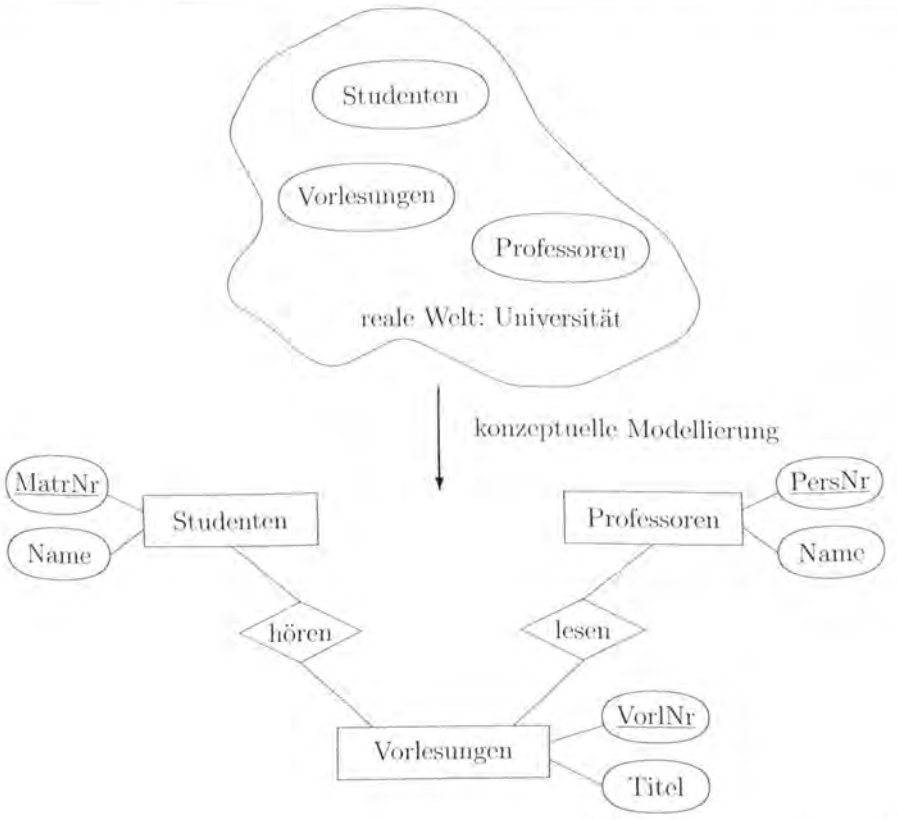


Abbildung 1.3: Konzeptuelle Modellierung einer (sehr kleinen) Beispielanwendung

- Netzwerkmodell, oder hierarchisches Datenmodell,
- relationales Datenmodell (Kapitel 3),
- objektorientiertes und objekt-relationales Datenmodell (Kapitel 13 und 14),
- deduktives Datenmodell (Kapitel 15),
- XML (Kapitel 20).

Die beiden erstgenannten Datenmodelle – das Netzwerk- und das hierarchische Datenmodell – werden oft als *satzorientierte* Datenmodelle zusammengefasst. Sie haben heute fast nur noch historische Bedeutung: Es gibt aber etliche Altinstallationen von Datenbanksystemen, die noch auf diesen Datenmodellen basieren. Eine Transition zu einem „modernen" Datenmodell ist natürlich mit hohen Kosten verbunden, da die installierten Datenbanken i.A. enorme Volumina angenommen haben. Brodie und Stonebraker (1995) behandeln dieses Problem der „legacy systems" – frei übersetzt: Altlasten. Datenbankverwaltungssysteme, die noch auf diesen satzorientierten Datenmodellen beruhen, sind beispielsweise *IMS* von IBM basierend auf dem hierarchischen Modell und *UDS* von Siemens basierend auf dem Netzwerkmodell.

Die relationalen Datenbanksysteme sind heute marktbeherrschend und bilden folglich den Schwerpunkt in diesem Buch. Wir haben aber auch den deduktiven und den objektorientierten Datenbanken je ein Kapitel gewidmet. Das deduktive Datenmodell stellt eine Erweiterung des relationalen Modells um eine Regel- oder Deduktionskomponente dar. Formal gesehen basiert das Modell auf der Logik erster Stufe und wird deshalb manchmal auch Logik-Datenmodell genannt. Die objektorientierten Datenbanksysteme werden heute vielfach als die nächste Generation der Datenbanktechnologie angesehen. Um dieser Herausforderung zu begegnen, haben die Hersteller relationaler Systeme versucht, einige Konzepte des objektorientierten Datenmodells ins relationale Modell zu übertragen. Aus diesem Grund kann man davon ausgehen, dass die Datenbanksysteme der nächsten Generation objektorientierte Modellierungskonstrukte beinhalten.

Zunehmend werden Daten heutzutage im XML-Format modelliert, wobei XML sich besonders als Datenaustauschformat zwischen heterogenen, verteilten Anwendungen durchsetzt. Auch für XML-basierte Daten bieten kommerzielle relationale Datenbanksysteme schon weitreichende Unterstützung.

Wir wollen hier noch kurz die relationale Darstellung eines Teils unseres konzeptuellen Universitäts-Schemas (Abbildung 1.3) zeigen. Nachfolgend sind die drei Relationen *Studenten*, *hören* und *Vorlesungen* gezeigt:

Studenten	
MatrNr	Name
26120	Fichte
25403	Jonas
...	...

hören	
MatrNr	VorlNr
25403	5022
26120	5001
...	...

Vorlesungen	
VorlNr	Titel
5001	Grundzüge
5022	Glaube und Wissen
...	...

Relationen kann man sich als „flache" *Tabellen* (engl. *table*) vorstellen. Die Zeilen entsprechen den Datenobjekten der realen Welt – hier also *Studenten* und *Vorlesungen*. Die Spalten geben die Eigenschaften der Datenobjekte an – z.B. die *MatrNr* (Matrikelnummer) und den *Namen* der Studenten. Die Relation *hören* nimmt eine gewisse Sonderstellung ein: Sie modelliert die Beziehung zwischen *Studenten* und *Vorlesungen*. Die Zeile [25403, 5022] gibt dabei zum Beispiel an, dass der Student namens „Jonas" mit der *MatrNr* 25403 als Hörer an der Vorlesung mit dem Titel „Glaube und Wissen" und der *VorlNr* (Vorlesungsnummer) 5022 teilnimmt.

Diese Tabellen stellen die logische Sicht der Datenbankbenutzer dar. Es gibt eine standardisierte DML namens SQL für die Manipulation und Abfrage dieser Tabellen. Wir wollen dem Leser hier anhand zweier Beispiele nur einen kleinen Vorgeschmack auf diese Sprache geben. Als erstes wollen wir die Namen der Studenten ermitteln, die an der Vorlesung „Grundzüge" teilnehmen:

```
select Name
from Studenten, hören, Vorlesungen
where Studenten.MatrNr = hören.MatrNr and
      hören.VorlNr = Vorlesungen.VorlNr and
      Vorlesungen.Titel = 'Grundzüge';
```

Bei dieser Anfrage werden die Inhalte der Relationen *Studenten*, *hören* und *Vorlesungen* kombiniert (verknüpft), um die gewünschte Information aus der Datenbank zu extrahieren. Für eine genauere Erläuterung verweisen wir auf Kapitel 4.

Im nächsten Beispiel wollen wir den Titel der Vorlesung mit der *VorlNr* 5001 in „Grundzüge der Logik" ändern:

```
update Vorlesungen
    set Titel = 'Grundzüge der Logik'
    where VorlNr = 5001;
```

In der **where**-Klausel wird also die zu ändernde Zeile bestimmt und in der **set**-Klausel wird der neue Wert der *Titel*-Spalte angegeben. Man kann in SQL auch mehrere Zeilen und/oder Spalten gleichzeitig ändern.

Wir sollten hier nochmals den Zusammenhang zwischen Datenbank-Schema und -Ausprägung betonen. Die Zeilen der Relationen stellen die Datenbankausprägung dar, die dem momentanen Zustand der Datenbasis entspricht. Die Struktur der Tabellen – also die Anzahl der Spalten, die Benennung der Spalten, die zulässigen Wertemengen für die Spalten, etc. – stellt das Datenbankschema dar, das nur sehr selten (wenn überhaupt) geändert wird.

1.7 Architekturübersicht eines DBMS

Abbildung 1.4 zeigt eine stark vereinfachte Darstellung der Architektur eines Datenbankverwaltungssystems. Im oberen Bereich befindet sich die Benutzerschnittstelle. Je nach Erfahrung und Verantwortlichkeit greifen unterschiedliche Benutzergruppen auf unterschiedliche Schnittstellen zu:

- Für häufig erledigte und immer ähnliche Aufgaben werden speziell abgestimmte Anwendungsprogramme zur Verfügung gestellt. Diese Anwendungsprogramme sind leichter zu erlernen und effizienter zu bedienen als eine komplette Anfragesprache. Oft sind diese Anwendungssysteme über eine Menü-gesteuerte Benutzerschnittstelle ausführbar – man denke etwa an ein Flugreservierungssystem, das von den Angestellten eines Reisebüros bedient wird.

- Fortgeschrittene Benutzer mit ständig wechselnden Aufgaben können interaktiv Anfragen in einer flexiblen Anfragesprache (wie SQL) eingeben.

- Anwendungsprogrammierer können durch „Einbettung" von Elementen der Anfragesprache in eine Programmiersprache besonders komplexe Datenverarbeitungsanforderungen erfüllen oder weniger geschulten Benutzern einfach zu bedienende Anwendungsprogramme zur Verfügung stellen. Dieser Mechanismus wird in Kapitel 4 besprochen.

- Die Schnittstelle für die Datenbankadministration ermöglicht unter anderem die Manipulation des Schemas und das Anlegen von Benutzerkennungen.

Anforderungen von Daten durch die Benutzer werden zunächst vom DML-Compiler untersucht und in eine für die Anfragebearbeitung verständliche Form gebracht. Die Anfragebearbeitung (siehe Kapitel 8) untersucht, wie die Anforderung effizient erfüllt werden kann und wandelt sie in Unterprogrammaufrufe des Datenbankmanagers um. Der Datenbankmanager ist das Kernstück des DBMS: Hier werden die Anfragen ausgeführt. Er bildet die Schnittstelle zur Dateiverwaltung.

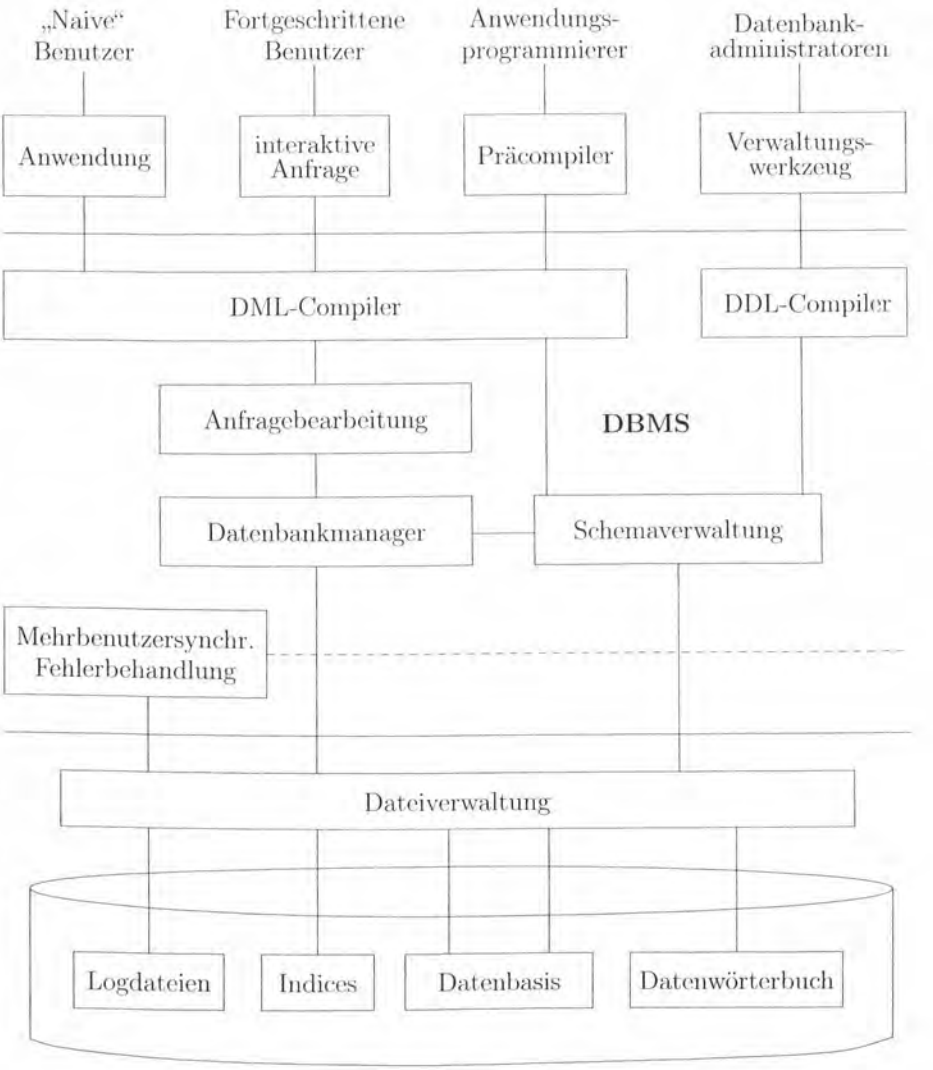


Abbildung 1.4: Architekturübersicht eines DBMS

Schemamanipulationen durch den DBA werden vom DDL-Compiler analysiert und in Metadaten übersetzt. Diese Metadaten werden von der Schemaverwaltung verarbeitet und im sogenannten Datenwörterbuch gespeichert.

Das für die Mehrbenutzersynchronisation und für die Fehlerbehandlung zuständige Modul (siehe Kapitel 9 bis Kapitel 11) verhindert die Zerstörung von Daten und ist für das Anlegen von Archivkopien und Protokollinformationen zuständig.

1.8 Übungen

1.1 In Abschnitt 1.1 haben wir davon gesprochen, dass unkontrollierte Redundanz unerwünscht ist. Können Sie sich eine sinnvolle Einsatzmöglichkeit für eine durch das DBMS kontrollierte Redundanz vorstellen?

1.2 In einer Universität soll ein DBMS eingesetzt werden. Überlegen Sie sich, welche Daten in einer Universität anfallen, welche Benutzergruppen es gibt und welche Anwendungsprogramme sinnvoll wären. Wie würde die notwendige Funktionalität ohne DBMS realisiert werden? Untersuchen Sie an konkreten Beispielen die in diesem Kapitel beschriebenen Probleme.

1.3 Konzipieren Sie ein Wahlinformationssystem für Bundestagswahlen.

1.9 Literatur

Die meisten einführenden Bücher über Datenbanksysteme sind englischsprachig – eine Ausnahme bildet das sehr umfangreiche Buch von Vossen (2008). Ein weiteres deutschsprachiges Buch ist von Schlageter und Stucky (1983), das aber mittlerweile etwas „in die Jahre gekommen ist". Neumann (1996) und Kleinschmidt und Rank (2002) haben sehr praxisnahe Bücher über die relationale Datenbankanwendung verfasst. Lockemann, Krüger und Krumm (1993) versuchen, in einer Einführungsveranstaltung die grundlegenden Konzepte aus dem Datenbank- und dem Telekommunikationsbereich zu vereinen. Das Buch von Biskup (1995) hat einen etwas stärkeren theoretischen „Touch". Saake, Sattler und Heuer (2013) betonen den Datenmodellierungsaspekt stärker. Lausen (2005) vermittelt die Grundlagen relationaler Datenbanken sowie der XML-Datenbanktechnologie. Das Buch von Silberschatz, Korth und Sudarshan (2010) kommt vom Lehrinhalt diesem Buch am nächsten. Die beiden Bücher von Ullman [Ullman (1988); Ullman (1989)] haben einen stärkeren theoretischen „Einschlag" – insbesondere wird das deduktive Datenmodell sehr ausführlich behandelt. Das sehr pragmatische Buch von Date (2003) ist schon fast ein Klassiker; es ist mittlerweile schon in der sechsten Auflage erschienen. Das Buch von Elmasri und Navathe (2010) ist sehr umfangreich und detailliert. Allen Lesern, die sich intensiver mit der Datenbankforschung auseinandersetzen möchten, sei der hervorragende Bibliographieserver von M. Ley (`http://dblp.uni-trier.de/`) in Trier empfohlen. Weiterhin sei die Mitgliedschaft bei ACM SIGMOD (Association for Computing Machinery, Special Interest Group on Management of Data, `http://www.acm.org`) empfohlen.

2. Datenbankentwurf

Der konzeptuell „saubere" Entwurf sollte die Voraussetzung aller Datenbankanwendungen sein. An dieser Stelle sei eindringlich davor gewarnt, den Datenbankentwurf unvollständig oder nicht mit der notwendigen Systematik durchzuführen. Derartige Versäumnisse rächen sich in späteren Phasen des Datenbankeinsatzes und sind dann oftmals nicht mehr zu korrigieren, weil viele andere Entwurfsentscheidungen (z.B. der Entwurf von Anwendungsprogrammen) davon abhängig sind. Es gibt die Faustregel, dass ein Fehler, der in der Anforderungsanalyse noch mit Kosten von 1 Euro zu korrigieren ist, in der Entwurfsphase schon 10 Euro und in der Realisierungsphase schon 100 Euro Kosten verursacht. Wird der Fehler erst im Einsatz aufgedeckt, sind die Kosten für dessen Behebung nochmals Größenordnungen höher.

2.1 Abstraktionsebenen des Datenbankentwurfs

Beim Entwurf einer Datenbankanwendung kann man drei Abstraktionsebenen unterscheiden:

1. konzeptuelle Ebene,
2. Implementationsebene,
3. physische Ebene.

Die konzeptuelle Ebene dient dazu, den projektierten Anwendungsbereich zu strukturieren. Diese Ebene wird unabhängig von dem zum Einsatz kommenden Datenbanksystem modelliert, und es sollte auch nur die Anwendersicht (im Gegensatz zur Realisierungssicht) modelliert werden. Wir werden später in diesem Kapitel das *Entity-Relationship-Modell* (Gegenstand-Beziehungs-Modell) kennenlernen, das für den konzeptuellen Entwurf eingesetzt wird. In diesem Modell werden Gegenstände zu Gegenstandsmengen und Beziehungen zwischen den Gegenständen zu Beziehungstypen abstrahiert. Weiterhin werden den Gegenstandstypen und den Beziehungstypen Attribute zugeordnet.

Auf der Implementationsebene wird die Datenbankanwendung in den Konzepten (d.h. in dem Datenmodell) des zum Einsatz kommenden Datenbanksystems modelliert. Beim relationalen Datenmodell hat man es hierbei mit Relationen, Tupeln und Attributen zu tun.

Die „niedrigste" Abstraktionsebene behandelt den physischen Entwurf. Hierbei geht es primär darum, die Leistungsfähigkeit (engl. performance) der Datenbankanwendungen zu erhöhen. Die im physischen Entwurf zu betrachtenden Strukturen sind z.B. Datenblöcke (Seiten), Zeiger und Indexstrukturen. Es ist eine tiefgehende Kenntnis des eingesetzten Datenbanksystems, des zugrundeliegenden Betriebssystems und sogar der Hardware erforderlich.

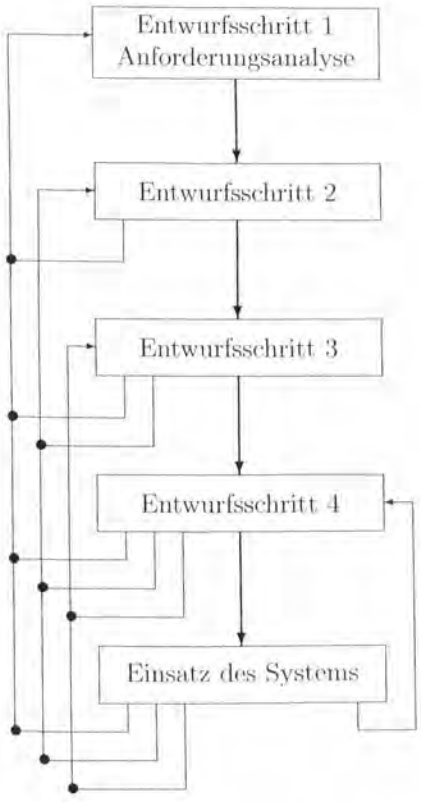


Abbildung 2.1: Allgemeine „top-down"-Entwurfsmethodik

2.2 Allgemeine Entwurfsmethodik

In diesem Abschnitt soll eine *Entwurfsmethodik* (oft auch *Entwurfsmethodologie* genannt) vorgestellt werden, die allgemeingültigen Charakter hat. Diese Methodik ist also geeignet, den Datenbankentwurf unterschiedlichster Anwendungsbereiche systematisch durchzuführen. Um die Komplexität beherrschbar zu machen, wird der Entwurf in mehreren aufeinander aufbauenden Schritten durchgeführt. Es handelt sich hierbei also um eine „top-down"-Vorgehensweise.

Die Systematik ist in Abbildung 2.1 skizziert. Im ersten Schritt, der Anforderungsanalyse, wird das Pflichtenheft (bzw. die Anforderungsspezifikation) erstellt. Hierauf bauen die nachfolgenden Schritte auf. Wichtig ist, dass der gesamte Entwurfsprozess konsistent gehalten wird. Das bedeutet, dass in nachfolgenden Entwurfsschritten vollzogene Änderungen – aufgrund geänderter Randbedingungen oder neu gewonnener Erkenntnisse – in den vorangehenden Schritten (d.h. den dort erzeugten Dokumenten) nachvollzogen werden. Dies ist in der Abbildung durch die nach oben (rück-) gerichteten Pfeile dargestellt.

Der letzte Schritt in diesem Diagramm – Einsatz des Systems – besteht aus der Überwachung (engl. Monitoring) des laufenden Systems, um daraus Rückschlüsse auf eventuell notwendige Änderungen (Adaptionen) ziehen zu können.

2.3 Die Datenbankentwurfsschritte

Die oben vorgestellte abstrakte „top-down"-Entwurfsmethodik wird nun auf den Lebenszyklus eines Datenbankentwurfs zugeschnitten. Der Datenbankentwurf orientiert sich an den in Abschnitt 2.1 beschriebenen Abstraktionsebenen einer Datenbankanwendung. Wie in jeder systematischen Entwurfsmethodik beginnt man auch im Datenbankentwurf mit der *Anforderungsanalyse*. Das dabei erstellte Entwurfsdokument nennt man *Anforderungsspezifikation* oder auch *Pflichtenheft*. In der Anforderungsanalyse müssen zum einen die Informationsanforderungen der zu modellierenden Welt (bzw. des relevanten Ausschnitts der realen Welt – auch Miniwelt genannt) und zum anderen die Datenverarbeitungsvorgänge berücksichtigt werden. Eine sorgfältig ausgeführte Anforderungsanalyse, die in enger Zusammenarbeit mit den projektierten Anwendern des Systems ausgeführt wird, ist die Grundvoraussetzung für die spätere Akzeptanz der Datenbankanwendung.

Anschließend – nach Fertigstellung der Anforderungsspezifikation – erfolgt der *konzeptuelle Entwurf*. In diesem Entwurfsschritt wird die Informationsstruktur auf einer konzeptuellen, d.h. anwenderorientierten Ebene festgelegt. Das am häufigsten für den konzeptuellen Entwurf verwendete Datenmodell ist das Entity-Relationship-Modell.

Als Ausgabe des konzeptuellen Entwurfs erhält man dann die Informationsstrukturbeschreibung in der Form eines Entity-Relationship-Schemas (kurz ER-Schema). Es ist wichtig zu betonen, dass dieser Entwurfsschritt noch gänzlich unabhängig vom eingesetzten Datenbanksystem durchgeführt wird. Das Datenmodell des eingesetzten DBMS kommt erst im *Implementationsentwurf* zum Tragen. Dabei wird das ER-Schema in ein entsprechendes Implementationsschema – oft auch logische Datenbankstruktur genannt – überführt. Beim Implementationsentwurf müssen aber auch die Datenverarbeitungsanforderungen berücksichtigt werden, um ein geeignetes Datenbankschema erstellen zu können.

Der letzte Schritt des Datenbankentwurfs, der *physische Entwurf*, verfolgt das Ziel der Effizienzsteigerung – ohne dabei die logische Struktur der Daten zu verändern. Für den physischen Entwurf ist eine detaillierte Kenntnis des zugrundeliegenden Datenbanksystems, aber auch der Hard- und Software (z.B. des Betriebssystems), auf der das DBMS installiert ist, notwendig.

Die Abfolge und der Zusammenhang dieser Entwurfsschritte ist grafisch in Abbildung 2.2 gezeigt.

2.4 Die Anforderungsanalyse

Das Ziel dieses Abschnitts besteht darin, ein „rezeptartiges" Vorgehen für die Erstellung der *Anforderungsspezifikation* vorzustellen. Die Anforderungsanalyse muss in intensiver Diskussion mit den vorgesehenen Anwendern des Datenbanksystems durchgeführt werden – nur so kann man sich (halbwegs) vor bösen Überraschungen bei der späteren Installation der Datenbankanwendung schützen.

Die Aufgabe der Anforderungsanalyse besteht darin, die durch Gespräche mit den zukünftigen Anwendern gewonnene Information in einem strukturierten Dokument festzuhalten. Ein mögliches Vorgehen kann wie folgt skizziert werden:

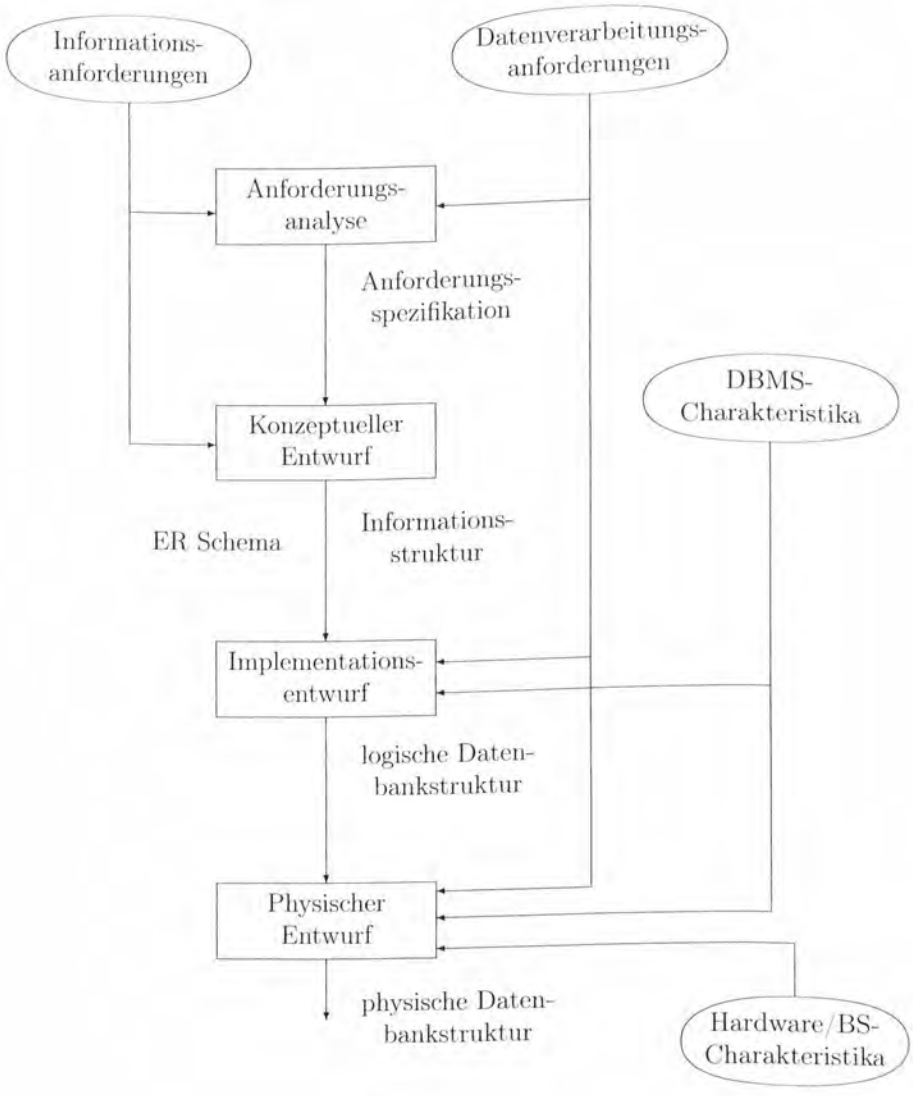


Abbildung 2.2: Die Phasen des Datenbankentwurfs

1. Identifikation von Organisationseinheiten,
2. Identifikation der zu unterstützenden Aufgaben,
3. Anforderungs-Sammelplan: Ermittlung der zu befragenden Personen,
4. Anforderungs-Sammlung,
5. Filterung: Gesammelte Information auf Verständlichkeit und Eindeutigkeit überprüfen,
6. Satzklassifikationen: Information wird Objekten, Beziehungen zwischen Objekten, Operationen und Ereignissen zugeordnet,
7. Formalisierung bzw. Systematisierung: Übertragung auf Verzeichnisse, die in ihrer Gesamtheit das Pflichtenheft repräsentieren.

Die Schritte 1. und 2. dienen dazu, den Anwendungsbereich abzugrenzen. In den Schritten 3. bis 6. werden vom Datenbankdesigner systematisch Informationen über das abgegrenzte Anwendungsgebiet gesammelt. Der Schritt 7. resultiert letztendlich in einem Pflichtenheft, das nach der Beschreibung der Informationsstrukturanforderungen und der Datenverarbeitungsanforderungen gegliedert sein sollte.

2.4.1 Informationsstrukturanforderungen

Für diese Beschreibung hat sich in der Praxis eine formularähnliche Gliederung bewährt, die natürlich – allein schon der Modifizierbarkeit wegen – in maschinenlesbarer Form auf dem Rechner vorliegen sollte. Bestandteile dieser Beschreibung sind

- *Objekte*, die schon zu Objekttypen abstrahiert werden sollten,
- *Attribute*, die diese Objekte beschreiben bzw. identifizieren,
- *Beziehungen* zwischen den Objekten, die auch schon zu Beziehungstypen abstrahiert werden sollten.

Die Objekt- und Attributbeschreibungen können in einem „Formular" zusammengefasst werden, das für das Beispiel *Uni-Angestellte* nachfolgend skizziert ist:

- **Objektbeschreibung:** *Uni-Angestellte*
  - Anzahl: 1000
  - Attribute
    - PersonalNummer
      - Typ: char
      - Länge: 9
      - Wertebereich: 0 . . . 999.999.99
      - Anzahl Wiederholungen: 0

- Definiertheit: 100%
- Identifizierend: ja

* Gehalt
  - Typ: dezimal
  - Länge: (8,2) [1]
  - Anzahl Wiederholungen: 0
  - Definiertheit: 90%
  - Identifizierend: nein
* Rang
  - ...
  - ...
  - ...

Die in der Anforderungsanalyse ermittelten Abschätzungen hinsichtlich Anzahl und Größe der Objekte bzw. der darin enthaltenen Attribute dienen dazu, schon frühzeitig das später anfallende Datenvolumen abzuschätzen. Bei den Attributbeschreibungen sollte schon der Wertebereich festgelegt werden, der Speicherbedarf *(Länge)*, die Anzahl der Wiederholungen (z.B. haben Personen oft 2 Adressen), die Wahrscheinlichkeit, dass das Attribut überhaupt mit einem Wert belegt sein wird *(Definiertheit)* und ob das Attribut das Objekt eindeutig identifiziert.

Die zwischen den Objekten existierenden Beziehungen sollten in einem ähnlich gestalteten Formular dokumentiert werden. Wiederum möge uns die Universitätswelt als Beispiel dienen. Hier gibt es eine (leidige) Beziehung zwischen *Professoren*, *Studenten* und *Vorlesungen* namens *prüfen*:

- **Beziehungsbeschreibung:** *prüfen*
  - Beteiligte Objekte:
    * Professor als Prüfer
    * Student als Prüfling
    * Vorlesung als Prüfungsstoff
  - Attribute der Beziehung
    * Datum
    * Uhrzeit
    * Note
  - Anzahl: 100 000 (pro Jahr)

Es sollte betont werden, dass die „Formulare" für die Objekt- und Beziehungsbeschreibungen nur Muster darstellen, die den jeweiligen Gegebenheiten angepasst werden sollten. Wichtig ist jedoch, dass überhaupt ein gut strukturiertes und konsistentes Dokument erstellt wird.

[1] achtstellige Dezimalzahl mit zwei Nachkommastellen

2.4.2 Datenverarbeitungsanforderungen

Neben der Informationsstruktur muss in der Anforderungsanalyse natürlich auch der operationale Aspekt – also die Datenverarbeitung – behandelt werden. Es empfiehlt sich, diesen Bereich in Einzelprozesse zu zergliedern, für die dann jeweils separate Anforderungsbeschreibungen erstellt werden. Genau wie für die Informationsstrukturbeschreibung, hat sich auch hierfür eine strukturierte Dokumentation in der Praxis bewährt. Wir demonstrieren dies am Beispiel der Zeugnisausstellung:

- **Prozessbeschreibung:** *Zeugnisausstellung*
  - Häufigkeit: halbjährlich
  - benötigte Daten
    - Prüfungen
    - Studienordnungen
    - Studenteninformation
    - ...
  - Priorität: hoch
  - zu verarbeitende Datenmenge
    - 500 Studenten
    - 3000 Prüfungen
    - 10 Studienordnungen

Wenn dies geeignet erscheint, kann man natürlich andere, anwendungsspezifischere „Formulare" entwerfen. Die Formulare müssen auf jeden Fall so gestaltet sein, dass man sie als Diskussionsgrundlage mit den zukünftigen Anwendern verwenden kann.

2.5 Grundlagen des Entity-Relationship-Modells

Wie der Name schon sagt, sind die grundlegendsten Modellierungsstrukturen dieses Modells die *Entities* (Gegenstände) und die *Relationships* (Beziehungen) zwischen den Entities. Zusätzlich „kennt" das Entity-Relationship-Modell (kurz ER-Modell genannt) noch *Attribute* und *Rollen*.

Gegenstände (bzw. Entities) sind wohlunterscheidbare physisch oder gedanklich existierende Konzepte der zu modellierenden Welt. Man abstrahiert ähnliche Gegenstände zu Gegenstandstypen (Entitytypen oder Entitymengen), die man grafisch als Rechtecke darstellt, wobei der Name des Entitytyps innerhalb des Rechtecks angegeben wird.

Beziehungen werden auf analoge Weise zu Beziehungstypen zwischen den Gegenstandstypen abstrahiert. Die Beziehungstypen werden als Rauten mit entsprechender Beschriftung repräsentiert. Die Rauten werden mit den beteiligten Gegenstandstypen über ungerichtete Kanten verbunden.

Im Folgenden werden wir oft die Unterscheidung zwischen Gegenständen und den Gegenstandstypen, bzw. zwischen Beziehungen und Beziehungstypen, vernachlässigen. Aus dem Kontext dürfte immer leicht ersichtlich sein, was gemeint ist.

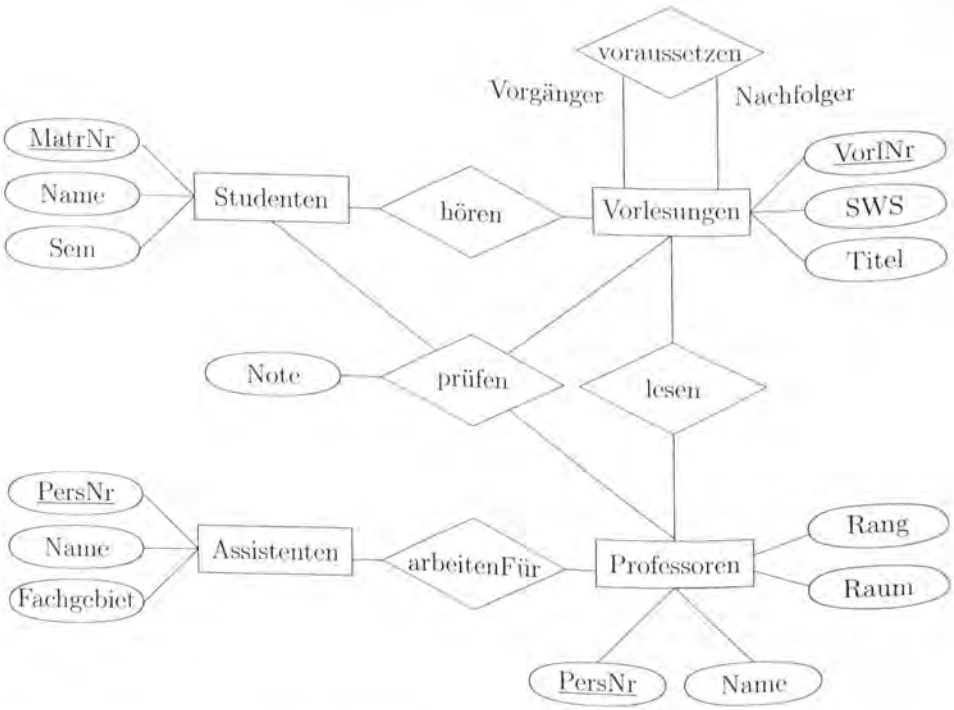


Abbildung 2.3: Ein konzeptuelles Universitätsschema

Beispiele für Gegenstandstypen in der Universitätswelt sind *Studenten*, *Vorlesungen*, *Professoren* und *Assistenten*. Beziehungen zwischen diesen Entitytypen sind z.B. *hören* (zwischen *Studenten* und *Vorlesungen*) und *prüfen* (zwischen *Professoren*, *Vorlesungen* und *Studenten*).

Attribute dienen dazu, Gegenstände bzw. Beziehungen zu charakterisieren. Folglich werden Attribute, die durch Kreise oder Ovale grafisch beschrieben werden, den Rechtecken (für Gegenstandstypen) bzw. den Rauten (für Beziehungstypen) durch verbindende Kanten zugeordnet.

In dem in Abbildung 2.3 gezeigten ER-Schema gibt es vier Gegenstandstypen (*Studenten*, *Vorlesungen*, *Professoren*, *Assistenten*) und fünf Beziehungstypen (*hören*, *prüfen*, *voraussetzen*, *arbeitenFür*, *lesen*). Den Gegenstandstypen sind jeweils ein identifizierendes Attribut als Schlüssel und noch weitere beschreibende Attribute zugeordnet. Zum Beispiel werden *Studenten* durch die *MatrNr* (Matrikelnummer) eindeutig identifiziert; wohingegen der *Name* bzw. das *Semester* als weitere Attribute angegeben sind, die aber i.A. einen Studenten nicht eindeutig identifizieren, sondern nur zur (detaillierteren) Beschreibung dienen.

Die Beziehungen *hören*, *lesen* und *arbeitenFür* sind *binäre* Beziehungen zwischen zwei unterschiedlichen Entitytypen. Auch die Beziehung *voraussetzen* ist binär, aber es ist nur ein Entitytyp beteiligt. In diesem Fall spricht man von einer *rekursiven* Beziehung. Weiterhin wurden in der Beschreibung der Beziehung *voraussetzen* Rollen zugeordnet, nämlich *Vorgänger* und *Nachfolger*. Dadurch wird die Rolle eines

Gegenstandes in dieser Beziehung dokumentiert, d.h. in diesem Fall legen die Rollen fest, ob die betreffende Vorlesung als Nachfolger auf der anderen Vorlesung aufbaut oder umgekehrt. Rollen werden als Text an die jeweiligen „Ausgänge" (Kanten) der Beziehungsraute geschrieben.

2.6 Schlüssel

Eine minimale Menge von Attributen, deren Werte das zugeordnete Entity eindeutig innerhalb aller Entities seines Typs identifiziert, nennt man *Schlüssel*. Sehr oft gibt es einzelne Attribute, die als Schlüssel „künstlich" eingebaut werden, wie z.B. Personalnummer (*PersNr*), Vorlesungsnummer (*VorlNr*), etc. Schlüsselattribute werden durch Unterstreichung (manchmal auch durch doppelt gezeichnete Kreise bzw. Ovale) gekennzeichnet.

Manchmal gibt es auch zwei unterschiedliche Schlüsselkandidaten: Dann wählt man einen dieser Kandidaten-Schlüssel als Primärschlüssel aus.

2.7 Charakterisierung von Beziehungstypen

Ein Beziehungstyp R zwischen den Entitytypen $E_1, E_2, \ldots, E_n$ kann als Relation im mathematischen Sinn angesehen werden. Demnach stellt die Ausprägung der Beziehung R eine Teilmenge des kartesischen Produkts der an der Beziehung beteiligten Entitytypen dar. Also gilt:

$$R \subseteq E_1 \times E_2 \times \cdots \times E_n$$

In diesem Fall bezeichnet man n als den Grad der Beziehung R – die in der Praxis mit Abstand am häufigsten vorkommenden Beziehungstypen sind *binär*.

Ein Element $(e_1, e_2, \ldots, e_n) \in R$ nennt man eine Instanz des Beziehungstyps, wobei $e_i \in E_i$ für alle $1 \leq i \leq n$ gelten muss. Eine solche Instanz ist also ein Tupel aus dem kartesischen Produkt $E_1 \times E_2 \times \cdots \times E_n$.

Man kann jetzt auch den Begriff der Rolle etwas formaler fassen. Dazu veranschaulichen wir uns nochmals die Beziehung *voraussetzen* aus unserem Beispielschema (siehe Abbildung 2.3). Gemäß dem oben skizzierten Formalismus gilt:

$$\text{voraussetzen} \subseteq \text{Vorlesungen} \times \text{Vorlesungen}$$

Um einzelne Instanzen $(v_1, v_2) \in$ *voraussetzen* genauer zu charakterisieren, wird die jeweilige Rolle, nämlich ($Vorgänger : v_1$, $Nachfolger : v_2$), benötigt. Dadurch wird also unmissverständlich festgelegt, dass die Vorlesung v_1 die Voraussetzung für die Vorlesung v_2 darstellt.

2.7.1 Funktionalitäten der Beziehungen

Man kann Beziehungstypen hinsichtlich ihrer *Funktionalität* charakterisieren. Ein binärer Beziehungstyp R zwischen den Entitytypen E_1 und E_2 heißt

- 1:1-*Beziehung*, 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 maximal ein Entity e_1 aus E_1 zugeordnet ist. Man beachte, dass es auch Entities aus E_1 (bzw. E_2) geben kann, denen kein „Partner" aus E_2 (bzw. E_1) zugeordnet ist.

  Ein Beispiel einer „realen" 1:1-Beziehung ist *verheiratet* zwischen den Entitytypen *Männer* und *Frauen* – zumindest nach europäischem Recht.

- 1:N-*Beziehung*, falls jedem Entity e_1 aus E_1 beliebig viele (also mehrere oder auch gar keine) Entities aus E_2 zugeordnet sein können, aber jedes Entity e_2 aus der Menge E_2 mit maximal einem Entity aus E_1 in Beziehung stehen kann.

  Ein anschauliches Beispiel für eine 1:N-Beziehung ist *beschäftigen* zwischen *Firmen* und *Personen*, wenn wir davon ausgehen, dass eine Firma i.A. mehrere Personen beschäftigt, aber eine Person nur bei einer (oder gar keiner) Firma angestellt ist.

- N:1-*Beziehung*, falls analoges zu obigem gilt.

- N:M-*Beziehung*, wenn keinerlei Restriktionen gelten müssen, d.h. jedes Entity aus E_1 mit beliebig vielen Entities aus E_2 in Beziehung stehen kann und umgekehrt jedes Entity aus E_2 mit beliebig vielen Entities aus E_1 assoziiert werden darf.

Man beachte, dass die Funktionalitäten Integritätsbedingungen darstellen, die in der zu modellierenden Welt immer gelten müssen. D.h. diese Bedingungen sollen nicht nur im derzeit existierenden Zustand der Miniwelt (rein zufällig) gelten, sondern sie sollen Gesetzmäßigkeiten darstellen, deren Einhaltung erzwungen wird.

Die binären 1:1-, 1:N- und N:1-Beziehungen kann man auch als *partielle Funktionen* ansehen. Bei einer 1:1-Beziehung R zwischen E_1 und E_2 kann die Funktion sowohl als $R : E_1 \to E_2$ wie auch als $R^{-1} : E_2 \to E_1$ gesehen werden.

Bezogen auf unser Beispiel einer 1 : 1-Beziehung haben wir also:

$$\begin{aligned} \text{Ehemann} &: \text{Frauen} \to \text{Männer} \\ \text{Ehefrau} &: \text{Männer} \to \text{Frauen} \end{aligned}$$

Bei einer 1:N-Beziehung ist die „Richtung" der Funktion zwingend. Die Beziehung *beschäftigen* ist z.B. eine partielle Funktion von *Personen* nach *Firmen*, also:

$$\text{beschäftigen} : \text{Personen} \to \text{Firmen}$$

Die Funktion geht also von dem „N"-Entitytyp zum „1"-Entitytyp. Wir werden später im Zusammenhang mit der Umsetzung von ER-Schemata in relationale Schemata nochmals auf diesen wichtigen Punkt zurückkommen. Analoges gilt natürlich wieder für N:1-Beziehungen, wobei wiederum die „Richtung" der Funktion zu beachten ist.

Die den 1:1- bzw. 1:N-Beziehungen zugeordneten Funktionen sind partiell, weil es Entities aus dem Definitionsbereich geben kann, die gar keine Beziehung eingehen. Für diese Entities ist die Funktion somit nicht definiert.

In Abbildung 2.4 sind die oben verbal beschriebenen Funktionalitäten grafisch veranschaulicht. Die Ovale repräsentieren die Entitytypen: das linke den Entitytyp

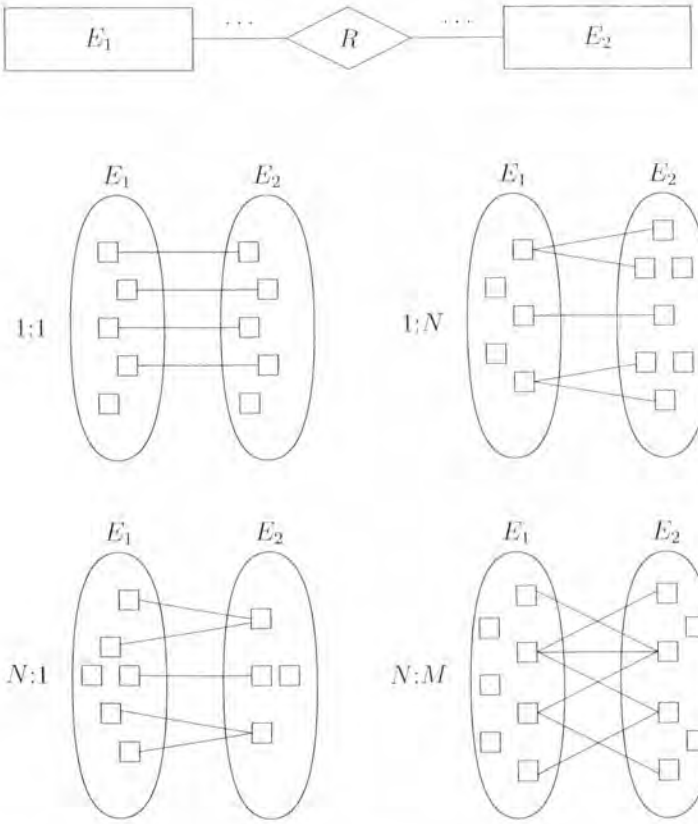


Abbildung 2.4: Grafische Veranschaulichung der Funktionalitäten einer binären Beziehung R zwischen E_1 und E_2.

E_1 und das rechte den Entitytyp E_2. Die kleinen Quadrate innerhalb der Ovale stellen die Entities des jeweiligen Typs dar und die die Entities verbindenden Linien repräsentieren jeweils eine Instanz der Beziehung R.

In Abbildung 2.7 (auf Seite 46) sind die Funktionalitäten des konzeptuellen Universitätsschemas eingezeichnet.

2.7.2 Funktionalitätsangaben bei n-stelligen Beziehungen

Die Angabe von Funktionalitäten wurde bisher nur für binäre Beziehungstypen definiert, sie kann aber auf n-stellige Beziehungen erweitert werden. Sei also R eine Beziehung zwischen den Entitymengen $E_1, \ldots, E_n$, wobei die Funktionalität bei der Entitymenge E_k $(1 \leq k \leq n)$ mit einer „1" spezifiziert sein soll, bei den anderen Mengen ebenfalls mit „1" oder mit einem in dem Beziehungstyp eindeutigen Buchstaben, der wie vorher „viele" repräsentiert. Dann muss gelten, dass durch R die

folgende partielle Funktion vorgegeben wird:

$$R : E_1 \times \ldots \times E_{k-1} \times E_{k+1} \times \ldots \times E_n \rightarrow E_k$$

Solche funktionalen Beziehungen müssen dann natürlich für alle Entitymengen von R gelten, die bei der Funktionalitätsangabe ebenfalls mit einer „1" gekennzeichnet sind. Die Leser mögen sich hier vergegenwärtigen, dass die in Abschnitt 2.7.1 vorgestellten Funktionalitäten für binäre Beziehungen Spezialfälle der obigen Definition sind.

Als anschauliches Beispiel ist in Abbildung 2.5 die dreistellige Beziehung *betreuen* zwischen den Entitytypen *Studenten*, *Professoren* und *Seminarthemen* mit Kardinalitätsangabe N:1:1 grafisch dargestellt. Gemäß obiger Definition kann man die Beziehung *betreuen* demnach als partielle Funktionen wie folgt auffassen:

$$\begin{aligned} \text{betreuen} &: \text{Professoren} \times \text{Studenten} \rightarrow \text{Seminarthemen} \\ \text{betreuen} &: \text{Seminarthemen} \times \text{Studenten} \rightarrow \text{Professoren} \end{aligned}$$

Die Kardinalitätsangaben dieses Beispielschemas legen folgende Konsistenzbedingungen fest, die im Wesentlichen die Studienordnung unserer Universität darstellen:

1. Studenten dürfen bei demselben Professor bzw. derselben Professorin nur ein Seminarthema „ableisten" (damit ein breites Spektrum abgedeckt wird).

2. Studenten dürfen dasselbe Seminarthema nur einmal bearbeiten – sie dürfen also nicht bei anderen Professoren ein schon einmal erteiltes Seminarthema nochmals bearbeiten.

Es sind aber folgende Datenbankzustände nach wie vor möglich:

- Professoren können dasselbe Seminarthema „wiederverwenden" – also dasselbe Thema auch mehreren Studenten erteilen.

- Ein Thema kann von mehreren Professoren vergeben werden – aber an unterschiedliche Studenten.

In Abbildung 2.6 sind vier legale Ausprägungen der Beziehung *betreuen* – mit b_1, b_2, b_3, b_4 markiert – und zwei illegale Ausprägungen b_5 und b_6 dargestellt. Die Ausprägung b_5 ist illegal, weil Student/in s_1 bei Professor/in p_1 zwei Seminare belegen will. Die Beziehungsausprägung b_6 ist illegal, weil Student/in s_3 – gemäß dem Prinzip des geringsten Aufwands – versucht, dasselbe Thema t_1 zweimal bei unterschiedlichen Professoren p_3 und p_4 zu bearbeiten.

Wir wollen die Angabe von Funktionalitäten jetzt noch für die ternäre Beziehung *prüfen* unseres konzeptuellen Universitätsschemas aus Abbildung 2.7 diskutieren. Studenten sollten daran gehindert werden, dieselben Vorlesungen von unterschiedlichen Professoren prüfen zu lassen. Mit anderen Worten, zu einem Paar aus *Studenten* und *Vorlesungen* soll es maximal einen Professor bzw. eine Professorin als Prüfer geben. Die Beziehung *prüfen* sollte also den Eigenschaften einer partiellen Funktion der folgenden Form genügen:

$$\text{prüfen} : \text{Studenten} \times \text{Vorlesungen} \rightarrow \text{Professoren}$$

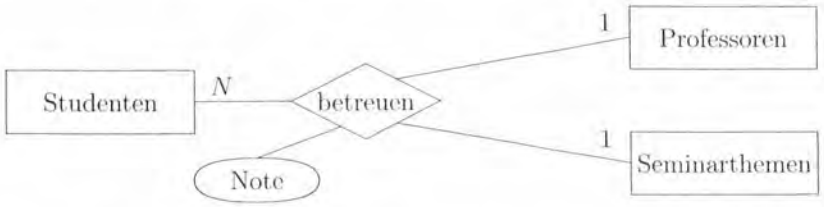


Abbildung 2.5: Die Betreuung von Seminarthemen als Entity-Relationship-Diagramm

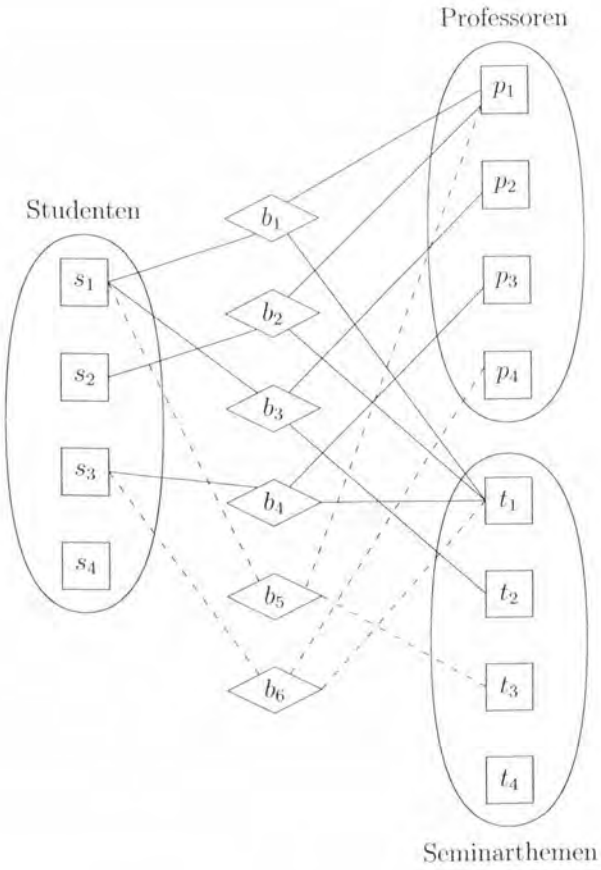


Abbildung 2.6: Ausprägungen der Beziehung *betreuen*: gestrichelte Linien markieren illegale Ausprägungen

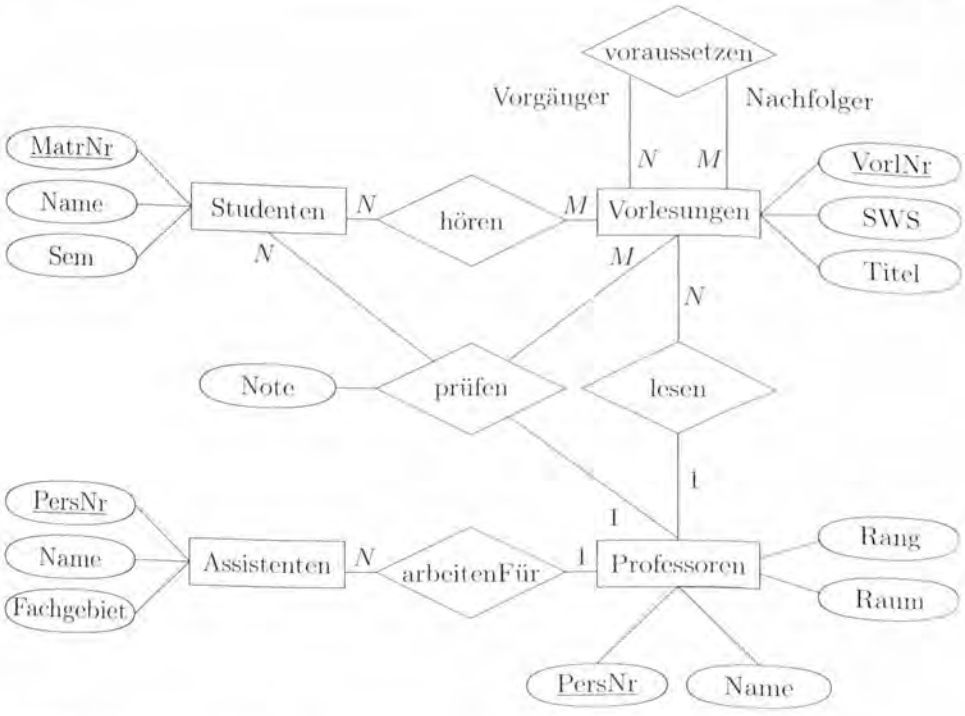


Abbildung 2.7: Markierung der Funktionalitäten im Universitätsschema

Dies wird durch die N:M:1-Funktionalitätsangabe im ER-Schema erzwungen. Weitere Einschränkungen dieser Art gibt es nicht, da wir annehmen, dass Studenten mehrere Vorlesungen von demselben Professor bzw. derselben Professorin abprüfen lassen können und Professoren selbstverständlich mehrere Studenten über dieselbe Vorlesung prüfen.

2.7.3 Die (min, max)-Notation

Im vorangegangenen Abschnitt haben wir die Funktionalitäten von Beziehungstypen beschrieben. Hier wollen wir einen Formalismus einführen, mit dem Beziehungen oft präziser charakterisiert werden können. Bei der Angabe der Funktionalität ist für ein Entity nur die maximale Anzahl von Beziehungsinstanzen relevant. Wann immer diese Zahl größer als eins ist, wird sie, ohne genauere Aussagen zu machen, als N oder M (also „viele“) angegeben. Bei der (min, max)-Notation werden nicht nur die Extremwerte – nämlich *eins* oder *viele* – angegeben sondern, wann immer möglich, präzise Unter- und Obergrenzen festgelegt. Es wird also auch die minimale Anzahl angegeben, weil dies für viele Beziehungstypen eine sinnvolle und manchmal auch essentielle Integritätsbedingung darstellt.

Bei der (min, max)-Notation wird für jeden an einem Beziehungstyp beteiligten Entitytyp 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

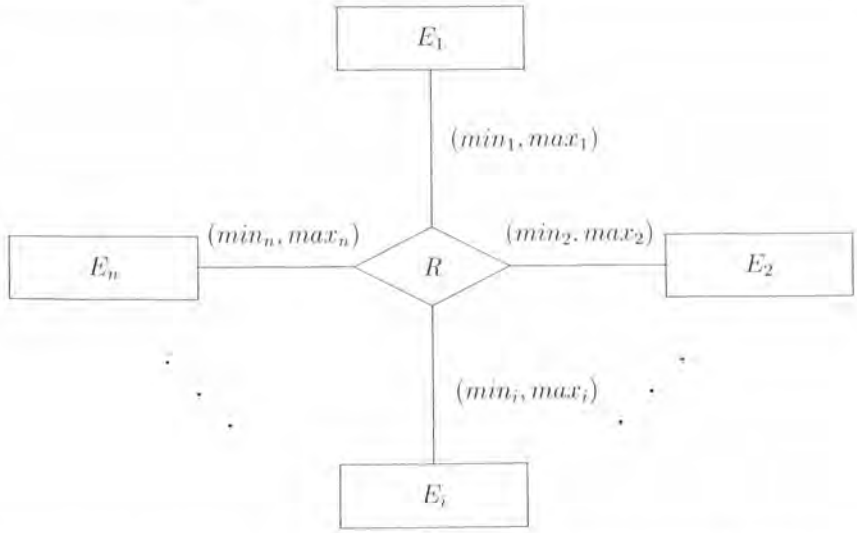


Abbildung 2.8: Abstrakte n-stellige Beziehung R mit (min, max)-Angabe

und höchstens max-mal.

Um das etwas formaler auszudrücken, schauen wir uns die Grafik in Abbildung 2.8 an. In dieser Abbildung ist eine abstrakte n-stellige Beziehung R gezeigt. Somit stellt R eine Relation zwischen den Entitymengen $E_1, E_2, \ldots, E_n$ dar, wobei also gilt:

$$R \subseteq E_1 \times \cdots \times E_i \times \cdots \times E_n$$

Die Markierung (min_i, max_i) gibt an, dass es für alle $e_i \in E_i$ mindestens min_i Tupel der Art $(\ldots, e_i, \ldots)$ und höchstens max_i viele solcher Tupel in R gibt. Wiederum ist gefordert, dass diese Angaben Gesetzmäßigkeiten der modellierten realen Welt darstellen und nicht einfach nur den derzeitigen zufälligen Zustand der Datenbasis beschreiben.

Sonderfälle werden wie folgt gehandhabt:

- 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.

Die Angabe $(0, *)$ ist somit die allgemeinste, da sie aussagt, dass betreffende Entities gar nicht oder beliebig oft in der Beziehung vorkommen können.

Wir wollen uns nun noch ein illustratives Beispiel für die (min, max)-Notation anschauen. Dazu verlassen wir (kurzzeitig) unsere Universitätswelt und schauen uns die Begrenzungsflächenmodellierung von Polyedern an. Ein Polyeder wird dabei durch die Hülle seiner begrenzenden Flächen beschrieben, diese wiederum werden durch ihre Begrenzung, bestehend aus Kanten, modelliert. Eine Kante hat einen Start und ein Ende in der Form eines Punktes im dreidimensionalen Raum. Diese Modellierung

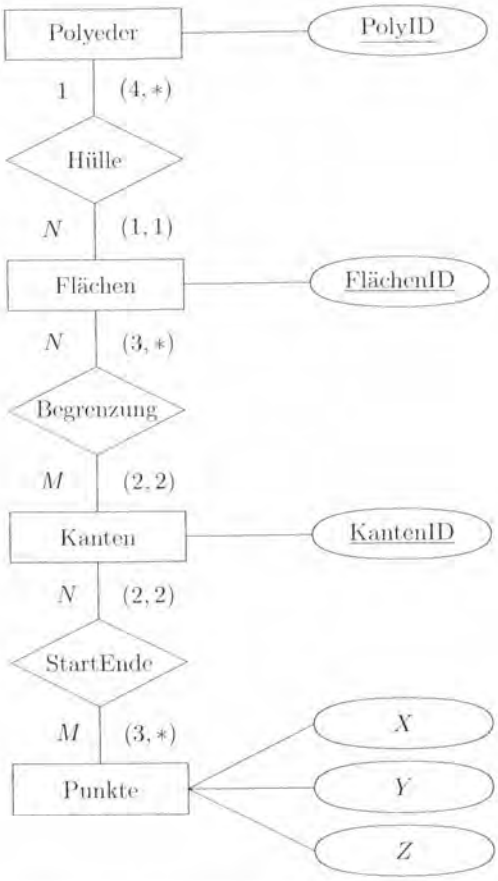


Abbildung 2.9: Konzeptuelles Schema der Begrenzungsflächendarstellung von Polyedern

von Polyedern ist in Abbildung 2.9 als ER-Schema mit der vergleichenden Angabe von Funktionalitäten und der (*min*, *max*)-Notation gezeigt.

Schauen wir uns zuerst die Funktionalitäten an: Ein Polyeder wird von mehreren Flächen umhüllt, wobei jede Fläche nur zu einem Polyeder gehören soll. Also ist *Hülle* eine 1:N-Beziehung. Eine Fläche wird von mehreren Kanten begrenzt und jede Kante gehört zu mehreren Flächen. Folglich ist *Begrenzung* eine allgemeine N:M-Beziehung. Auch *StartEnde* ist eine N:M-Beziehung, da jeder Kante zwei (d.h. mehrere) Punkte zugeordnet sind und ein Punkt mehreren (sogar beliebig vielen) Kanten eines Polyeders zugeordnet sein kann.

Diese sehr grobe Charakterisierung der Beziehungen kann durch Verwendung der (*min*, *max*)-Notation viel präziser erfolgen. Für die Angabe der in Abbildung 2.9 gezeigten *min*-Werte sollten wir uns den Polyeder mit der minimalen Flächen-, Kanten- und Punkteanzahl vergegenwärtigen: Eine grafische Skizze des Tetraeders ist in Abbildung 2.10 gezeigt.

In dieser Abbildung ist erkennbar, dass ein Polyeder minimal vier umhüllende

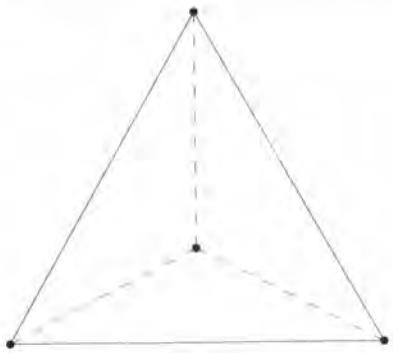


Abbildung 2.10: Grafische Darstellung eines Tetraeders

Flächen besitzt – die maximale Anzahl ist beliebig und wird somit durch $*$ angegeben. Eine Fläche wird – im Falle eines Dreiecks – durch ein Minimum von drei Kanten begrenzt; wiederum ist die maximale Anzahl von begrenzenden Kanten beliebig. Jede Kante begrenzt bei einem Polyeder genau zwei Flächen. Eine Kante wird durch genau zwei Punkte beschrieben, die über die Beziehung *StartEnde* der Kante zugeordnet werden. Bei einem Polyeder gehört jeder Begrenzungspunkt – und nur solche sind in der Entitymenge *Punkte* enthalten – zu mindestens drei Kanten (siehe Tetraeder) und maximal zu beliebig vielen.

Wir sollten noch darauf hinweisen, dass der Vergleich der 1:N-Angabe mit der (min, max)-Notation in gewisser Weise „kontra-intuitiv" ist. Dazu betrachte man die 1:N-Beziehung *Hülle* aus Abbildung 2.9. Hierbei ist der Entitytyp *Flächen* mit N markiert; andererseits „wandert" bei der (min, max)-Notation die Angabe „viele", also das Sternchen $*$, zu dem Entitytyp *Polyeder*. Dies liegt in der Definition der (min, max)-Notation begründet – siehe dazu auch die Übungsaufgabe 2.1.

In Abbildung 2.14 (auf Seite 53) sind die (min, max)-Angaben für unser Universitätsschema angegeben. Alle Vorlesungen werden von mindestens drei Studenten gehört – andernfalls finden sie nicht statt. Assistenten sind genau einem Professor bzw. einer Professorin als Mitarbeiter/in zugeordnet. Weiterhin werden Vorlesungen von genau einem Professor bzw. einer Professorin gelesen. Professoren können zeitweilig – aufgrund von Forschungssemestern oder anderer Verpflichtungen in der Universitätsleitung – vom Vorlesungsbetrieb freigestellt werden; daraus resultiert die Markierung $(0, *)$.

Man beachte, dass die Ausdruckskraft der Funktionalitätsangaben und der (min, max)-Angaben bei n-stelligen Beziehungen mit $n > 2$ unvergleichbar ist: Es gibt Konsistenzbedingungen, die mit Funktionalitätsangaben, aber nicht mit (min,max)-Angaben ausdrückbar sind und wiederum andere Konsistenzbedingungen, die mit der (min,max)-Angabe formulierbar sind, aber nicht durch Funktionalitätseinschränkungen. Siehe dazu auch Übungsaufgabe 2.2.

Grundsätzlich gilt natürlich, dass es viele anwendungsspezifische Konsistenzbedingungen gibt, die mit den Mitteln des Entity-Relationship-Modells nicht formulierbar sind. Diese müssen im Zuge der Anforderungsanalyse und des konzeptuellen Entwurfs anderweitig (z.B. in natürlicher Sprache) festgehalten werden, damit sie

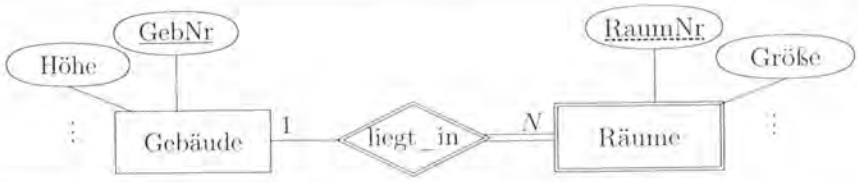


Abbildung 2.11: Ein existenzabhängiger (schwacher) Entitytyp

beim Implementationsentwurf und bei der Realisierung der Anwendungsprogramme berücksichtigt werden.

2.8 Existenzabhängige Entitytypen

Bislang waren wir immer davon ausgegangen, dass Entities autonom existieren und innerhalb ihrer Entitymenge über die Schlüsselattribute eindeutig identifizierbar sind. In der Realität gibt es aber oft auch sogenannte *schwache* Entities, bei denen dies nicht gilt. Diese Entities sind also

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

Als Beispiel betrachten wir den Entitytyp *Räume* aus Abbildung 2.11. Die Entities dieses Typs sind existenzabhängig von dem *Gebäude*, in dem der betreffende Raum liegt. Dies ist natürlich intuitiv auch einsichtig: Wenn man das Gebäude abreißt, verschwinden damit automatisch auch alle in dem betreffenden Gebäude liegenden Räume.

Schwache Entities werden durch doppelt umrandete Rechtecke repräsentiert. Die Beziehung zu dem übergeordneten Entitytyp wird ebenfalls durch eine Verdopplung der Raute und der von dieser Raute zum schwachen Entitytyp ausgehenden Kante markiert. Die Beziehung zum übergeordneten Entitytyp hat i.A. eine 1:N-Funktionalität oder, in selteneren Fällen, eine 1:1-Funktionalität. Die Leser mögen sich überlegen, warum diese Beziehung keine N:M-Funktionalität haben kann (siehe Übungsaufgabe 2.9).

Existenzabhängige Entitytypen haben, wie oben schon angemerkt, i.A. keinen eigenständigen Schlüssel, der alle Entities der Entitymenge eindeutig identifiziert. Stattdessen gibt es ein Attribut (oder manchmal auch eine Menge von Attributen), dessen Wert alle schwachen Entities, die *einem* übergeordneten Entity zugeordnet sind, voneinander unterscheidet. Diese Attribute werden in der grafischen Notation (siehe Abbildung 2.11) gestrichelt unterstrichen. In unserem Beispiel handelt es sich hierbei um das Attribut *RaumNr*: Alle *Räume* in demselben *Gebäude* haben eine eindeutige *RaumNr*; aber verschiedene *Gebäude* können durchaus *Räume* mit derselben *RaumNr* haben. Demnach werden *Räume* global eindeutig durch die *GebNr* – also dem Schlüsselwert des übergeordneten Entities – *und* der *RaumNr* identifiziert.

Wir wollen noch ein weiteres Beispiel eines schwachen Entitytyps diskutieren. Anstatt einer Beziehung *prüfen* – wie in unserem Universitätsschema aus Abbil-

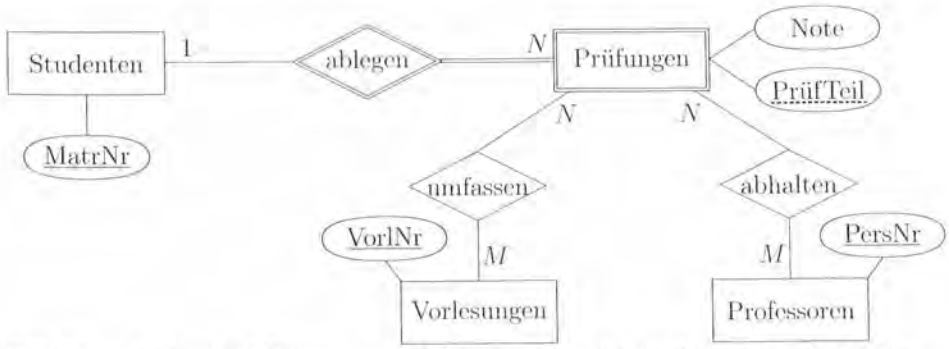


Abbildung 2.12: Modellierung von Prüfungen als existenzabhängigem Entitytyp

dung 2.7 – könnte man auch einen Entitytyp *Prüfungen* einführen. Diesen Entitytyp könnte man, wie in Abbildung 2.12 gezeigt, dem Entitytyp *Studenten* als schwachen Entitytyp unterordnen, so dass *Prüfungen* durch die *MatrNr* des Prüflings und dem *PrüfTeil*, also einer Kennung des abgelegten Prüfungsteils (wie z.B. Informatik I, Informatik II, etc.) global eindeutig identifiziert werden. Wir haben in diesem Beispiel bewusst eine N:M-Funktionalitätsangabe sowohl zwischen *Prüfungen* und *Vorlesungen* als auch zwischen *Prüfungen* und *Professoren* festgelegt. Daher kann eine Prüfung mehrere Vorlesungen umfassen und an einer Prüfung können auch mehrere Professoren als Prüfer beteiligt sein. Man beachte, dass diese Modellierung von der in Abbildung 2.7 durchgeführten Modellierung als dreistellige 1:N:M-Beziehung *prüfen* zwischen *Studenten*, *Professoren* und *Vorlesungen* abweicht. Es sei den Lesern überlassen, zu diskutieren, welche Modellierung ihre jeweilige Prüfungsordnung am besten widerspiegelt.

2.9 Generalisierung

Die *Generalisierung* wird im konzeptuellen Entwurf eingesetzt, um eine bessere (d.h. natürlichere und übersichtlichere) Strukturierung der Entitytypen zu erzielen. Insofern handelt es sich bei der Generalisierung um eine *Abstraktion* auf Typebene. Die analoge Abstraktion auf Instanzebene bestand ja gerade darin, ähnliche Entities in der Form eines gemeinsamen Entitytyps zu modellieren.

Bei der Generalisierung werden die Eigenschaften ähnlicher Entitytypen – im ER-Entwurf sind dies im Wesentlichen die Attribute und Beziehungen, da man Operationen vernachlässigt – „herausfaktorisiert" und einem gemeinsamen *Obertyp* zugeordnet. Die ähnlichen Entitytypen, aus denen diese Eigenschaften faktorisiert werden, heißen *Untertypen* (oder Kategorien) des generalisierten Obertyps.

Diejenigen Eigenschaften, die nicht „faktorisierbar" sind, weil sie nicht allen Untertypen gemein sind, verbleiben beim jeweiligen Untertyp. In dieser Hinsicht stellt der Untertyp eine *Spezialisierung* des Obertyps dar. Ein Schlüsselkonzept der Generalisierung ist die *Vererbung*: Ein Untertyp erbt sämtliche Eigenschaften des Obertyps.

Ein Obertyp ist also eine Generalisierung der Untertypen, wobei es sich dabei um eine Betrachtung auf der Typebene handelt. Wie sieht die Generalisierung auf

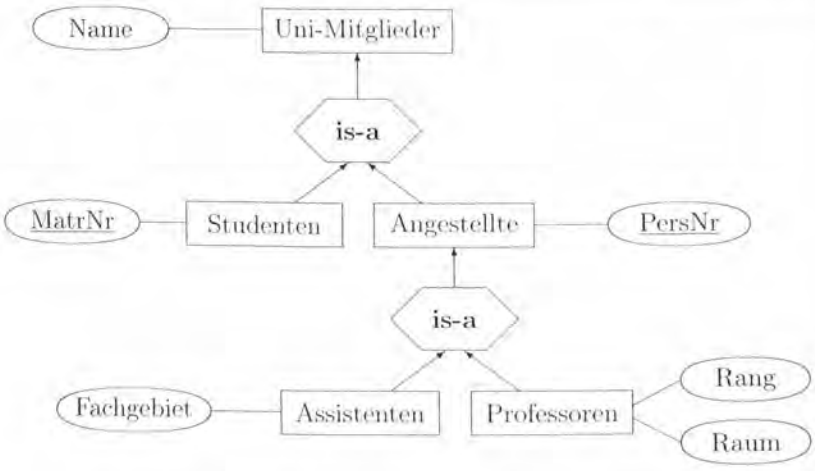


Abbildung 2.13: Generalisierung der Universitätsmitglieder

Instanzebene aus? Die Entities eines Untertyps werden implizit auch als Entities (Elemente) des Obertyps betrachtet. Dies motiviert die Bezeichnung **is-a** in der grafischen Darstellung von Generalisierungen (vgl. Abbildung 2.13). Es handelt sich also um eine besondere Art der Beziehung. Deshalb verwendet man oft auch ein anderes grafisches Symbol, nämlich ein Sechseck anstatt einer Raute.

Somit ist die Entitymenge des Untertyps eine Teilmenge der Entitymenge des Obertyps. Hinsichtlich der Teilmengensicht sind bei der Generalisierung bzw. Spezialisierung zwei Fälle von besonderem Interesse:

- *disjunkte* Spezialisierung: Dies ist der Fall, wenn die Entitymengen aller Untertypen eines Obertyps paarweise disjunkt sind.
- *vollständige* Spezialisierung: Die Spezialisierung ist vollständig, wenn die Entitymenge des Obertyps keine *direkten* Elemente enthält – sich also nur aus der Vereinigung der Entitymengen der Untertypen ergibt.

In unserem Beispiel aus Abbildung 2.13 ist die Spezialisierung von *Uni-Mitglieder* in *Studenten* und *Angestellte* vollständig und disjunkt – sieht man von der Möglichkeit ab, dass Angestellte gleichzeitig studieren könnten. Die Spezialisierung von *Angestellte* in *Assistenten* und *Professoren* ist sicherlich disjunkt, aber nicht vollständig, da es noch andere, nichtwissenschaftliche Angestellte (z.B. Sekretärinnen) gibt, die direkt in der Entitymenge *Angestellte* enthalten wären. In Abbildung 2.14 ist unser Universitätsbeispiel vervollständigt: Es enthält jetzt die Generalisierung von *Assistenten* und *Professoren* zu *Angestellte*. Außerdem sind in diesem ER-Diagramm die Beziehungen durch die (*min*, *max*)-Angabe charakterisiert.

2.10 Aggregation

Während man bei der Generalisierung gleichartige Entitytypen strukturiert, werden bei der Aggregation unterschiedliche Entitytypen, die in ihrer Gesamtheit einen

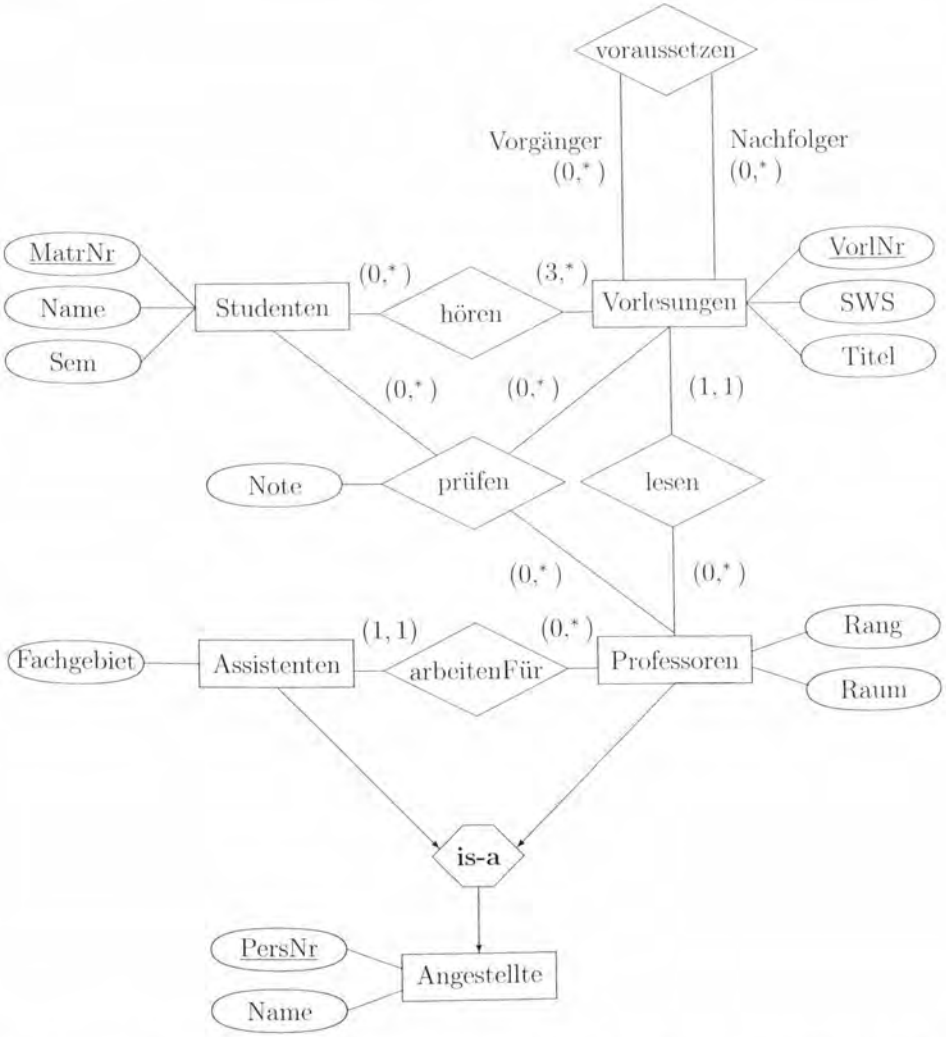


Abbildung 2.14: Das Beispielschema der Universität mit (*min*, *max*)-Angabe und einer Generalisierung

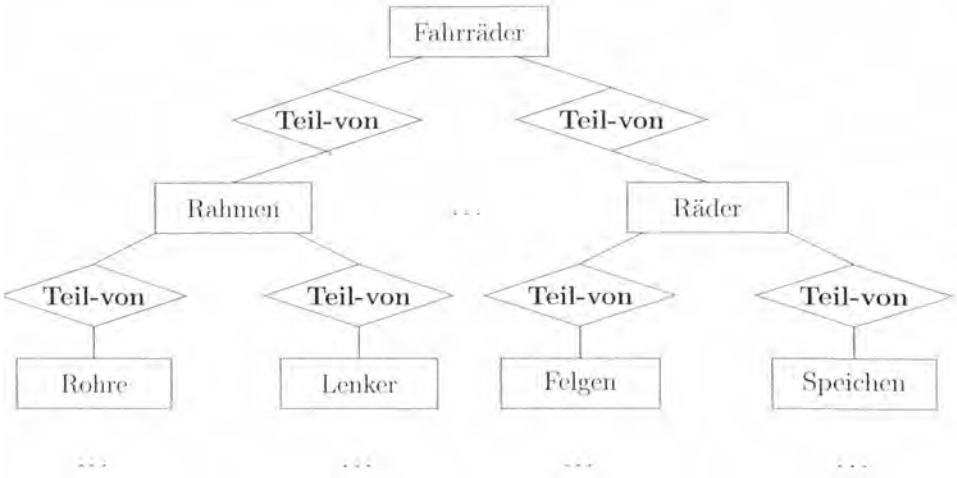


Abbildung 2.15: Aufbau eines Fahrrads

strukturierten Objekttypen bilden, einander zugeordnet. In dieser Hinsicht kann man die Aggregation als einen besonderen Beziehungstyp deuten, der einem übergeordneten Entitytyp[2] mehrere untergeordnete Entitytypen zuordnet. Diese Beziehung wird als **Teil-von** (engl. **part-of**) bezeichnet, um zu betonen, dass die untergeordneten Entities Teile (also Komponenten) der übergeordneten (zusammengesetzten) Entities sind.

Um ein anschauliches Beispiel für eine Aggregationshierarchie zu geben, verlassen wir kurz den Universitätsbereich und schauen uns den Aufbau eines Fahrrads an. Fahrräder bestehen u.a. aus einem Rahmen und zwei Rädern. Räder bestehen selbst wieder aus einer Felge und mehreren Speichen. Der Rahmen ist aufgebaut aus den Rohren und dem Lenker. Dieser stark vereinfachte Aufbau eines Fahrrads ist als Entity-Relationship-Diagramm in Abbildung 2.15 gezeigt. Wenn dies aus dem Kontext nicht eindeutig ersichtlich ist, kann man durch Rollen markieren, welches Entity Teilobjekt und welches das übergeordnete Aggregatobjekt ist.

2.11 Kombination von Generalisierung und Aggregation

Die beiden Abstraktionskonzepte Generalisierung und Aggregation können natürlich in einem ER-Schema auch kombiniert zur Strukturierung der Entitytypen eingesetzt werden. Dies wird an dem in Abbildung 2.16 gezeigten Schema verdeutlicht. Hier wird zum einen eine Generalisierungshierarchie mit *Fahrzeuge* als Obertyp aufgebaut. Zum anderen wird für einen der Untertypen von *Fahrzeuge*, nämlich für *Fahrräder*, die Aggregationshierarchie gezeigt. Analog hätte man die Aggregation auch

[2]Dieser Entitytyp wird in der Literatur manchmal auch als Obertyp bezeichnet. Wir vermeiden dies, um den grundlegenden Unterschied zwischen Generalisierung und Aggregation zu betonen.

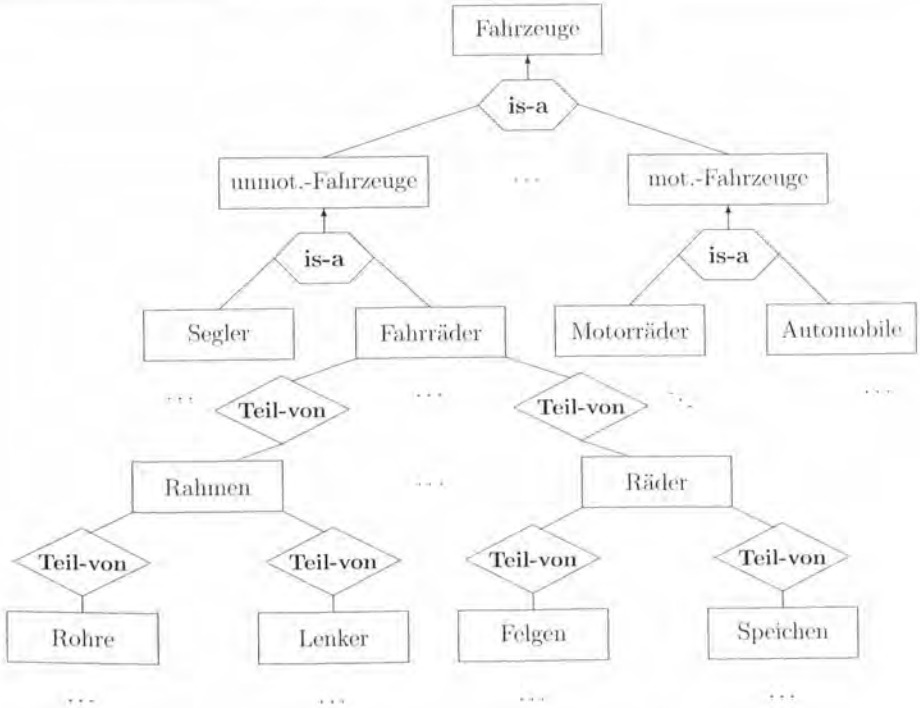


Abbildung 2.16: Zusammenspiel von Generalisierung und Aggregation

für die anderen Fahrzeugarten in das Schema aufnehmen können – aus Platzgründen wurde das hier ausgelassen.

2.12 Konsolidierung, Sichtenintegration

Bei größeren Anwendungen ist es nicht praktikabel, den konzeptuellen Entwurf „in einem Guss" durchzuführen. Es bietet sich an, den konzeptuellen Entwurf (und die vorgelagerte Anforderungsanalyse) gemäß den in der zu modellierenden Organisation vorgegebenen verschiedenen Anwendersichten aufzuteilen. Für unseren Universitätsbereich wären z.B. folgende Sichten denkbar:

1. Professorensicht,
2. Studentensicht,
3. Sicht der Universitätsleitung,
4. Hausmeistersicht,
5. Sicht des Studentenwerks.

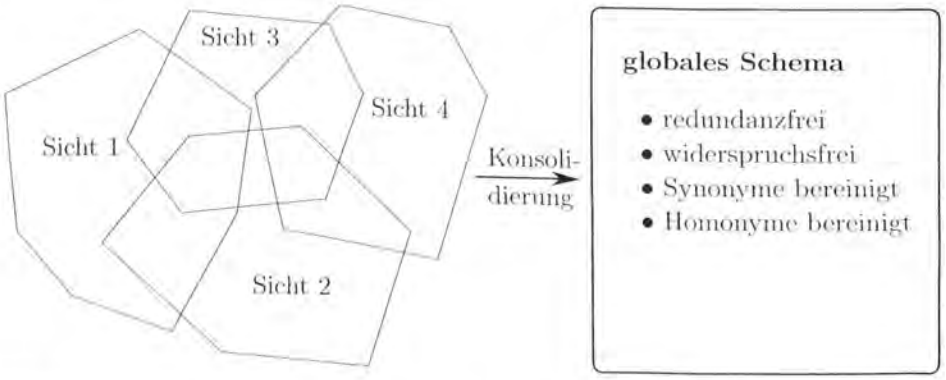


Abbildung 2.17: Veranschaulichung der Konsolidierung überlappender Sichten

Die Datenbankentwerfer müssen in Zusammenarbeit mit den Anwendern der jeweiligen Sicht einen konzeptuellen Entwurf durchführen, der auf deren spezielle Bedürfnisse zugeschnitten ist. Nachdem die einzelnen Sichten modelliert sind, müssen sie aber zu einem globalen Schema zusammengefasst werden, damit man letztendlich zu einem redundanzfreien Datenbankschema gelangt.

Es ist nämlich keinesfalls so, dass die verschiedenen Sichten disjunkte Ausschnitte der realen Welt modellieren. Vielmehr überschneiden sich die Datenbestände der verschiedenen Sichten in mehr oder weniger hohem Ausmaß. Deshalb reicht es nicht aus, die isoliert voneinander entwickelten konzeptuellen Schemata zu vereinen, um daraus ein umfassendes (globales) Schema zu erhalten. Vielmehr muss man bei der Herleitung des globalen Schemas die jeweiligen Teilschemata konsolidieren. Diesen Vorgang bezeichnet man als *Sichtenintegration* oder *Konsolidierung*.

In Abbildung 2.17 wird die Konsolidierung unabhängig voneinander entwickelter Teilschemata zu einem globalen Schema illustriert. Das im Zuge der Konsolidierung entwickelte globale Schema muss redundanzfrei und widerspruchsfrei sein. Widersprüche können sich in den Sichten z.B. durch *Synonyme* – gleiche Sachverhalte wurden unterschiedlich benannt – und *Homonyme* – unterschiedliche Sachverhalte wurden gleich benannt – ergeben. Außerdem können in den Sichten strukturelle Widersprüche oder widersprüchliche Konsistenzbedingungen vorkommen. Ein *struktureller Widerspruch* existiert beispielsweise, wenn derselbe Sachverhalt in einem Schema als Beziehung und im anderen als eigenständiger Entitytyp modelliert ist – vergleiche dazu etwa unsere vorangegangene Diskussion bezüglich der Beziehung *prüfen* und des Entitytyps *Prüfungen*. Eine häufig vorkommende Art eines strukturellen Widerspruchs ergibt sich, wenn ein Sachverhalt in einer Sicht als Attribut und in einer anderen Sicht als Beziehung zu einem Entitytyp modelliert wird. Ein Beispiel dafür ist das Attribut *Raum* des Entitytyps *Professoren* in unserem Universitätsschema, das man auch als Beziehung zu einem Entitytyp *Räume* modellieren könnte. Widersprüche können auch in Bezug auf Konsistenzbedingungen festgestellt werden. Beispielsweise könnte dieselbe Beziehung in unterschiedlichen Sichten widersprüchliche Funktionalitätsangaben haben. Außerdem könnten verschiedene Sichten widersprüchliche Datentypen für dieselben Attribute festlegen oder auch widersprüchliche

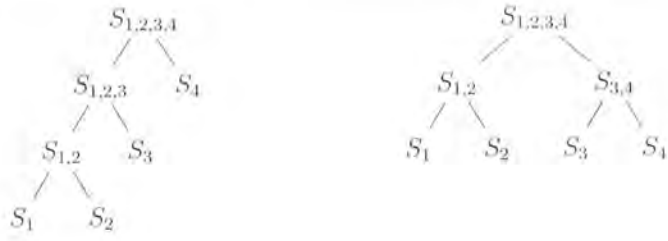


Abbildung 2.18: Mögliche Konsolidierungsbäume zur Herleitung des globalen Schemas $S_{1,2,3,4}$ aus 4 Teilschemata S_1, S_2, S_3 und S_4: links ein maximal hoher und rechts ein minimal hoher Konsolidierungsbaum

Schlüsselattribute spezifizieren. Widersprüche aller Art müssen natürlich in Absprache mit den Datenbankanwendern ausgeräumt werden, damit man letztendlich ein konsistentes Globalschema festlegen kann.

Bei der Konsolidierung einer größeren Anwendung sollte man schrittweise vorgehen, so dass man jeweils nur zwei Teilschemata gleichzeitig betrachtet. Auf diese Weise erhält man einen (binären) Konsolidierungsbaum, dessen Wurzel letztlich das globale konsolidierte Schema repräsentiert. Zwei mögliche Konsolidierungsbäume für vier Teilschemata sind in Abbildung 2.18 gezeigt. Bei dem linken Konsolidierungsbaum wird jeweils ein neues Teilschema in die Konsolidierung mit aufgenommen, so dass der Konsolidierungsbaum die maximale Höhe hat. Eine andere Vorgehensweise besteht darin, zuerst je zwei Teilschemata zu konsolidieren, dann von diesen je zwei zu konsolidieren, usw. Dies führt zu einem minimal hohen Konsolidierungsbaum. Dieser Konsolidierungsbaum ist für die vier Teilschemata in der Abbildung 2.18 rechts gezeigt. Welche dieser beiden Vorgehensweisen (oder einer daraus abgeleiteten „hybriden" Technik) in der Praxis am sinnvollsten ist, hängt sehr stark von der Anwendung, der Anzahl der unabhängig entwickelten Teilschemata und dem Grad der Überlappung der Teilschemata ab.

Wie bereits angemerkt, müssen im Zuge der Konsolidierung Redundanzen und Widersprüche, die sich durch die Vereinigung zweier (oder mehrerer) Teilschemata ergeben würden, bereinigt werden. Weiterhin sollte man bei der Konsolidierung darauf achten, dass eine „saubere" und übersichtliche Strukturierung der Entitytypen erzielt wird. Hierzu können insbesondere die oben beschriebenen Abstraktionskonzepte der Generalisierung und der Aggregation sehr effektiv eingesetzt werden.

Durch die Generalisierung kann man ähnliche Entitytypen, die in unterschiedlichen Teilschemata definiert wurden, zu einem generischen Obertyp zusammenfassen, ohne die Besonderheiten der in den Teilschemata vorhandenen Entitytypen, die dann zu Untertypen werden, zu verwischen. Entweder existiert der generische Obertyp schon in einem der Teilschemata oder er wird im Zuge der Sichtenintegration neu eingeführt. Bei der Generalisierungskonsolidierung muss man insbesondere die Vererbung beachten, so dass man den Untertypen keine Attribute oder Beziehungen zuordnet, die sie schon von Obertypen erben.

Durch Anwendung der Aggregation kann man auf ähnliche Weise übergeordnete, zusammengesetzte Objektstrukturen entwickeln, deren Teilobjekte in den ein-

zelnen Subschemata definiert wurden. In einer betrieblichen Datenbankanwendung wären beispielsweise die Konstruktions- und Fertigungsabteilungen an den Einzelkomponenten eines Produkts interessiert, wohingegen die Marketingabteilung nur die aggregierte Produktbeschreibung interessieren dürfte.

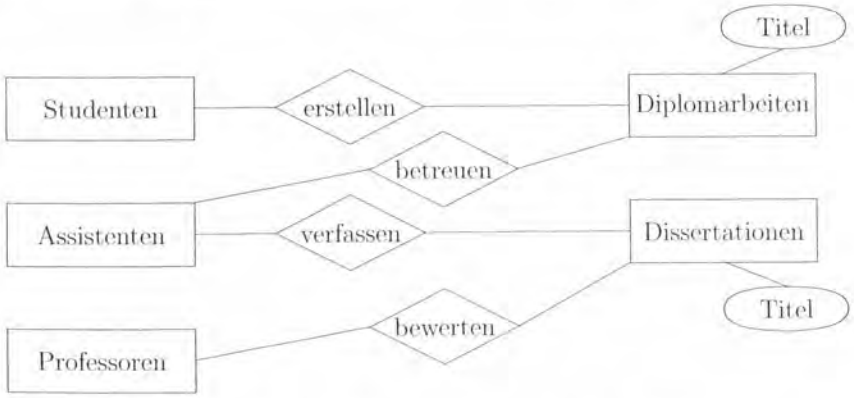
Wir wollen die Sichtenintegration jetzt noch an einem kleinen Beispiel illustrieren. Dazu betrachten wir die drei in Abbildung 2.19 dargestellten Sichten, die sich alle mit Dokumenten innerhalb einer Universität befassen. In der Sicht 1 wird die Erstellung und Betreuung von Diplomarbeiten und Dissertationen modelliert. In Sicht 2 werden die Fakultätsbibliotheken einer Universität konzeptuell modelliert. Sicht 3 legt das Schema für Buchempfehlungen seitens der Dozenten für die jeweiligen Vorlesungen fest.

Folgende Beobachtungen sind für die Konsolidierung der drei Teilschemata wichtig:

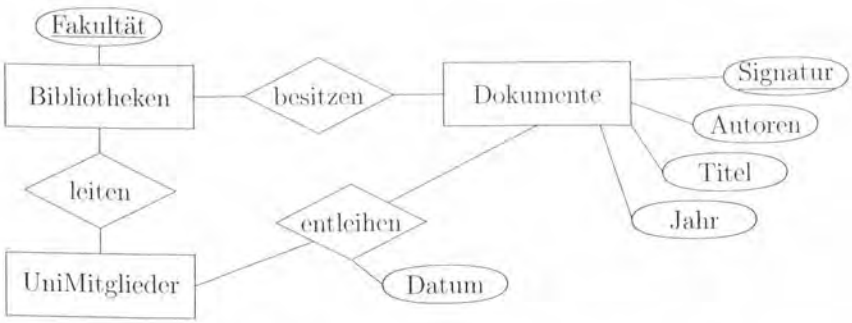
- Die Begriffe *Dozenten* und *Professoren* sind synonym verwendet worden.
- Der Entitytyp *UniMitglieder* ist eine Generalisierung von *Studenten*, *Professoren* und *Assistenten*.
- Fakultätsbibliotheken werden sicherlich von *Angestellten* (und nicht von *Studenten*) geleitet. Insofern ist die in Sicht 2 festgelegte Beziehung *leiten* revisionsbedürftig, sobald wir im globalen Schema ohnehin eine Spezialisierung von *UniMitglieder* in *Studenten* und *Angestellte* vornehmen.
- *Dissertationen*, *Diplomarbeiten* und *Bücher* sind Spezialisierungen von *Dokumenten*, die in den *Bibliotheken* verwaltet werden.
- Wir können davon ausgehen, dass alle an der Universität erstellten *Diplomarbeiten* und *Dissertationen* in *Bibliotheken* verwaltet werden.
- Die in Sicht 1 festgelegten Beziehungen *erstellen* und *verfassen* modellieren denselben Sachverhalt wie das Attribut *Autoren* von *Büchern* in Sicht 3.
- Alle in einer Bibliothek verwalteten Dokumente werden durch die *Signatur* identifiziert.

In Abbildung 2.20 ist ein konsolidiertes Schema dieser drei Sichten dargestellt. Generalisierungen sind zur Vereinfachung der Notation durch fettgedruckte Pfeile, die vom spezialisierten zum generalisierten Entitytyp zeigen, dargestellt. Wir haben also zwei Generalisierungshierarchien eingeführt: eine mit dem „obersten" Obertyp *Personen* als Wurzel und eine mit dem Obertyp *Dokumente* als Wurzel.

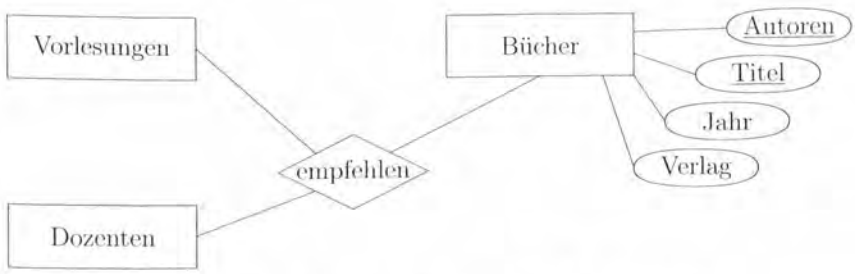
Wir sollten jetzt noch kurz unseren Lösungsvorschlag bezüglich der Redundanz zwischen dem Attribut *Autoren* und den Beziehungen *erstellen* und *verfassen* diskutieren. Wir haben uns dafür entschieden, *Autoren* als Beziehung zwischen *Dokumenten* und *Personen* zu modellieren. Zu diesem Zweck benötigten wir den Entitytyp *Personen*, der *UniMitglieder* generalisiert. Damit sind die beiden Beziehungen *erstellen* zwischen *Studenten* und *Diplomarbeiten* und *verfassen* zwischen *Assistenten* und *Dissertationen* redundant, da dieser Sachverhalt durch die geerbte Beziehung *Autoren* bereits abgedeckt wird. Man muss sich aber auch klar machen, dass bei



Sicht 1: Erstellung von Dokumenten als Prüfungsleistung



Sicht 2: Bibliotheksverwaltung



Sicht 3: Buchempfehlungen für Vorlesungen

Abbildung 2.19: Drei Sichten einer Universitäts-Datenbank

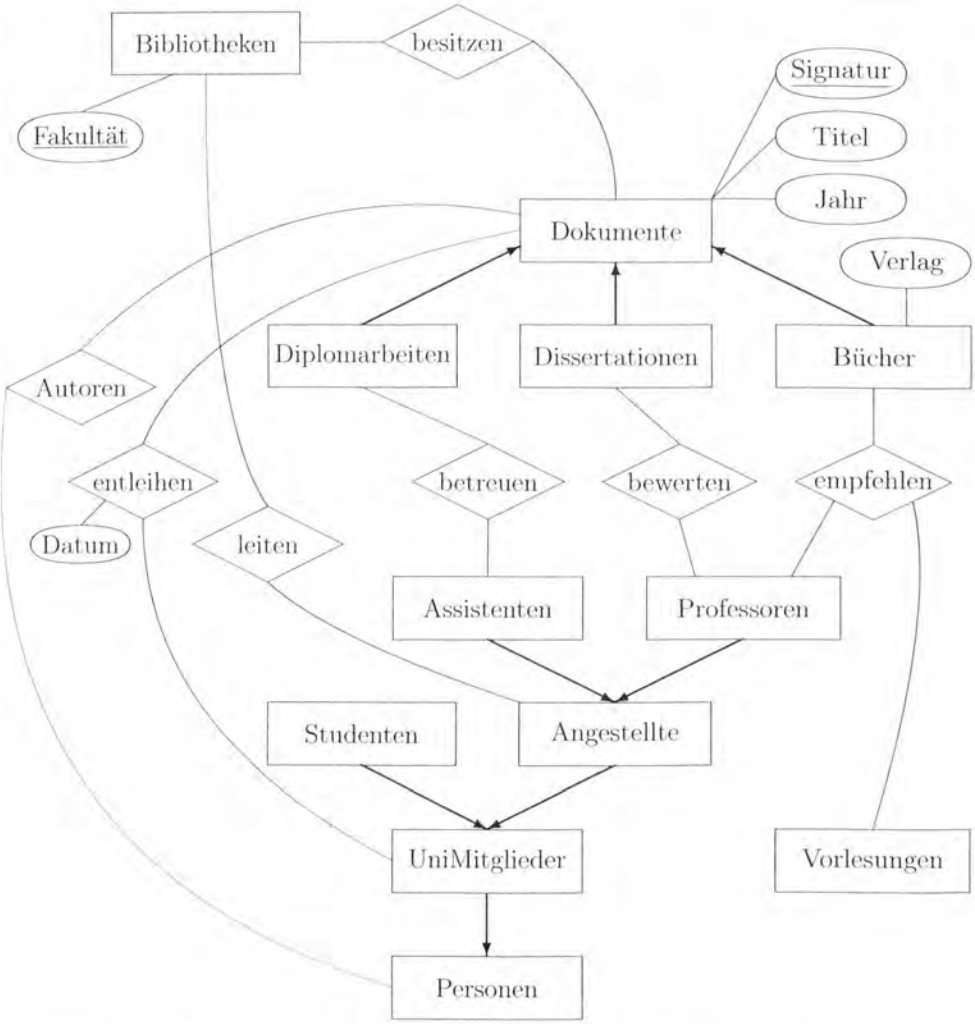


Abbildung 2.20: Konsolidiertes Schema der Universitäts-Datenbank

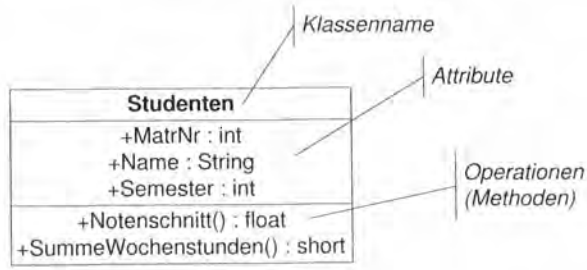


Abbildung 2.21: Eine Beispielklasse *Studenten*

dieser Modellierung etwas an Semantik eingebüßt wurde: Im konsolidierten Schema ist nicht mehr festgelegt, dass *Diplomarbeiten* von *Studenten* geschrieben werden. Wenn man aber beachtet, dass Diplomarbeiten in den Bibliotheken auch noch nach der Exmatrikulation der Diplomanden erhalten bleiben, ist die konsolidierte Modellierung u.U. sogar besser. Analoges gilt für die Verfasser von Dissertationen.

2.13 Konzeptuelle Modellierung mit UML

Im Software-Engineering hat sich die objekt-orientierte Modellierung durchgesetzt, die mittlerweile auch durch viele verfügbare Modellierungswerkzeuge unterstützt wird. Über lange Zeit gab es konkurrierende Modelle für den objektorientierten Softwareentwurf. Glücklicherweise haben sich einige Forscher und Entwickler zusammengetan und einen de-facto Standard in der Form der Unified Modeling Language (UML) entworfen. In diesem Modell gibt es sehr viele Untermodelle für den Entwurf von Softwaresystemen auf den verschiedensten Abstraktionsebenen. Unter anderen gibt es Teilmodelle für die statische Struktur, worunter die Klassenstruktur des Softwaresystems verstanden wird. Das Zusammenspiel von Objekten bzw. deren Operationen in komplexeren Anwendungen lässt sich mit Hilfe von Sequenzdiagrammen beschreiben. Auf einer noch höheren Ebene kann man Anwendungsfälle (use cases) des zu realisierenden Systems grafisch beschreiben. Aktivitäts- und Zustandsdiagramme werden für die Spezifikation von Zustandsübergängen als Folge von ausgeführten Aktivitäten (Operationen, Benutzerinteraktionen) verwendet. Weiterhin gibt es grafische Notationen für die Zerlegung des Systems in Teilsysteme (Komponenten, Packages).

Für den Datenbankentwurf ist die strukturelle Modellierung des Systems bestehend aus Objektklassen und Assoziationen zwischen den Klassen am wichtigsten. Objekte entsprechen den Entities und Objektklassen beschreiben eine Menge von gleichartigen Objekten/Entities. Zusammenhänge (Beziehungen) zwischen Objekten werden als Assoziationen zwischen den Klassen beschrieben.

2.13.1 UML-Klassen

Anders als im Entity/Relationship-Modell beschreibt eine Klasse nicht nur die strukturelle Repräsentation (Attribute) der Objekte, sondern auch deren Verhalten in der

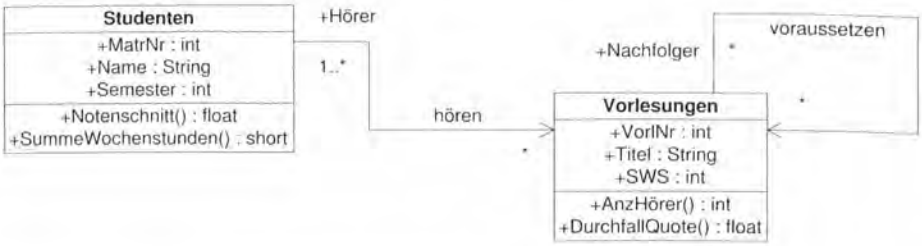


Abbildung 2.22: Assoziationen zwischen den Klassen *Studenten* und *Vorlesungen*

Form von zugeordneten Operationen. Wir wollen uns gleich die in Abbildung 2.21 gezeigte Beispielklasse *Studenten* anschauen: Demnach wird der Zustand von Objekten der Klasse *Studenten* durch die Attribute *MatrNr*, *Name* und *Semester* repräsentiert. Weiterhin sind der Klasse *Studenten* exemplarisch die beiden Operationen *Notenschnitt()* und *SummeWochenstunden()* zugeordnet. Erstere ermittelt aus den abgelegten Prüfungen (siehe unten) die Durchschnittsnote und letztere errechnet aus den belegten Vorlesungen die Anzahl der Semesterwochenstunden.

In UML unterscheidet man zwischen öffentlich sichtbaren (mit + gekennzeichneten), privaten (mit – gekennzeichneten) und in Unterklassen sichtbaren (mit # gekennzeichneten) Attributen bzw. Operationen. Beim Datenbankentwurf sind in der Regel alle Attribute sichtbar, da man den Zugriff auf Attribute auf einer detaillierteren Ebene über die Autorisierung des Datenbanksystems steuert.

In UML fehlt das Konzept eines Schlüssels, da Objekte immer einen systemweit eindeutigen Objektidentifikator zugeordnet bekommen. Dieser Objektidentifikator bleibt während der gesamten Lebenszeit des Objekts invariant und kann dadurch zum einen zur eindeutigen Identifikation und zum anderen auch zur Realisierung von Verweisen auf das Objekt benutzt werden.

2.13.2 Assoziationen zwischen Klassen

Den Beziehungstypen im Entity/Relationship-Modell entsprechen die Assoziationen zwischen Klassen in UML. Als Beispiel zeigen wir in Abbildung 2.22 die binären Assoziationen *hören* und *voraussetzen*. Zusätzlich zum Assoziationsnamen haben wir auch die Rollen angegeben: *Studenten* haben in der Assoziation *hören* die Rolle der *Hörer*. Wichtiger ist die Rollenspezifikation bei der rekursiven Assoziation *voraussetzen*. Hierbei fungiert eine Vorlesung als *Nachfolger*, die andere ist dann implizit (ohne dass wir dies nochmals explizit angegeben haben) die Vorgänger-Lehrveranstaltung.

Anders als im Entity/Relationship-Modell kann man Assoziationen in UML eine Richtung zuordnen. Dadurch wird angegeben, in welcher Richtung man auf die assoziierten Objekte zugreifen kann. In unserem Beispiel wurde angegeben, dass man von einer/m Student/in aus die gehörten Vorlesungen ermitteln kann; umgekehrt kann man von einer Vorlesung aus die hörenden Studenten nicht (so leicht) ermitteln. Bei der Assoziation *voraussetzen* ist es analog: Von einer Vorlesung kann man zu den Voraussetzungen (also Vorgänger-Vorlesungen) traversieren. Wir möchten aber

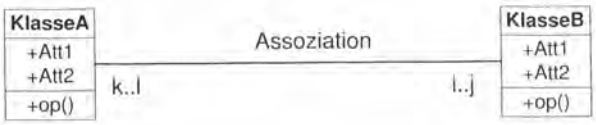


Abbildung 2.23: Die Multiplizität einer abstrakten Assoziation

schon jetzt darauf hinweisen, dass diese Traversierungsrichtung in den Datenbanken eine untergeordnete Rolle spielt, da man in Datenbankanfragen eine Assoziation immer in beiden Richtungen traversieren kann. Bei der objekt-orientierten Programmierung ist das anders, da man dort die Assoziationen als Referenzen auf das/die assoziierten Objekte modelliert. Diese Referenzen würden dann gemäß der Traversierungsrichtung in dem Objekt abgespeichert, von dem die gerichtete Assoziation ausgeht.

Ähnlich wie im Entity/Relationship-Modell kann man in UML die Beziehungen (Assoziationen) genauer beschreiben. In UML-Terminologie bezeichnet man dies als *Multiplizität* der Assoziation. In unserem Beispiel *hören* wurde diese als 1..* auf der Seite der *Studenten* und als * auf der Seite der *Vorlesungen* angegeben. Die Intervallangabe 1..* bedeutet, dass eine Vorlesung mindestens 1 Hörer/in und maximal beliebig viele Hörer hat. Studenten hören beliebig viele Vorlesungen, was mit dem * auf der Seite der Vorlesungen angegeben ist.

Betrachten wir das abstrakte Beispiel in Abbildung 2.23: Wenn man in UML an einer Seite der binären Assoziation die Multiplizitätsangabe $i..j$ macht, so bedeutet dies, dass jedes Objekt der Klasse auf der anderen Seite mit mindestens i und höchstens j Objekten der Klasse auf dieser Seite in Beziehung stehen muss. Bezogen auf unser Beispiel bedeutet dies, dass jedes Objekt der *KlasseA* mit mindestens i und mit maximal j Objekten der *KlasseB* in Beziehung stehen muss/darf. Analog muss jedes Objekt der *KlasseB* mit mindestens k Objekten der *KlasseA* in Beziehung stehen und es darf maximal mit l Objekten der *KlasseA* in dieser Beziehung stehen. Wenn die minimale und die maximale Anzahl übereinstimmt, also $m..m$ gilt, so vereinfacht man dies in UML zu einem einzigen Wert m. Dies gilt auch für $0..*$ was meist als $*$ angegeben wird.

Man beachte, dass die Multiplizitätsangabe in UML sich an die Funktionalitätsnotation des ER-Modells anlehnt; nicht an die (min,max)-Notation. Anders als die UML-Multiplizitätsangabe bezog sich die (min,max)-Angabe darauf, wie oft jedes Entity „auf *dieser* Seite" an der Beziehung teilnehmen muss/darf. In Übungsaufgabe 2.12 soll der Zusammenhang detailliert herausgearbeitet werden.

2.13.3 Aggregation in UML

Als besondere Form von Beziehungen/Assoziationen hatten wir im ER-Modell schon die Aggregation eingeführt. In UML werden zwei Arten der Aggregation unterschieden: Die „normale" Teil/Ganzes-Beziehung sowie die Komposition. Die exklusive Zuordnung von existenzabhängigen Teil-Objekten zu *einem* übergeordneten Objekt wird als *Komposition* bezeichnet. In Abbildung 2.24 wird die Zuordnung der *Prüfungen* zu der einen Studentin bzw. dem einen *Studenten* – dem Prüfling – ge-

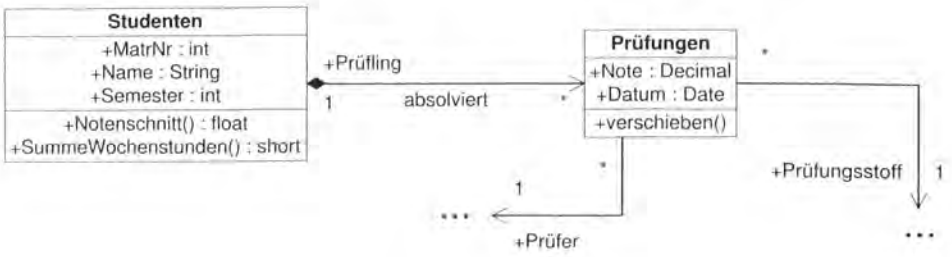


Abbildung 2.24: Die Aggregation zwischen *Studenten* und *Prüfungen*

zeigt. Diese exklusive Zuordnung existenzabhängiger Unterobjekte wird in UML mit der ausgefüllten Raute auf der Seite der übergeordneten Klasse angegeben. Wegen der Existenzabhängigkeit und der Exklusivität muss auf der Seite der übergeordneten Objektklasse immer die Multiplizität 1 (was ja dem Intervall 1..1 entspricht) angegeben sein, da jedes untergeordnete Objekt wegen der Existenzabhängigkeit mindestens einem übergeordneten Objekt zugeordnet sein muss und wegen der Exklusivität maximal einem übergeordneten Objekt zugeordnet sein darf.

Die exklusive, existenzabhängige Aggregation entspricht im Entity/Relationship-Modell der Beziehung zwischen einem schwachen Entitytyp und dem übergeordneten Entitytyp, wie wir dies für genau dasselbe Beispiel der *Prüfungen* in Abschnitt 2.8 diskutiert haben.

2.13.4 Anwendungsbeispiel: Begrenzungsflächendarstellung von Polyedern in UML

Neben der exklusiven, existenzabhängigen Aggregation gibt es in UML noch eine schwächere Form der Aggregation, die zwar eine *besteht aus*-Beziehung darstellt, wobei aber die Unterobjekte nicht exklusiv zugeordnet sein müssen. Am besten macht man sich diesen Unterschied an einem Beispiel klar: In Abbildung 2.25 ist die Begrenzungsflächendarstellung für Polyeder als UML-Klassendiagramm gezeigt. Die Aggregation *Hülle* ist eine exklusive Zuordnung von Flächen zu Polyedern. Andererseits kann die Assoziation von *Kanten* zu *Flächen* nicht exklusiv sein, da sich immer 2 Flächen dieselbe Kante teilen. In der anderen Richtung der Assoziation gilt, dass eine Fläche mindestens 3 Kanten hat. Analog zur Aggregation *Begrenzung* gilt für die Aggregationsassoziation *StartEnde*, dass die Zuordnung von *Punkten* zu *Kanten* nicht exklusiv ist, da sich mindestens 3 Kanten eines Polyeders einen Punkt teilen. Jede Kante wird von genau 2 Punkten begrenzt.

In diesem Beispiel haben wir den Objektklassen auch jeweils einige Operationen zugeordnet. Es ist zu erkennen, dass der Objektklasse *Polyeder* die Operationen *skalieren()*, *rotieren()* und *verschieben()* zugeordnet sind, die es auch bei der Objektklasse *Punkte* gibt. Hierbei handelt es sich natürlich nicht um Vererbung, sondern um einfaches Überladen (Gleichbenennung) der Operationen, die semantisch ähnlich sind. In der Tat würde man diese Operationen des übergeordneten Objekttyps

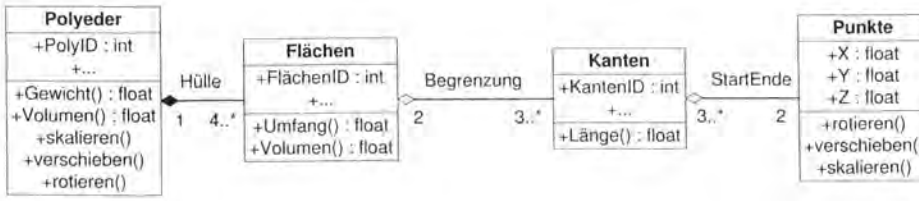


Abbildung 2.25: Die Begrenzungsflächendarstellung von Polyedern

Polyeder auf der Basis der gleichbenannten Operationen des untergeordneten Objekttyps *Punkte* realisieren. Beispielsweise verschiebt man einen Polyeder, indem man jeden Begrenzungspunkt verschiebt. Man bezeichnet dies – bezogen auf die Implementierung der *verschieben()*-Operation des *Polyeders* – auch als *Delegation*.

2.13.5 Generalisierung in UML-Notation

Wir haben Generalisierung bzw. Spezialisierung schon als Abstraktionsmittel für die Strukturierung von Entitytypen kennengelernt. Zusammen mit der Vererbung von Attributen und Operationen spielen Generalisierungs-/Spezialisierungshierarchien eine wichtige Rolle, um wiederverwendbare Objektklassen zu entwerfen. Wir wollen uns wiederum auf ein Beispiel konzentrieren: In Abbildung 2.26 ist die Generalisierung von *Assistenten* und *Professoren* zu *Angestellten* gezeigt. Dieser Generalisierungs-Klasse *Angestellte* sind 2 Attribute, *Name* und *PersNr*, und 2 Operationen, *Gehalt()* und *Steuern()*, zugeordnet. Sowohl die beiden Attribute als auch die beiden Operationen werden von den spezialisierten Klassen, den sogenannten Unterklassen, *Professoren* und *Assistenten* geerbt. Als Besonderheit zeigen wir hier noch, dass die Operation *Gehalt* in den Unterklassen neu definiert wird – wir sagen, sie wird in den Unterklassen *verfeinert*, um je nach Objektklasse das Gehalt auf spezialisierte Weise zu berechnen. Bei den *Steuern* ist das nicht nötig, da für alle *Angestellten* dieselbe Steuerberechnungsformel angewendet wird.

2.13.6 Die Modellierung der Universität in UML

In Abbildung 2.27 ist das konzeptuelle Schema der Universitätsanwendung als UML-Klassendiagramm gezeigt. Mit der Ausnahme der Operationen (die im ER-Modell fehlen) und der Schlüsselspezifikationen (die im UML-Modell fehlen) ist das resultierende Schema dem Entity/Relationship-Schema, das wir in Abbildung 2.14 entwickelt hatten, recht ähnlich. Es ist im Wesentlichen eine „Geschmacksfrage" ob man den Datenbankentwurf lieber in der Form eines „puristischen" ER-Modells oder eines ausdrucksmächtigeren UML-Modells durchführen möchte. Wenn man allerdings für die weitergehende Anwendungsprogramm-Entwicklung ohnehin ein UML-basiertes Entwurfswerkzeug einsetzt, so empfiehlt es sich sicherlich, alle Entwurfsaufgaben (Datenbank- und Software-Entwurf) mit Hilfe der UML-Methode zu vereinheitlichen.

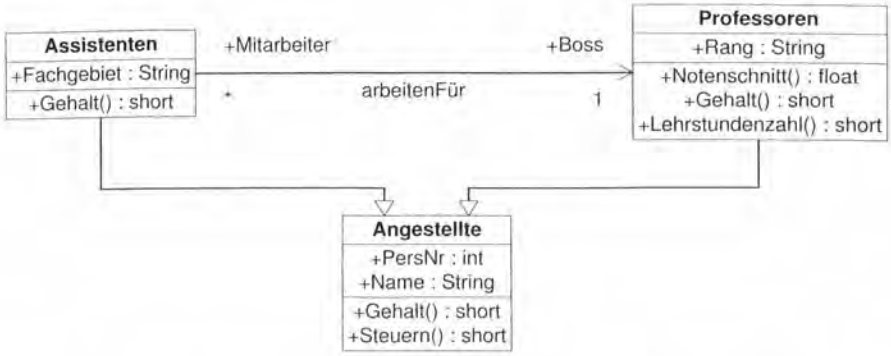


Abbildung 2.26: Die Generalisierung/Spezialisierung in UML-Notation

2.13.7 Verhaltensmodellierung in UML

Zum Abschluss der UML-Diskussion wollen wir noch kurz auf die weitergehenden Möglichkeiten von UML zur Modellierung komplexer Anwendungsszenarien und Objektinteraktionen eingehen. Die bisher durchgeführte strukturelle Modellierung von Objektklassen dient als Grundlage für die Implementierung von komplexeren Funktionsabläufen, bei der u.U. viele Objekte unterschiedlichen Typs miteinander kooperieren.

2.13.8 Anwendungsfall-Modellierung (use cases)

UML bietet auch für die dem Software-Entwurf vorgelagerte Phase der Anforderungsanalyse eine grafische Modellierungsmethode zur Definition von *Anwendungsfällen* (engl. *use cases*). Darin werden die wesentlichen Komponenten des zu erstellenden Informationssystems sehr abstrakt identifiziert. „Use Cases" sollen typische Anwendungssituationen dokumentieren, die nach der Anforderungsanalyse die weitere Softwareentwicklung begleiten. Die grafische Darstellung der „Use Cases" erleichtert die Kommunikation mit den späteren Anwendern (Kunden) des Systems und fördert das Verständnis der Entwickler für die spezielle Anwendung. Erste Schritte lassen sich besser mit konkreten Beispielen beschreiben als mit abstrakteren allgemeinen Fällen. Während der Modellierung und Softwareentwicklung dienen sie als Testfälle, um die Tauglichkeit des Systems zu überprüfen.

Das Augenmerk bei der Modellierung von Anwendungsfällen liegt auf der Identifikation der Akteure, die in diesen Anwendungsfällen miteinander bzw. mit dem System interagieren. Wir haben dies exemplarisch und auf einer sehr hohen Abstraktionsstufe für unser Universitäts-Informationssystem in Abbildung 2.28 dargestellt. Hier sind die bei der Abhaltung von Vorlesungen und Prüfungen interagierenden Akteure, nämlich die *Studenten*, *Professoren* und *Assistenten*, hervorgehoben.

Natürlich muss die gezeigte grafische Darstellung der Anwendungsfälle durch entsprechenden erläuternden Text begleitet werden, damit die intendierte Funktionalität des Softwaresystems hinreichend genau beschrieben ist.

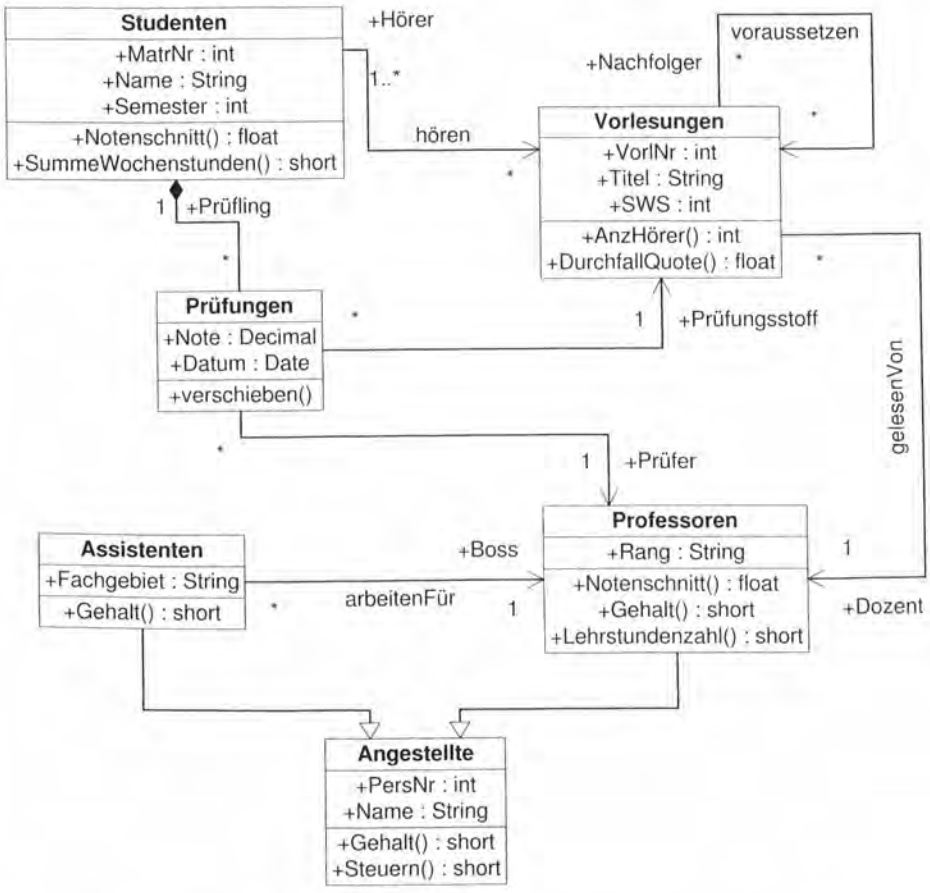


Abbildung 2.27: Die konzeptuelle Modellierung der Universität in UML

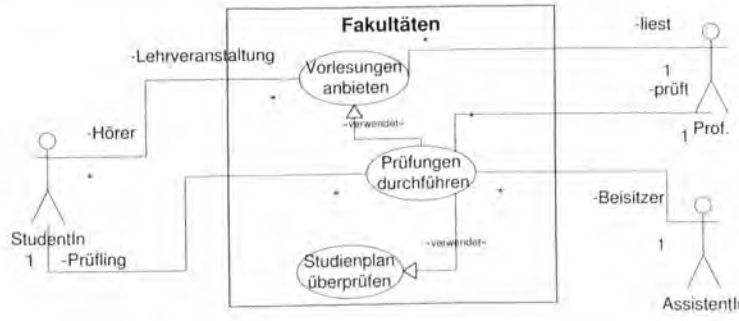


Abbildung 2.28: Identifizierung von Anwendungsfällen und Akteuren

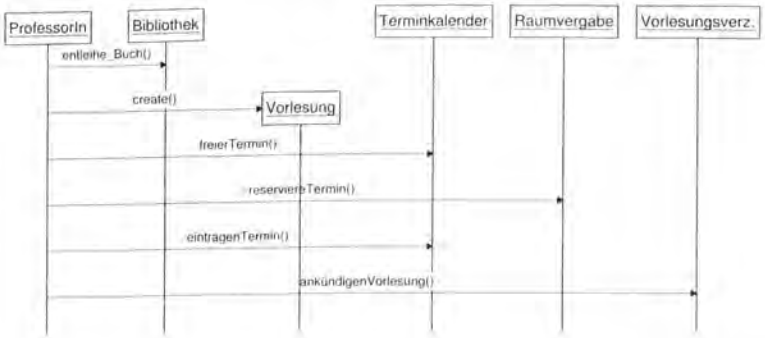


Abbildung 2.29: Das UML-Interaktionsdiagramm zur Planung einer neuen Vorlesung

2.13.9 Interaktionsdiagramme

Die Anwendungsfall-Modellierung (use cases) war auf einer sehr abstrakten Ebene, die die Nutzung des Informationssystems betont. Nachdem in der darauffolgenden Entwurfsphase die Objekttypen und deren Beziehungen festgelegt wurden, können die informell spezifizierten Use Cases detaillierter modelliert werden. Dazu gibt es in UML die sogenannten *Objekt-Interaktionsdiagramme*. Dabei wird das Zusammenwirken von Objekten verdeutlicht. Einen Teilausschnitt des Use Cases aus dem letzten Abschnitt, nämlich die Planung einer neuen Vorlesung, ist in Abbildung 2.29 dargestellt. Eine Verbindung zwischen zwei Objekten bezeichnet einen Kommunikationsvorgang. Die Richtung der Kommunikation wird durch einen Pfeil angedeutet. Bei dieser Darstellung wird die Ablaufreihenfolge stark betont. Die am Ablauf beteiligten Objekte werden horizontal nebeneinander dargestellt, die Operationen nach ihrer Reihenfolge vertikal von oben nach unten. Die gedachte Zeitachse verläuft also vertikal von oben nach unten.

2.13.10 Interaktionsdiagramm zur Prüfungsdurchführung

Wir wollen die Modellierung komplexer Anwendungsabläufe mit Hilfe von Interaktionsdiagrammen auch auf den zweiten Teilbereich des in Abbildung 2.28 dargestellten Anwendungsfalls, der Durchführung von Prüfungen, demonstrieren. Diese Interaktion wird durch die Anmeldung eines/r Student/in initiiert. Vom Prüfungsamt wird die zu prüfende Vorlesung „befragt", um die/den lesenden Professor/in für die Prüfung zu benachrichtigen. Professoren melden freie Termine, die dann bestätigt und dem Prüfling mitgeteilt werden. Das neue Objekt vom Typ *Prüfung* wird vom Prüfungsamt kreiert. Assistenten werden von den Professoren für den Beisitz ausgewählt. Nach der Bewertung der Prüfung sind sie für die Protokollierung zuständig.

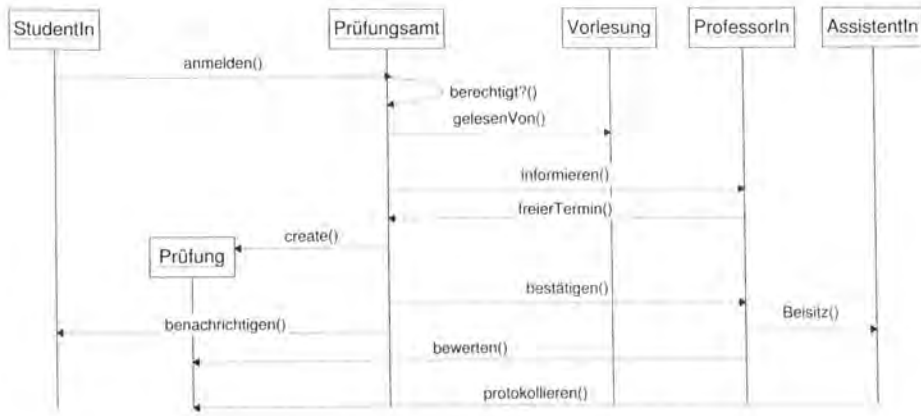


Abbildung 2.30: Das UML-Interaktionsdiagramm zur Anmeldung und Durchführung einer Prüfung

2.14 Übungen

2.1 Charakterisieren Sie die 1:1-, 1:N-, N:1- und N:M-Beziehungstypen mittels der (min, max)-Notation. Für eine abstrakte binäre Beziehung R zwischen den beiden Entitytypen E_1 und E_2 sollen jeweils die (min_1, max_1)- und (min_2, max_2)-Wertepaare angegeben werden, die sich aus den (gröberen) Funktionalitätsangaben herleiten lassen.

2.2 Zeigen Sie, dass die Ausdruckskraft der Funktionalitätsangaben und der (min, max)-Angaben bei n-stelligen Beziehungen mit $n > 2$ unvergleichbar ist: Finden Sie realistische Beispiele von Konsistenzbedingungen, die mit Funktionalitätsangaben, aber nicht mit (min,max)-Angaben ausdrückbar sind, und wiederum andere Konsistenzbedingungen, die mit der (min,max)-Angabe formulierbar sind aber nicht durch Funktionalitätseinschränkungen.

2.3 Beim konzeptuellen Entwurf hat man gewisse Freiheitsgrade hinsichtlich der Modellierung der realen Welt. Unter anderem hat man folgende Alternativen, die Sie an unserem Universitätsschema beispielhaft illustrieren sollten:

- Man kann ternäre Beziehungen in binäre Beziehungen transformieren. Betrachten Sie dazu die Beziehung *prüfen* und erläutern Sie die Vor- und Nachteile einer solchen Transformation.
- Man hat manchmal die Wahl, ein Konzept der realen Welt als Beziehung oder als Entitytyp zu modellieren. Erörtern Sie dies wiederum am Beispiel der Beziehung *prüfen* im Gegensatz zu einem eigenständigen Entitytyp *Prüfungen*.

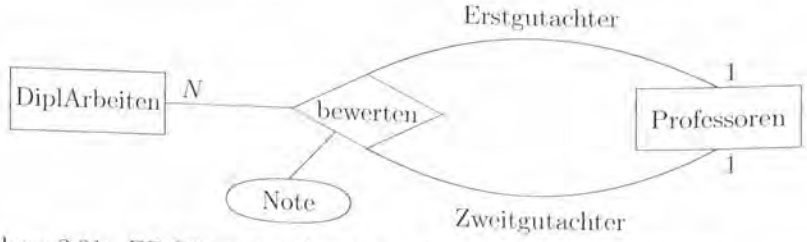


Abbildung 2.31: ER-Diagramm der dreistelligen Beziehung *bewerten*

- Ein Konzept der realen Welt kann manchmal als Entitytyp mit zugehörigem Beziehungstyp und manchmal als Attribut dargestellt werden. Ein Beispiel hierfür ist das Attribut *Raum* des Entitytyps *Professoren* in unserem Schema aus Abbildung 2.14. Diskutieren Sie die Alternativen.

2.4 In Abbildung 2.31 ist die dreistellige Beziehung *bewerten* zwischen den Entitytypen *DiplArbeiten*, *Professoren* in der Rolle als *Erstgutachter* und *Professoren* in der Rolle als *Zweitgutachter* grafisch dargestellt. Gemäß obiger Erläuterung kann man die Beziehung *bewerten* demnach als partielle Funktionen wie folgt auffassen:

$$\begin{aligned} \textit{bewerten} &: \textit{DiplArbeiten} \times \textit{Erstgutachter} \rightarrow \textit{Zweitgutachter} \\ \textit{bewerten} &: \textit{DiplArbeiten} \times \textit{Zweitgutachter} \rightarrow \textit{Erstgutachter} \end{aligned}$$

Diskutieren Sie, ob man diese Beziehung auch durch (mehrere) zweistellige Beziehungen modellieren kann, ohne dass ein Semantikverlust auftritt.

2.5 Finden Sie eine dreistellige $1:1:1$-Beziehung aus dem Kontext einer Universitätsverwaltung. Es sollte eine dreistellige Beziehung sein, die nicht durch (mehrere) zweistellige Beziehungen dargestellt werden kann. Unter welchen Bedingungen ist dies der Fall?

2.6 Modellieren Sie ein Zugauskunftsystem, in dem die wichtigsten Züge (z.B. die Intercity- und Eurocity-Züge) repräsentiert werden. Aus dem System sollen die Start- und Zielbahnhöfe und die durch den Zug verbundenen Bahnhöfe einschließlich Ankunfts- und Abfahrtszeiten ersichtlich sein. Geben Sie die Funktionalitäten der Beziehungstypen an.

2.7 Erweitern Sie das in Aufgabe 2.6 erstellte Modell um die Personaleinsatzplanung. Insbesondere sollten Sie die Zugführer, deren Stellvertreter, die IC-Chefs, die Schaffner, die Köche und die Kellner in Ihr Schema aufnehmen. Weiterhin muss man die Zusammenstellung der „Mannschaft" – IC/EC-Team genannt – für die einzelnen Züge vornehmen können. Verwenden Sie bei der Modellierung das Abstraktionskonzept der Generalisierung bzw. Spezialisierung.

2.8 Modellieren Sie die Grundlagen eines Krankenhausverwaltungssystems. Insbesondere sollten die Patienten, deren Stationen, deren Zimmer, die behandelnden Ärzte und die betreuenden Pfleger modelliert werden. Verwenden Sie wiederum die Generalisierung zur Strukturierung Ihrer Entitytypen.

2.9 Im Text hatten wir hervorgehoben, dass die Beziehung zwischen einem schwachen Entitytyp und dem starken Entitytyp, von dessen Existenz die schwachen Entities abhängig sind, keine N:M-Beziehung sein kann. Erläutern Sie, warum das so ist. Denken Sie an die Existenzabhängigkeit und die Identifikation der schwachen Entities. Geben Sie etliche Beispiele schwacher Entitytypen und charakterisieren Sie die Beziehung zu den starken Entitytypen.

2.10 Schwache Entitytypen kann man immer auch als „starke" (normale) Entitytypen modellieren. Was muss dabei beachtet werden? Erläutern Sie dies am Beispiel aus Abbildung 2.11.

2.11 Modellieren Sie ein Auto, wobei Sie besonders auf die Aggregation, d.h. die **part-of** Beziehungen eingehen. Welcher Zusammenhang besteht zwischen dem Konzept der schwachen Entities und einer Aggregationshierarchie?

2.12 Man beachte, dass die Multiplizitätsangaben in UML sich an die Funktionalitätsangaben des ER-Modells anlehnen; nicht an die (*min*,*max*)-Notation. Arbeiten Sie den Zusammenhang detailliert heraus, d.h. illustrieren Sie die UML-Multiplizitäten, die sich für $1:1$, $1:N$ bzw. $N:M$-Beziehungen ergeben.

2.13 Modellieren Sie das Verwaltungssystem für eine Leichtathletik-WM.

2.14 Modellieren Sie das in Aufgabe 1.3 eingeführte Wahlinformationssystem.

2.15 Literatur

Das Entity-Relationship-Modell mit der grafischen Notation wurde in einem wegweisenden Aufsatz von Chen (1976) eingeführt. Die ersten und wichtigsten Arbeiten zur Generalisierung und Aggregation wurden für den Datenbankbereich von Smith und Smith (1977) durchgeführt. Aufbauend auf dieser Arbeit klassifizierten Batory und Buchmann (1984) komplexere Entitytypen als molekulare Objekte.

Es gibt mehrere Erweiterungsvorschläge für das ER-Modell; z.B. schlugen Teorey, Yang und Fry (1986) die Erweiterung um Generalisierung und Aggregation vor. Für den konzeptuellen Datenbankentwurf wurden mehrere andere Datenmodelle konzipiert, die aber nicht die gleiche praktische Bedeutung erlangt haben. Diese Modelle werden oftmals als *semantische Datenmodelle* bezeichnet, weil sie die Bedeutung (Semantik) der Anwendungsobjekte in natürlicher Weise zu modellieren gestatten. Das von Hammer und McLeod (1981) vorgeschlagene SDM (semantic data model) ist ein solches Modell. Weitere semantische Datenmodelle werden in einem Übersichtsaufsatz von Hull und King (1987) abgehandelt. Abiteboul und Hull (1987) konzipierten das formale semantische Datenmodell IFO. Karl und Lockemann (1988) beschreiben ein semantisches Datenmodell, das sich anwendungsspezifisch erweitern lässt. Hohenstein und Engels (1992) haben eine auf dem Entity-Relationship-Modell basierende Anfragesprache vorgeschlagen.

In einem Artikel von Liddle, Embley und Woodfield (1993) werden eine Vielzahl von semantischen Datenmodellen im Hinblick auf ihre Fähigkeiten, Kardinalitätsvorgaben für Beziehungstypen auszudrücken, untersucht. Es wird dort auch die Er-

weiterung der Funktionalitätsangaben im ER-Modell auf mehrstellige Beziehungen erläutert. Lockemann et al. (1992) schlugen ein Entwurfsmodell mit frei definierbaren Modellierungskonzepten vor.

Von Tjoa und Berger (1993) wird die Umsetzung einer natürlichsprachigen Anforderungsspezifikation in ein erweitertes ER-Modell diskutiert.

Es gibt einige dedizierte Lehrbücher, die sich mit dem konzeptuellen Datenbankentwurf befassen, z.B. von Teorey (1994) oder von Batini, Ceri und Navathe (1992). Auch ist im Datenbank-Handbuch von Lockemann und Schmidt (1987) ein Kapitel zur konzeptuellen Datenmodellierung von Mayr, Dittrich und Lockemann (1987) enthalten. Weiterhin enthalten die Bücher von Dürr und Radermacher (1990) und Lang und Lockemann (1995) sehr praxisorientierte Kapitel zum Datenbankentwurf.

Das Thema der Sichtenkonsolidierung (oder Sichtenintegration) konnten wir hier nur anreißen. Mehr darüber findet sich in den oben angeführten Lehrbüchern zum Datenbankentwurf; eine formalere Behandlung der Sichtenintegration wurde von Biskup und Convent (1986) vorgestellt.

Im Bereich der objektorientierten Datenmodellierung wurden Erweiterungen des Entity-Relationship Modells postuliert. Die bekannteren Methoden sind die von Rumbaugh et al. (1991) vorgeschlagene OMT-Technik sowie das von Booch (1991) propagierte Modell. Daraus hat sich die Modellierungssprache UML als Standard entwickelt, die von Booch, Rumbaugh und Jacobson (1998) beschrieben wurde. Oestereich und Bremer (2009) haben ein deutschsprachiges Lehrbuch über die objektorientierte Modellierung mit UML verfasst. Das Buch von Muller (1999) beleuchtet die UML-Modellierung im Kontext des Datenbankentwurfs. Kappel und Schrefl (1988) und Eder et al. (1987) haben die Erweiterung der ER-Modellierung um objektorientierte Konzepte vorgeschlagen. Hartel et al. (1997) berichten über Erfahrungen mit dem Einsatz formaler Spezifikationsmethoden beim Datenbankentwurf. Richters und Gogolla (2000) und Stumptner und Schrefl (2000) beschäftigen sich mit der formalen Validierung von UML-Entwürfen. Preuner, Conrad und Schrefl (2001) behandeln die Sichtenintegration in objektorientierten Datenbanken. Artale und Franconi (1999) benutzen logikbasierte Formalismen für die konzeptuelle Modellierung temporaler Zusammenhänge. Thalheim (2000) hat ein sehr umfangreiches Buch über das Entity/Relationship-Modell geschrieben. Kemper und Wallrath (1987) haben den konzeptuellen Datenbankentwurf für Computergeometrie-Anwendungen untersucht. Oberweis und Sander (1996) modellieren das Verhalten von Informationssystemen auf der Basis von Petrinetzen.

Es gibt etliche kommerziell verfügbare Produkte für den rechnergestützten Datenbankentwurf. Zu den bekannteren zählen *ERwin* der Firma Logic-Works (1997) und *PowerDesigner* (früher *S-Designor*) von der Firma Powersoft (1997). Auch Visio von Microsoft unterstützt die ER-Modellierung. Weiterhin verfügen viele Datenbankprodukte über entsprechende Module, die den Datenbankentwurf unterstützen. Für den objektorientierten UML-Entwurf hat sich das Produkt Rose von der Firma Rational Software Corporation (1997) (jetzt IBM) etabliert. Die objekt-orientierte Entwurfsmethodik ist sehr detailliert in Brügge und Dutoit (2004) abgehandelt.

3. Das relationale Modell

Anfang der siebziger Jahre wurde das relationale Datenmodell konzipiert. Die Besonderheit dieses Datenmodells besteht in der *mengenorientierten* Verarbeitung der Daten im Gegensatz zu den bis dahin vorherrschenden *satzorientierten* Datenmodellen, nämlich dem Netzwerkmodell und dem hierarchischen Modell. In diesen beiden letztgenannten Modellen – die heute fast nur noch historische Bedeutung haben – werden die Informationen auf Datensätze, die miteinander über Referenzen verknüpft sind, abgebildet. Die Verarbeitung der Daten erfolgt dann, indem man von einem Datensatz zum nächsten über diese Referenzen „navigiert".

Das relationale Datenmodell ist im Vergleich zu den satzorientierten Modellen sehr einfach strukturiert. Es gibt im wesentlichen nur flache Tabellen (Relationen), in denen die Zeilen den Datenobjekten entsprechen. In dieser sehr einfachen – fast schon spartanischen – Struktur liegt aber wahrscheinlich der Erfolg der relationalen Datenbanktechnologie begründet, die heute eine marktdominierende Stellung besitzt.

Die in den Tabellen (Relationen) gespeicherten Daten werden durch entsprechende Operatoren ausschließlich mengenorientiert verknüpft und verarbeitet.

3.1 Definition des relationalen Modells

3.1.1 Mathematischer Formalismus

Gegeben seien n nicht notwendigerweise unterschiedliche *Wertebereiche* (auch *Domänen* genannt) $D_1, D_2, \ldots, D_n$, d.h. $D_i = D_j$ ist durchaus zulässig für $i \neq j$. Diese Domänen dürfen nur *atomare* Werte enthalten, die nicht strukturiert sein dürfen. Gültige Domänen sind z.B. Zahlen, Zeichenketten, etc., wohingegen Records oder Mengen wegen ihrer (internen) Strukturierung nicht zulässige Wertebereiche darstellen.

Eine Relation R ist definiert als eine Teilmenge des kartesischen Produkts (Kreuzprodukts) der n Domänen:

$$R \subseteq D_1 \times \cdots \times D_n$$

Korrekterweise müsste man noch zwischen dem *Schema* einer Relation, das durch die n Domänen gegeben ist, und der aktuellen *Ausprägung* (Instanz) dieses Relationenschemas, die durch die Teilmenge des Kreuzproduktes gegeben ist, unterscheiden. Wir werden aber oft keine klare Unterscheidung zwischen der Metaebene (Schema) und der Instanzebene (Ausprägung) machen; es sollte aus dem Kontext jeweils leicht ersichtlich sein, welche Ebene der Datenbank gemeint ist.

Ein Element der Menge R wird als *Tupel* bezeichnet, dessen *Stelligkeit* (engl. *arity*) sich aus dem Relationenschema ergibt. Im abstrakten Beispiel ist die Stelligkeit n.

3.1.2 Schema-Definition

Der oben angegebene Formalismus stammt aus der Mathematik. Im Datenbankbereich gibt man den einzelnen Komponenten der Tupel noch Namen. Wir wollen dies anschaulich erläutern: Ein einfaches Beispiel für eine Relation ist das *Telefonbuch*, das unter vereinfachenden Annahmen eine Teilmenge des folgenden Kreuzproduktes darstellt:

$$\text{Telefonbuch} \subseteq \text{string} \times \text{string} \times \text{integer}$$

Hierbei repräsentiere der erste *string*-Wert den Namen des Teilnehmers, der zweite die Adresse und der *integer*-Wert die Telefonnummer. In vielen kommerziellen Systemen werden Relationen auch als *Tabellen* (engl. *table*) bezeichnet, weil man sich die Ausprägung einer Relation wie folgt als flache Tabelle veranschaulichen kann:

Telefonbuch		
Name	Straße	Telefon#
Mickey Mouse	Main Street	4711
Minnie Mouse	Broadway	94725
Donald Duck	Highway	95672
...	...	...

Hierbei werden die Spalten als *Attribute* (oder manchmal auch Felder) bezeichnet. Die Attribute müssen innerhalb einer Relation eindeutig benannt sein. Zwei unterschiedliche Relationen dürfen aber durchaus gleiche Attributnamen enthalten. Die Zeilen der Tabelle entsprechen den Tupeln der Relation. In diesem Beispiel enthält die Relation (Tabelle) drei dreistellige Tupel, deren Attributwerte aus den Wertebereichen *string*, *string* und *integer* stammen.

Relationenschemata werden wir nach folgendem Muster spezifizieren:

$$\text{Telefonbuch} : \{[\text{Name} : \text{string}, \text{Adresse} : \text{string}, \underline{\text{Telefon\#} : \text{integer}}]\}$$

Hierbei wird in den inneren eckigen Klammern $[\ldots]$ angegeben, wie die einzelnen Tupeln aufgebaut sind, d.h. welche Attribute vorhanden sind und welchen Typ (Wertebereich) die Attribute haben. Die geschweiften Klammern $\{\ldots\}$ sollen ausdrücken, dass es sich bei einer Relationsausprägung um eine Menge von Tupeln handelt. Hierdurch wird also die Datentyp-Sichtweise betont: die eckigen Klammern $[\ldots]$ repräsentieren den Tupelkonstruktor (analog zur Definition eines Recordtyps) und die geschweiften Klammern $\{\ldots\}$ stellen den Mengenkonstruktor dar. Eine Relationsausprägung ist also als eine Menge von Tupeln $\{[\ldots]\}$ aufzufassen.

Wie wir oben schon angemerkt haben, werden wir in diesem Buch keine „dogmatische" Trennung zwischen Schema- und Ausprägungsebene machen. Die Ausprägung und das Schema werden meistens mit demselben Namen (hier *Telefonbuch*) angesprochen. An einigen Stellen des Buchs ist aber eine präzisere Sicht notwendig. Dann bezeichnen wir mit $\mathbf{sch}(R)$ oder mit $\mathcal{R}$ die Menge der Attribute einer Relation und mit R die aktuelle Ausprägung. Mit $\mathbf{dom}(A)$ bezeichnen wir die Domäne eines Attributs A. Also gilt für das Schema (d.h., die Attributmenge) $\mathcal{R} = \{A_1, \ldots, A_n\}$, dass die Relation R eine Teilmenge des kartesischen Produkts der n Domänen $\mathbf{dom}(A_1)$, $\mathbf{dom}(A_2)$, ..., $\mathbf{dom}(A_n)$ ist, also:

$$R \subseteq \mathbf{dom}(A_1) \times \mathbf{dom}(A_2) \times \cdots \times \mathbf{dom}(A_n)$$

Der Primärschlüssel der Relation wird durch Unterstreichung gekennzeichnet. In dem *Telefonbuch*-Beispiel gehen wir davon aus, dass es pro *Telefon#* nur einen Eintrag geben kann[1].

3.2 Umsetzung eines konzeptuellen Schemas in ein relationales Schema

Das Entity-Relationship Modell besitzt zwei grundlegende Strukturierungskonzepte:

1. Entitytypen und
2. Beziehungstypen.

Dem steht im relationalen Modell nur ein einziges Strukturierungskonzept – nämlich die Relation – gegenüber. Es werden also sowohl Entitytypen als auch Beziehungstypen jeweils auf eine Relation abgebildet.

3.2.1 Relationale Darstellung von Entitytypen

Wir wollen uns zunächst die vier Entitytypen unseres Universitätsschemas anschauen (Das ER-Schema ist in Abbildung 3.1 nochmals dargestellt):

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

Wir haben an dieser Stelle vorerst bewusst auf die Modellierung der Generalisierung von *Assistenten* und *Professoren* zu *Angestellten* im relationalen Schema verzichtet. Die Generalisierung wird nämlich im relationalen Modell nicht explizit unterstützt; sie kann aber durch Ausnutzung der verfügbaren Strukturen „imitiert" werden – mehr dazu in Abschnitt 3.3.4.

3.2.2 Relationale Darstellung von Beziehungen

Wir müssen uns nun Gedanken zur Umsetzung der Beziehungstypen in ein relationales Schema machen. Im *Initial*-Entwurf wird für jeden Beziehungstyp eine eigene Relation definiert – einige dieser Relationen können später in der Schemaverfeinerung (siehe Abschnitt 3.3) mit anderen Relationen zusammengefasst und somit wieder eliminiert werden.

[1] Man beachte, dass dies in der Realität nicht der Fall ist, da man (in Deutschland) zu einem Telefonanschluss zusätzliche Einträge ins Telefonbuch einfügen lassen kann.

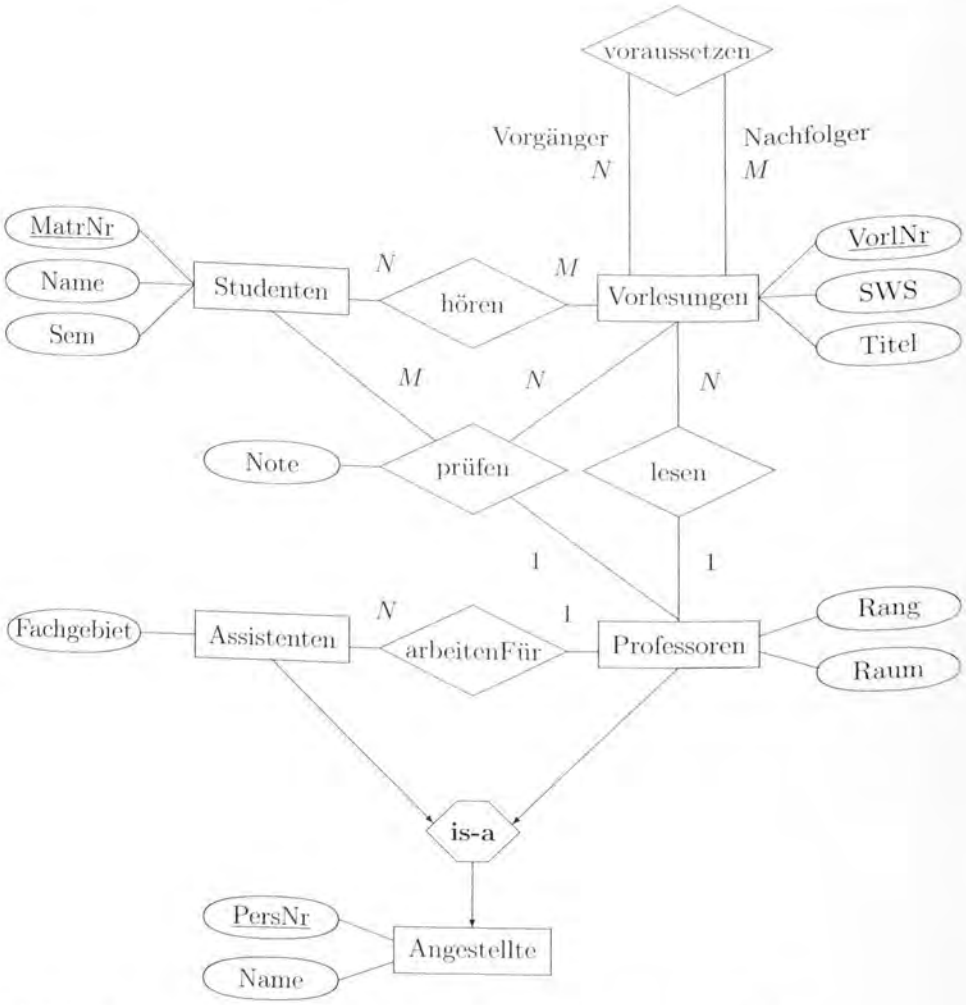


Abbildung 3.1: Konzeptuelles Schema der Universität (wiederholt)

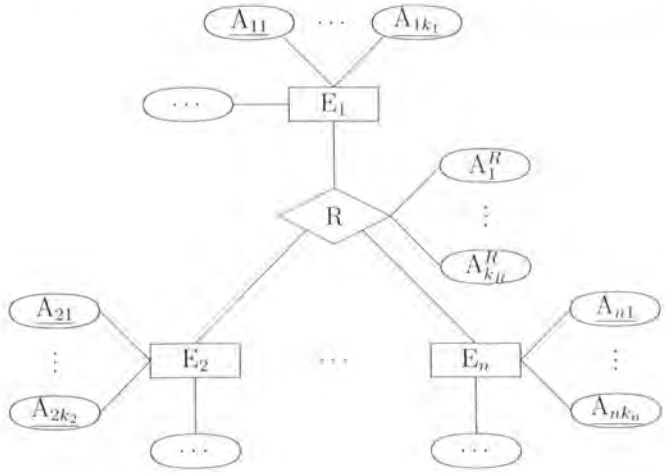


Abbildung 3.2: Beispiel einer allgemeinen n-stelligen Beziehung

Wir wollen zunächst das Grundprinzip der Umsetzung von Beziehungstypen vorstellen. Dazu betrachten wir die abstrakte n-stellige Beziehung R in Abbildung 3.2. Dieser Beziehungstyp stellt eine Zuordnung zwischen n Entitytypen her. Das relationale Pendant hat folgende Form:

$$R : \{[\underbrace{A_{11}, \ldots, A_{1k_1}}_{\text{Schlüssel von } E_1}, \underbrace{A_{21}, \ldots, A_{2k_2}}_{\text{Schlüssel von } E_2}, \ldots, \underbrace{A_{n1}, \ldots, A_{nk_n}}_{\text{Schlüssel von } E_n}, \underbrace{A^R_1, \ldots, A^R_{k_R}}_{\text{Attribute von } R}]\}$$

Die zugehörige Relation R enthält also alle Schlüsselattribute der Entitytypen $E_1, \ldots, E_n$ und zusätzlich die der Beziehung zugeordneten Attribute $A^R_1, \ldots, A^R_{k_R}$. Diese Schlüsselattribute der Entitytypen nennt man *Fremdschlüssel*, da sie dazu dienen, Tupel (bzw. Entities) aus anderen Relationen (bzw. Entitytypen) zu identifizieren, um dadurch die Zuordnung innerhalb der Beziehung R zu modellieren.

Es kann notwendig sein, dass man in der relationalen Modellierung einige der aus den Entitytypen übernommenen Attributnamen umbenennen muss. Dies kann zum einen zwingend notwendig sein, wenn Attribute in unterschiedlichen Entitytypen gleichbenannt sind. Zum anderen kann eine Umbenennung sinnvoll sein, um die Bedeutung der Attribute als Fremdschlüssel zu betonen.

Für unser Universitätsschema ergibt sich – gemäß der oben am abstrakten Beispiel dargestellten Vorgehensweise – folgende Modellierung der Beziehungstypen:

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

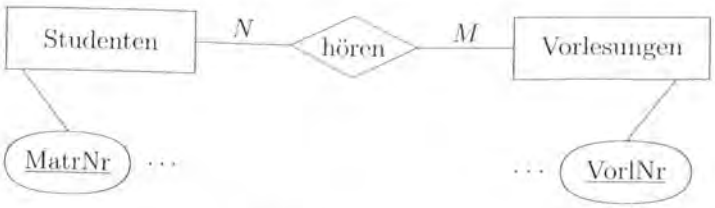


Abbildung 3.3: Beispiel einer N:M-Beziehung: *hören*

In obigem Relationenschema sind die *Schlüssel* wiederum durch Unterstreichung gekennzeichnet. Wir wollen an dieser Stelle nur intuitiv den Schlüsselbegriff, der schon für das ER-Modell eingeführt worden war, erläutern: Ein Schlüssel einer Relation stellt eine *minimale* Menge von Attributen dar, deren Werte die Tupel innerhalb der Relation eindeutig identifizieren. Mit anderen Worten, es können nicht mehrere Tupel mit gleichen Werten für alle zu einem Schlüssel gehörenden Attribute existieren. Wenn es mehrere Schlüssel (-Kandidaten) gibt, wird meist ein sogenannter *Primärschlüssel* ausgewählt. Eine detailliertere und formalere Behandlung des Schlüsselbegriffs wird in Kapitel 6 gegeben.

Wenden wir uns nun nochmals der Beziehung *hören* zu, die in Abbildung 3.3 isoliert dargestellt ist. Die zugehörige Relation *hören* hat den Schlüssel {*MatrNr*, *VorlNr*}, da Studenten i.A. mehrere Vorlesungen belegen und umgekehrt Vorlesungen i.A. von mehreren Studenten besucht werden. Generell gilt bei der Umsetzung von N:M-Beziehungen, dass die Menge *aller* Fremdschlüsselattribute den Schlüssel der Relation bildet. Wir wollen dies an einer Beispielausprägung der Relation *hören* intuitiv demonstrieren:

Studenten	
MatrNr	...
26120	...
27550	...
...	...

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

Vorlesungen	
VorlNr	...
5001	...
4052	...
...	...

Es ist ersichtlich, dass die Werte des Attributs *MatrNr* aus *hören* als Fremdschlüssel auf Tupel der Relation *Studenten* verweisen. Analog verweisen die Werte des Attributs *VorlNr* aus *hören* auf Tupel der Relation *Vorlesungen*. Zu einer gegebenen *MatrNr* – z.B. 27550 – gibt es mehrere Einträge in der Relation *hören*.

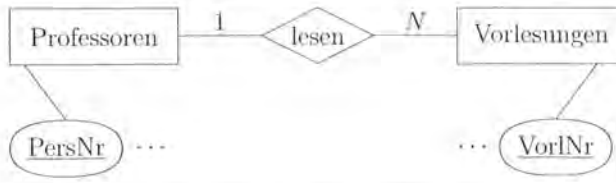


Abbildung 3.4: Beispiel einer 1:N-Beziehung: *lesen*

Gleichfalls gibt es zu einer gegebenen *VorlNr* mehrere Einträge.

Anders verhält es sich bei der Umsetzung von 1:N-Beziehungen. Ein Beispiel dafür ist die Relation (bzw. der Beziehungstyp) *lesen*, wodurch Professoren mit den von ihnen gehaltenen Vorlesungen assoziiert werden (siehe Abbildung 3.4).

Da eine Vorlesung von nur einem Professor bzw. einer Professorin gehalten wird, hat die zugehörige Relation *lesen* den Schlüssel $\{VorlNr\}$. Dies ergibt sich auch aus der funktionalen Sichtweise der Beziehung *lesen*, die man – wie in Kapitel 2 beschrieben – als eine (partielle) Funktion der folgenden Form auffassen kann:

$$\text{lesen} : \text{Vorlesungen} \rightarrow \text{Professoren}$$

Es sollte ausdrücklich betont werden, dass die Funktion bei einer 1:N-Beziehung nicht in der anderen „Richtung" gegeben ist. Also ist *lesen* keine Funktion von *Professoren* nach *Vorlesungen*, da *Professoren* i.A. mehrere *Vorlesungen* halten.

Die beiden restlichen binären Beziehungstypen unseres ER-Schemas werden wie folgt modelliert:

- *arbeitenFür* ist eine 1:N-Beziehung zwischen *Professoren* und *Assistenten* und kann als Funktion von *Assistenten* nach *Professoren* gesehen werden. Demzufolge hat die Relation *arbeitenFür* den Schlüssel $\{AssiPersNr\}$.

- *voraussetzen* ist eine rekursive N:M-Beziehung. Bei der relationalen Modellierung wurden die Rollen des ER-Schemas – nämlich *Vorgänger* und *Nachfolger* – als Attributnamen gewählt. Da es sich hierbei um eine N:M-Beziehung handelt, wird der Schlüssel der Relation *voraussetzen* von beiden Attributen $\{Vorgänger, Nachfolger\}$ gebildet.

Bei der Beziehung *prüfen* handelt es sich um eine ternäre Beziehung. Gemäß unserer allgemeinen Vorgehensweise bei der Umsetzung von Beziehungstypen in Relationen übernehmen wir den Schlüssel *MatrNr* aus *Studenten*, den Schlüssel *VorlNr* aus *Vorlesungen* und den Schlüssel *PersNr* aus *Professoren*. Zusätzlich hat die Beziehung *prüfen* noch das Attribut *Note*. Der Schlüssel der Relation *prüfen* ergibt sich aus der Funktionalitätsangabe N:M:1 im ER-Schema, die aussagt, dass *prüfen* den Eigenschaften einer partiellen Funktion

$$\text{prüfen} : \text{Studenten} \times \text{Vorlesungen} \rightarrow \text{Professoren}$$

genügen muss. Mit anderen Worten, darf es zu einem Studenten/Vorlesungs-Paar höchstens einen Professor bzw. eine Professorin geben. Daraus folgt, dass *MatrNr* **und** *VorlNr* den Schlüssel der Relation *prüfen* darstellen.

Wir überlassen es den Lesern in Übungsaufgabe 3.2, die in Abschnitt 2.7.2 eingeführte dreistellige Beziehung *betreuen* relational umzusetzen.

3.3 Verfeinerung des relationalen Schemas

Das im Initialentwurf erzeugte relationale Schema lässt sich oftmals noch verfeinern. Dabei werden einige der Relationen eliminiert, die für die Modellierung von Beziehungstypen eingeführt worden waren. Dies ist aber *nur* für solche Relationen möglich, die 1:1-, 1:N- oder N:1-Beziehungen repräsentieren. Die Elimination der Relationen, die die allgemeinen N:M-Beziehungstypen repräsentieren, ist nicht sinnvoll und würde i.A. zu schwerwiegenden „Anomalien" führen.

Bei der Eliminierung von Relationen gilt es folgende Regel zu beachten:

Nur Relationen mit gleichem Schlüssel zusammenfassen!

3.3.1 1:N-Beziehungen

Wir betrachten dazu nochmals den Beziehungstyp *lesen*, der in Abbildung 3.4 dargestellt ist. Im Initialentwurf gab es drei Relationen:

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

Gemäß der oben gegebenen Regel kann man die Relationen *Vorlesungen* und *lesen* zusammenfassen, so dass für diesen Ausschnitt zwei relevante Relationen im Schema verbleiben:

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

Hierbei stellt das Attribut *gelesenVon* einen Fremdschlüssel auf die Relation *Professoren* dar, d.h. Werte von *gelesenVon* entsprechen den *PersNr*-Werten von *Professoren*. Da jede Vorlesung nur von einer Person gelesen wird, gibt es für jedes Tupel in *Vorlesungen* nur ein einziges zugeordnetes Tupel aus *Professoren*, das über das Attribut *gelesenVon* referenziert wird. Die Relationen *Vorlesungen* und *Professoren* haben dann z.B. folgende Ausprägung:

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

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

Hierbei verweist z.B. der Wert 2137 des Attributs *gelesenVon* im ersten Tupel von *Vorlesungen* auf das letztgezeigte Tupel namens „Kant" in *Professoren*.

Es kann gar nicht eindringlich genug davor gewarnt werden, Relationen mit unterschiedlichen Schlüsseln zusammenzufassen. Ein häufig vorkommender Fehler besteht z.B. für die hier betrachtete Beziehung *lesen* darin, die Information in die Relation *Professoren* zu integrieren. Dies könnte man **fälschlicherweise** wie folgt versuchen:

$$\text{Professoren}' \; : \; \{[\text{PersNr}, \underline{\text{liestVorl}}, \text{Name}, \text{Rang}, \text{Raum}]\}$$

Hierdurch ändert sich natürlich der Schlüssel von *Professoren*, der jetzt aus dem Attribut $\{liestVorl\}$ besteht – anstatt $\{PersNr\}$. Dies führt zu einer Redundanz von Teilen der gespeicherten Information, wie die folgende Beispielausprägung zeigt:

Professoren′				
PersNr	liestVorl	Name	Rang	Raum
2125	5041	Sokrates	C4	226
2125	5049	Sokrates	C4	226
2125	4052	Sokrates	C4	226
2126	5043	Russel	C4	232
2126	5052	Russel	C4	232
2126	5216	Russel	C4	232
...	...	...	...	...

Bei dieser Modellierung werden z.B. der *Name*, der *Rang* und der *Raum* von Sokrates und Russel dreimal abgespeichert. Das hat zum einen einen höheren Speicherbedarf zur Folge, zum anderen führt die Redundanz zum schwerwiegenderen Problem der sogenannten Update-Anomalien.[2] Wenn z.B. Russel vom Raum 232 in den Raum 278 umzieht, muss man dies in allen drei gespeicherten Tupeln ändern um eine konsistente Datenbasis zu gewährleisten.

Auf analoge Weise wird die Beziehung *arbeitenFür* behandelt. Wegen der 1:N-Funktionalität kann man die Beziehung in der Relation *Assistenten* modellieren, so dass diese Relation folgendes Schema hat:

$$\text{Assistenten} : \{[\underline{\text{PersNr}}, \text{Name}, \text{Fachgebiet}, \text{Boss}]\}$$

[2] Dies wird systematisch in Kapitel 6 behandelt.

Abbildung 3.5: Beispiel einer 1:1-Beziehung

3.3.2 1:1-Beziehungen

Bei der relationalen Modellierung von 1:1-Beziehungen hat man mehr Freiraum als bei 1:N-Beziehungen. Betrachten wir dazu die Beziehung *Dienstzimmer* aus Abbildung 3.5. Hier wird die Zuordnung zwischen *Professoren* und den *Räumen*, die sie als *Dienstzimmer* verwenden, explizit als Beziehungstyp dargestellt. In unserem konzeptuellen Schema aus Abbildung 3.1 hatten wir dies vereinfacht als ein Attribut *Raum* modelliert.

Im Initialentwurf könnte man diesen Ausschnitt wie folgt repräsentieren:

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

In der obigen Schemadefinition ist *PersNr* als Primärschlüssel von *Dienstzimmer* markiert. Wir hätten aber genauso gut die *RaumNr* als Primärschlüssel auswählen können, da auch *RaumNr* wegen der 1:1-Funktionalität der Beziehung einen Kandidatenschlüssel bildet.

Da also *Professoren* und *Dienstzimmer* denselben Schlüssel haben, kann man sie nach der oben angegebenen Regel zusammenfassen. Nach Zusammenfassung der Relationen *Professoren* und *Dienstzimmer* erhält man folgendes Schema:

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

Die Relation *Professoren* entspricht jetzt wieder unserem Originalentwurf, wobei das Attribut *Raum* eigentlich die Nummer des betreffenden Raums repräsentiert.

Wir hätten aber ebensogut auch die Relationen *Dienstzimmer* und *Räume* zusammenfassen können, da, wie gesagt, auch *RaumNr* einen Schlüssel von *Dienstzimmer* bildet:

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

Hierbei verweist das Attribut *ProfPersNr* auf Tupel der Relation *Professoren*. Diese Modellierung hat allerdings den Nachteil, dass viele Tupel einen sogenannten Null-Wert für das Attribut *ProfPersNr* haben, da viele Räume nicht als Dienstzimmer von Professoren genutzt werden. Ein *NULL*-Eintrag repräsentiert den Wert „unbekannt" oder „nicht anwendbar". Wegen der Problematik im Umgang mit Null-Werten ist die erste Modellierung der Beziehung *Dienstzimmer* vorzuziehen.

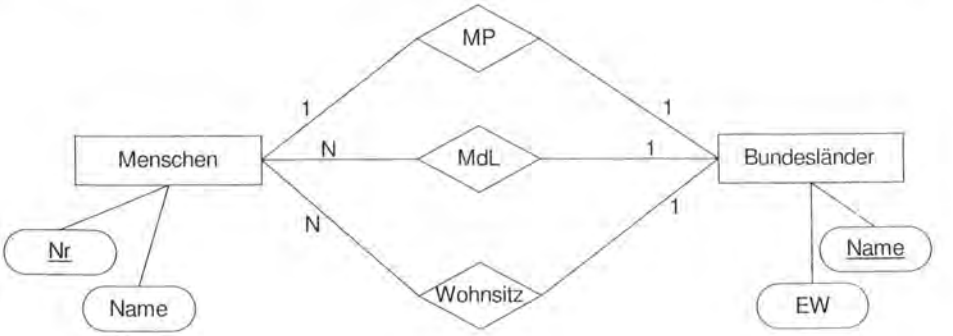


Abbildung 3.6: Beziehungen zwischen *Menschen* und *Bundesländern*

3.3.3 Vermeidung von Null-Werten

Bei der Repräsentation von $1:1$- oder $1:N$-Beziehungen muss man auch beachten, dass einige oder sogar viele der betreffenden Entities die Beziehung gar nicht eingehen. Wenn man die Beziehung als Fremdschlüssel in eine der Relationen, die die an der Beziehung beteiligten Entitytypen modellieren, hineinzieht, wäre der Fremdschlüssel in diesen Tupeln undefiniert. Der Fremdschlüssel enthält dann *Null*-Werte, was man möglichst vermeiden sollte. Wir wollen dies an einem Beispiel illustrieren, das in Abbildung 3.6 als Entity/Relationship-Schema gegeben ist. *Menschen* können Ministerpräsident/in (*MP*) *eines* Bundeslandes sein, sie können Mitglied des Landtags (*MdL*) *eines* Bundeslandes sein und sie haben ihren (Erst-)*Wohnsitz* in *einem* Bundesland. *Bundesländer* haben *einen* Menschen als Ministerpräsident/in.

Nach den oben schon vorgestellten Regeln darf man eine binäre Beziehung auch in der Relation repräsentieren, die den Entitytyp modelliert, der "gegenüber" von einer **1** liegt. Wir überlassen es den Lesern nachzuweisen, dass man dadurch von unserer Regel der Zusammenlegung von Relationen mit gleichem Schlüssel nicht abweicht. In unserem Beispiel dürfte man also alle drei Beziehungen *MP*, *MdL* und *Wohnsitz* in der Relation *Menschen* repräsentieren, so dass wir folgendes Schema erhalten würden:

Menschen				
Nr	Name	Wohnsitz	MPvon	MdLvon
4711	Kemper	Bayern	–	–
4813	Seehofer	Bayern	Bayern	Bayern
5833	Maget	Bayern	–	Bayern
6745	Woidke	Brandenburg	Brandenburg	Brandenburg
8978	Schröder	Niedersachsen	–	–
...	...	...	...	...

Diese Modellierung hat den klaren Nachteil, dass viele Tupel Null-Werte enthalten. Von den ca. 80 Mio Einwohnern Deutschlands sind nur sehr wenige (im Bereich von ein paar Tausend) Mitglied eines Landtags und noch viel weniger sind Ministerpräsident/in eines Bundeslandes. Somit enthalten die Fremdschlüssel-Attribute

MdLvon und *MPvon* bei fast allen Tupeln Null-Werte. Anders verhält es sich bei dem Fremdschlüssel *Wohnsitz*: Wenn wir nur in Deutschland lebende Menschen speichern, ist dieser Fremdschlüssel bei allen Menschen (mit festem Wohnsitz) definiert.

Aus diesen Überlegungen folgt:

- Der *Wohnsitz* kann als Fremdschlüssel in der Entity-Relation *Menschen* bleiben.
- Die Beziehung *MP* modelliert man am besten als Fremdschlüssel in *Bundesländer*, da alle Bundesländer *einen* MP haben.
- Die Beziehung *MdL* repräsentiert man als eigenständige Relation mit den Fremdschlüsseln *Nr* auf Menschen und *Bundesland* auf *Bundesländer*.

Nachfolgend sind die revidierten Relationenschemata mit den resultierenden Beispiel-Tupeln gezeigt:

Menschen		
Nr	Name	Wohnsitz
4711	Kemper	Bayern
4813	Seehofer	Bayern
5833	Maget	Bayern
6745	Woidke	Brandenburg
8978	Schröder	Niedersachsen
...	...	...

MdL	
Nr	Bundesland
4813	Bayern
5833	Bayern
6745	Brandenburg
...	...

Bundesländer		
Name	EW	MP
Bayern	12443893	4813
Brandenburg	2562946	6745
...	...	...

3.3.4 Relationale Modellierung der Generalisierung

Schauen wir uns nochmals die in Abbildung 3.7 isoliert dargestellte Generalisierung von *Assistenten* und *Professoren* zu *Angestellte* an. Eine sehr einfache relationale Repräsentation wäre die folgende:

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

Hierbei würde die Information zu einem Professor bzw. einer Professorin, wie z.B.

[2136, Curie, C4, 36]

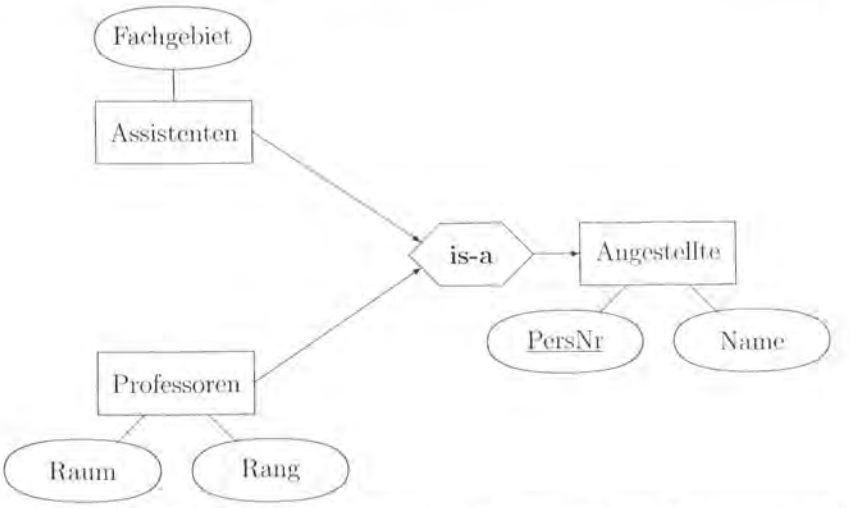


Abbildung 3.7: Isolierte Darstellung der Generalisierung von *Professoren* und *Assistenten* zu *Angestellte*

auf zwei Tupel aufgeteilt, nämlich

$$[2136, \text{Curie}] \quad \text{und} \quad [2136, \text{C4}, 36]$$

Das erste Tupel ist in der Relation *Angestellte*, das zweite in der Relation *Professoren* zu finden. Um die vollständige Information zu Curie zu erhalten, muss man die beiden Tupel verbinden (joinen).

Diese Darstellung hat also den Nachteil, dass in den Relationen, die Spezialisierungen repräsentieren, nicht die volle Information verfügbar ist. Mit anderen Worten, die *Vererbung* ist nicht realisiert.

Leider verfügt das relationale Modell über keine Vererbungskonstrukte. Wir werden lediglich in Abschnitt 4.19 diskutieren, inwieweit das Sichtenkonzept relationaler DBMS die Modellierung von Generalisierung/Spezialisierung und der damit verbundenen Vererbung unterstützen kann.

3.3.5 Beispielausprägung der Universitäts-Datenbank

In Abbildung 3.8 ist eine Beispielausprägung der Universitäts-Datenbasis gezeigt. Das zugrundeliegende Schema enthält folgende Relationen:

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

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

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

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

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

hören	
MatrNr	VorlNr
26120	5001
27550	5001
27550	4052
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 3.8: Beispielausprägung unserer Universitäts-Datenbank

3.3.6 Relationale Modellierung schwacher Entitytypen

Obwohl unser Universitätsschema keine schwachen Entitytypen enthält, wollen wir doch nicht versäumen, deren relationale Repräsentation zu diskutieren. Dazu betrachten wir das Beispielschema aus Abbildung 2.12 (auf Seite 51), in dem *Prüfungen* als existenzabhängiger, schwacher Entitytyp dem „starken" Entitytyp *Studenten* untergeordnet wurde. Dieser schwache Entitytyp lässt sich relational wie folgt repräsentieren:

$$\text{Prüfungen} : \{[\underline{\text{MatrNr} : \text{integer}, \text{PrüfTeil} : \text{string}}, \text{Note} : \text{decimal}]\}$$

Die Relation *Prüfungen* hat also einen zusammengesetzten Schlüssel bestehend aus der *MatrNr* aus *Studenten* und der *PrüfTeil*-Kennung, die alle Prüfungen eines Studenten bzw. einer Studentin eindeutig identifiziert.

Die dem Entitytyp *Prüfungen* zugeordneten binären Beziehungen *umfassen* und *abhalten* lassen sich jetzt wie gehabt repräsentieren:

$$\begin{aligned} \text{umfassen} &: \{[\underline{\text{MatrNr} : \text{integer}, \text{PrüfTeil} : \text{string}, \text{VorlNr} : \text{integer}}]\} \\ \text{abhalten} &: \{[\underline{\text{MatrNr} : \text{integer}, \text{PrüfTeil} : \text{string}, \text{PersNr} : \text{integer}}]\} \end{aligned}$$

Man beachte, dass in diesem Fall der (global eindeutige) Schlüssel der Relation *Prüfungen*, nämlich *MatrNr* **und** *PrüfTeil* als Fremdschlüssel in die Relationen *umfassen* und *abhalten* übernommen werden muss. Da es sich bei diesen beiden Beziehungen um allgemeine N:M-Beziehungen handelt, bilden alle Fremdschlüssel zusammen den Schlüssel – und es ist deshalb auch keine Zusammenfassung einer dieser Relationen mit der Relation *Prüfungen* möglich. Die Leser mögen bitte diskutieren, inwieweit sich eine Einschränkung auf einen Prüfer pro Prüfung auf dieses Schema auswirken würde.

3.4 Die relationale Algebra

Natürlich benötigt man neben der Strukturbeschreibung (d.h. dem Datenbankschema) auch eine Sprache, mit der man Informationen aus der Datenbank extrahieren kann.[3] Es gibt zwei formale Sprachen, die für die Anfrageformulierung in relationalen Datenbanken konzipiert wurden. Es handelt sich hierbei um

1. die relationale Algebra (oder Relationenalgebra) und
2. den Relationenkalkül.

Der Relationenkalkül ist eine rein deklarative Sprache, in der spezifiziert wird, *welche* Daten man erhalten will bzw. welche Kriterien diese Daten erfüllen müssen, aber nicht *wie* die Anfrage ausgewertet werden kann. Die erstgenannte Sprache, die

[3] Zusätzlich benötigt man selbstverständlich auch noch eine Datenmanipulationssprache (DML) zum Einfügen, Verändern und Löschen von Informationen. Die DML wird erst in Kapitel 4 eingeführt.

relationale Algebra, ist stärker *prozedural* orientiert, d.h. ein relationenalgebraischer Ausdruck beinhaltet implizit einen Abarbeitungsplan, wie die Anfrage auszuwerten ist. Deshalb spielt die Relationenalgebra eine größere Rolle bei der Realisierung von Datenbanksystemen – und insbesondere bei der Anfrageoptimierung (siehe Kapitel 8). Beide Sprachen sind *abgeschlossen*, d.h. die Ergebnisse der Anfragen sind wiederum Relationen.

Wir werden zunächst die Operatoren der Relationenalgebra behandeln und anschließend in Abschnitt 3.5 die zwei Ausprägungsformen des Relationenkalküls beschreiben.

3.4.1 Selektion

Bei der *Selektion* werden diejenigen Tupel einer Relation ausgewählt, die das sogenannte *Selektionsprädikat* erfüllen. Die Selektion wird mit σ bezeichnet und hat das Selektionsprädikat als Subskript. Ein Beispiel für die Selektion wäre folgende Anfrage:

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

In dieser Anfrage werden die „Dauerstudenten" aus der Datenbank extrahiert. Das Ergebnis wäre dann für unsere Beispielausprägung wie folgt:

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

Man kann sich die Auswertung der Selektion so vorstellen, dass jedes Tupel der Argumentrelation (hier *Studenten*) einzeln inspiziert wird um das Prädikat (hier „*Semester* > 10") auszuwerten. Falls das Prädikat erfüllt ist, wird das Tupel in die Ergebnisrelation kopiert.

Allgemein ist das Selektionsprädikat eine Formel F, die aufgebaut ist aus:

1. Attributnamen der Argumentrelation R oder Konstanten als Operanden,
2. den arithmetischen Vergleichsoperatoren $=, <, \leq, >, \geq, \neq$ und
3. den logischen Operatoren $\wedge$ (und), $\vee$ (oder) und $\neg$ (nicht).

Dann besteht das Ergebnis der Selektion

$$\sigma_F(R)$$

aus allen Tupeln $t \in R$, für die die Formel F erfüllt ist, wenn jedes Vorkommen eines Attributnamens A in F durch den Wert $t.A$ ersetzt wird.

3.4.2 Projektion

Während bei der Selektion einzelne Zeilen (Tupel) einer Tabelle (Relation) ausgewählt werden, werden bei der Projektion Spalten (Attribute) der Argumentrelation extrahiert. Die Projektion wird mit dem Operatorsymbol Π bezeichnet und enthält die Menge der Attributnamen im Subskript. Als Beispiel betrachten wir folgenden relationenalgebraischen Ausdruck:

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

An diesem Beispiel ist schon ersichtlich, dass sich eine „saloppe" Terminologie eingebürgert hat: Die Attributmenge, auf die projiziert werden soll, wird oft nicht als Menge, sondern einfach als durch Kommata getrennte Sequenz angegeben – man lässt also die Mengenklammern weg. In der obigen Anfrage werden die in der aktuellen Ausprägung vorkommenden Werte des *Rang*-Attributs aus der Argumentrelation *Professoren* gesucht. Das Ergebnis der Anfrage sieht dann wie folgt aus:

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

Es ist zu beachten, dass Duplikattupel, die durch die Beschränkung auf eine Teilmenge der Attribute der Argumentrelation auftreten können, vor der Ergebnisausgabe eliminiert werden müssen.

Die Duplikateliminierung kann entfallen, wenn in der Projektion ein (vollständiger) Schlüssel der Relation enthalten ist. In diesem Fall können wegen der Schlüsseleigenschaft keine zwei Tupel vollständig gleiche Attributwerte besitzen, und somit wäre die Duplikateliminierung wirkungslos (aber nicht kostenlos – in Bezug auf die Laufzeit eines realen Systems).

3.4.3 Vereinigung

Zwei Relationen mit gleichem Schema – d.h. mit gleichen Attributnamen und Attributtypen (Domänen) – kann man durch die Vereinigung zu einer Relation zusammenfassen. Als Beispiel betrachten wir folgenden Ausdruck:

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

In dieser Anfrage werden zunächst die beiden Relationen *Assistenten* und *Professoren* durch die jeweiligen Projektionen in das gleiche Schema „gezwängt". Danach kann die Vereinigung auf der Basis dieser beiden schemagleichen, temporären Argumentrelationen, nämlich $\Pi_{PersNr,\ Name}(\text{Assistenten})$ und $\Pi_{PersNr,\ Name}(\text{Professoren})$, durchgeführt werden. Als Ergebnis erhalten wir in diesem Beispiel eine Relation mit insgesamt 13 Tupeln. Auch bei der Vereinigungsoperation muss eine Duplikatelimination durchgeführt werden, da durch das Zusammenbringen der Tupel aus zwei Argumentrelationen u.U. Duplikate auftreten können.[4]

[4] Bei der Vereinigung muss in jedem Fall die Duplikateliminierung durchgeführt werden – auch wenn von beiden Relationen der Schlüssel übernommen wird. Warum?

3.4.4 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.

Als Beispiel können wir die Menge der Studenten – genauer gesagt, deren *MatrNr* – ermitteln, die noch keine Prüfung abgelegt haben:

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

3.4.5 Kartesisches Produkt (Kreuzprodukt)

Das Kreuzprodukt zweier Relationen R und S wird als

$$R \times S$$

gebildet und enthält alle $|R| * |S|$ möglichen Paare von Tupeln aus R und S. Das Schema der Ergebnisrelation, also $\mathbf{sch}(R \times S)$, ist die Vereinigung der Attribute aus $\mathbf{sch}(R)$ und $\mathbf{sch}(S)$:

$$\mathbf{sch}(R \times S) = \mathbf{sch}(R) \cup \mathbf{sch}(S) = \mathcal{R} \cup \mathcal{S}$$

Beim kartesischen Produkt kann es natürlich vorkommen, dass die beiden Argumentrelationen zwei (oder mehr) gleich benannte Attribute enthalten. In diesem Fall wird eine eindeutige Benennung dadurch erzwungen, dass dem Attributnamen der Relationenname, gefolgt von einem Punkt, vorangestellt wird. Das Attribut A aus R wird beispielsweise mit $R.A$ und das gleichnamige Attribut A aus S mit $S.A$ bezeichnet. Diese um den Relationennamen erweiterten Attributbezeichner nennt man auch *qualifizierte* Attributnamen.

Als Beispiel einer Kreuzprodukt-Operation betrachten wir *Professoren* × *hören*. Die Ergebnisrelation hat folgende Gestalt:

Professoren × hören					
Professoren				hören	
PersNr	Name	Rang	Raum	MatrNr	VorlNr
2125	Sokrates	C4	226	26120	5001
...	...	...	...	...	...
2125	Sokrates	C4	226	29555	5001
...	...	...	...	...	...
2137	Kant	C4	7	29555	5001

Diese Relation ist also sechsstellig und enthält insgesamt 91 (= 7 * 13) Tupel – bezogen auf unsere Beispielausprägungen von *Professoren* (mit 7 Tupeln) und *hören* (mit 13 Tupeln). In diesem Beispiel ist die Qualifizierung der Attribute nicht notwendig, da die Attributnamen paarweise unterschiedlich sind.

3.4.6 Umbenennung von Relationen und Attributen

Manchmal ist es notwendig, dieselbe Relation mehrfach in einer Anfrage zu verwenden. Dazu wird dann – zumindest logisch – eine vollständige zusätzliche Kopie der Relation generiert. In diesem Fall muss zumindest eine der Relationen umbenannt werden. Dafür wird der Operator ρ verwendet, wobei im Subskript der neue Name der Relation angegeben wird. Z.B. kann man die Relation *voraussetzen* wie folgt in *V1* umbenennen:

$$\rho_{V1}(\text{voraussetzen})$$

Betrachten wir nun ein Beispiel, in dem die Umbenennung notwendig ist: Wir wollen die indirekten Vorgänger 2. Stufe (also die Vorgänger der Vorgänger) der Vorlesung mit Nummer 5216 herausfinden. Dies lässt sich mit folgendem relationenalgebraischen Ausdruck erreichen:

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

Diese Anfrage wird abgearbeitet, indem zunächst das kartesische Produkt der beiden Relationen *V1* und *V2* gebildet wird, die jeweils Kopien der (gespeicherten) Relation *voraussetzen* darstellen:

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

Unter den Tupeln des Kreuzprodukts werden diejenigen ausgewählt (selektiert), für die die beiden nachfolgenden Bedingungen gelten:

1. V2.Nachfolger = 5216
2. V1.Nachfolger = V2.Vorgänger

Aus diesen selektierten Tupeln wird dann das Attribut *V1.Vorgänger* projiziert. Für unsere Beispiel-Datenbank aus Abbildung 3.8 besteht das Ergebnis dieser Anfrage aus dem einen einstelligen Tupel [5001]. Die Herleitung des Ergebnisses ist oben gezeigt: Die Vorlesung 5001 ist der Vorgänger von Vorlesung 5041, die wiederum Vorgänger der Vorlesung 5216 ist.

Der ρ-Operator wird auch zur Umbenennung von Attributen einer Relation verwendet. Als Beispiel betrachten wir die Umbenennung des Attributs *Vorgänger* der Relation *voraussetzen*:

$$\rho_{\text{Voraussetzung} \leftarrow \text{Vorgänger}}(\text{voraussetzen})$$

Hierdurch wird das Attribut in *Voraussetzung* umbenannt. Da eine Relation i.A. mehrere Attribute besitzt, müssen bei der Attributumbenennung der Originalname (auf der rechten Seite vom Pfeil) und der neue Name (links vom Pfeil) angegeben werden. Wir werden später in Abschnitt 3.4.8 die Notwendigkeit der Attributumbenennung im Zusammenhang mit dem Joinoperator sehen.

3.4.7 Definition der relationalen Algebra

Die bislang eingeführten Operatoren sind ausreichend, um die relationale Algebra formal definieren zu können. Die Basisausdrücke der relationalen Algebra sind entweder

- Relationen der Datenbank oder
- konstante Relationen.

Ein allgemeiner Relationenalgebra-Ausdruck wird aus „kleineren" Algebraausdrücken konstruiert. Seien E_1 und E_2 relationale Algebraausdrücke – also z.B. Basisausdrücke oder auch selbst schon komplexere zusammengesetzte Ausdrücke. Dann sind die folgenden Ausdrücke auch gültige Algebraausdrücke:

- $E_1 \cup E_2$, wobei E_1 und E_2 das gleiche Schema haben müssen – also $\mathbf{sch}(E_1) = \mathbf{sch}(E_2)$.
- $E_1 - E_2$, wiederum Schemagleichheit vorausgesetzt.
- $E_1 \times E_2$.
- $\sigma_P(E_1)$, mit einem Prädikat P über den Attributen in E_1.
- $\Pi_S(E_1)$, mit einer Attributliste S, deren Attribute in dem Schema von E_1 vorkommen.
- $\rho_V(E_1)$ und $\rho_{A \leftarrow B}(E_1)$, wobei B ein Attributname der Relation E_1 ist, und A nicht als Attributname in E_1 vorkommt.

Im Folgenden werden einige weitere Operatoren der Relationenalgebra eingeführt. Streng genommen handelt es sich bei diesen zusätzlichen Operatoren um sogenannten „syntaktischen Zucker", da sie die Ausdrucksfähigkeit nicht erhöhen. Sie können alle durch Kombinationen der bereits eingeführten Operatoren ausgedrückt werden.

3.4.8 Der relationale Verbund (Join)

Wir haben bereits das Kreuzprodukt (kartesisches Produkt) als einen relationalen Operator zur Verknüpfung von zwei (oder mehreren) Relationen eingeführt. Der Nachteil des Kreuzproduktes besteht in der immensen „Aufblähung" der zu verarbeitenden Tupelmenge, da das Ergebnis des Kreuzprodukts $n * m$ Tupel enthält, wenn die beiden Argumentrelationen n bzw. m Tupel enthalten. Im Allgemeinen sind die meisten der durch das Kreuzprodukt entstehenden Tupel auch irrelevant – vgl. dazu die Beispielanfrage in Abschnitt 3.4.5. Deshalb wird beim Verbundoperator (Join) gleich eine Filterung (bzw. Vorauswahl) der verknüpften Tupel vorgenommen.

Der natürliche Verbund

Der sogenannte *natürliche* Verbund zweier Argumentrelationen 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 $n + k$ Attribute $B_1, \ldots, B_k, C_1, \ldots, C_n$ hat, dann hat $R \bowtie S$ die Stelligkeit $m + n + k$. Hierbei wird angenommen, dass die Attribute A_i und C_j für $(1 \leq i \leq m, 1 \leq j \leq n)$ jeweils paarweise unterschiedlich sind – d.h. R und S haben nur $B_1, \ldots, B_k$ als gleichbenannte Attribute. In diesem Fall ist das Ergebnis von $R \bowtie S$ wie folgt definiert:

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

Logisch wird also das Kreuzprodukt gebildet, aus dem dann nur diejenigen Tupel selektiert werden, deren Attributwerte für gleichbenannte Attribute der beiden Argumentrelationen gleich sind. Weiterhin werden diese gleichbenannten Attribute in das Ergebnis nur einmal übernommen – das wird durch die abschließende Projektion erzielt. Tabellarisch kann man sich das Schema der Ergebnisrelation wie folgt veranschaulichen:

$R \bowtie S$											
$\mathcal{R} - \mathcal{S}$				$\mathcal{R} \cap \mathcal{S}$				$\mathcal{S} - \mathcal{R}$			
A_1	A_2	$\ldots$	A_m	B_1	B_2	$\ldots$	B_k	C_1	C_2	$\ldots$	C_n
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Als Beispiel für den natürlichen Verbund betrachten wir folgende Datenbankanfrage, in der wir *Studenten* mit den von ihnen gehörten *Vorlesungen* assoziieren:

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

In dieser Anfrage wird der Join zunächst für die Argumentrelationen *Studenten* und *hören* durchgeführt. Die so erzeugte Ergebnisrelation wird dann noch mit *Vorlesungen* verbunden. Das Ergebnis dieses sogenannten 3-Wege-Joins sieht dann wie folgt aus:

(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	2134
28106	Carnap	3	4052	Wissenschaftstheorie	3	2126
...	...	...	...	...	...	...

Man beachte, dass beim natürlichen Join die Qualifizierung der Attribute – also das Voranstellen des Relationennamens zur eindeutigen Benennung – nicht notwendig ist, da gleichbenannte Attribute nur einmal übernommen werden. Bei der Auswertung des ersten Joins *Studenten* $\bowtie$ *hören* erhält man eine vierstellige Ergebnisrelation, da *Studenten* (mit drei Attributen) und *hören* (mit zwei Attributen) nur ein gleichbenanntes Attribut – nämlich *MatrNr* – haben. Dieses Attribut nennt man

das *Joinattribut*. Der zweite Join wird über dem Joinattribut *VorlNr* ausgewertet, dem einzigen gleichbenannten Attribut von (*Studenten* ⋈ *hören*) und *Vorlesungen*.

Man hätte übrigens bei der Formulierung des obigen „3-Wege-Joins" die Klammerung auch weglassen können, also:

$$\text{Studenten} \bowtie \text{hören} \bowtie \text{Vorlesung}$$

Diese Formulierung hätte dann in zwei unterschiedlichen Reihenfolgen abgearbeitet werden können:

1. (Studenten ⋈ hören) ⋈ Vorlesungen
2. Studenten ⋈ (hören ⋈ Vorlesungen)

Der Joinoperator ist aber *assoziativ*, so dass die beiden Alternativen äquivalent sind. Logischerweise ist der Joinoperator auch *kommutativ* – siehe dazu Übungsaufgabe 3.13.

Es kommt manchmal vor, dass man Relationen über zwei Attribute „joinen" will, die zwar die gleiche Bedeutung aber unterschiedliche Benennungen haben. Ein Beispiel ist die Verbindung von *Vorlesungen* mit *Professoren*, wobei der Join über den Attributen *Vorlesungen.gelesenVon* und *Professoren.PersNr* auszuwerten ist. In diesem Fall ist eine Umbenennung zumindest eines der Attribute notwendig. Dies kann man mit dem überladenen ρ-Operator, der sowohl zur Relationen- als auch zur Attributumbenennung verwendet wird, durchführen. Für unser Beispiel erhalten wir also folgende Anfrage:

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

Die resultierende Relation hat dann folgendes Schema:

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

Allgemeiner Join

Beim natürlichen Verbund werden alle gleichbenannten Attribute der beiden Argumentrelationen betrachtet. Qualifizierende Tupel müssen für alle diese Attribute gleiche Werte aufweisen, um in das Ergebnis einzugehen. Der allgemeine Joinoperator, auch *Theta-Join* genannt, erlaubt die Spezifikation eines beliebigen Joinprädikats θ. Als abstraktes Beispiel betrachten wir die Relationen R und S mit Attributen $A_1, \ldots, A_n$ und $B_1, \ldots, B_m$. Ein Theta-Join dieser beiden Relationen sieht wie folgt aus:

$$R \bowtie_\theta S$$

Hierbei ist θ ein beliebiges Prädikat über den Attributen $A_1, \ldots, A_n, B_1, \ldots, B_m$. Ein Beispiel wäre:

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

Das Ergebnis dieses Joins hat $n + m$ Attribute – unabhängig davon, ob einige Attribute gleich benannt sind. Insbesondere wenn Attribute in R und S gleich benannt sind, werden sie durch Qualifizierung mit dem Namen ihrer Ursprungsrelation eindeutig benannt, also z.B. $R.A_1$ oder $S.B_1$.

$R \bowtie_\theta S$							
$\mathcal{R}$				$\mathcal{S}$			
A_1	A_2	...	A_n	B_1	B_2	...	B_m
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Der Theta-Join ist also nur eine vereinfachte Formulierung des kartesischen Produkts gefolgt von der Selektion:

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

Der wesentliche Unterschied besteht darin, dass der Join i.A. effizienter ausgewertet werden kann, da schon frühzeitig nicht qualifizierende Tupel aus dem Kreuzprodukt der beiden Argumentrelationen eliminiert werden können.

Einen Theta-Join der Form $R \bowtie_{R.A_i = S.B_j} S$ nennt man Equi-Join. Es können beim Equi-Join auch mehrere „=“-Vergleiche konjunktiv verknüpft werden. Der Unterschied zum natürlichen Verbund besteht darin, dass die Attribute nicht notwendigerweise gleich benannt sein müssen und dass beide Attribute – sowohl $R.A_i$ als auch $S.B_j$ – ins Ergebnis übernommen werden; auch wenn sie gleich benannt sein sollten.

Um ein sinnvolles Beispiel für einen allgemeinen Join formulieren zu können, wollen wir das Schema der Relationen *Professoren* und *Assistenten* jeweils um das Attribut *Gehalt* erweitern. Dann könnte man an den folgenden Paaren von *Professoren* und *Assistenten* interessiert sein:

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

Hierbei werden den *Professoren* die ihnen „untergebenen“, aber besser bezahlten *Assistenten* zugeordnet. Das Schema der Ergebnisrelation hat demnach folgende Attribute: *Professoren.PersNr*, *Professoren.Name*, *Professoren.Rang*, *Professoren.Raum*, *Professoren.Gehalt*, *Assistenten.PersNr*, *Assistenten.Name*, *Assistenten.Boss*, *Assistenten.Fachgebiet*, *Assistenten.Gehalt*.

Weitere Join-Operatoren

Die bislang eingeführten Join-Operatoren nennt man manchmal auch „innere“ Joins. Bei diesen Operatoren gehen im Ergebnis diejenigen Tupel der Argumentrelationen verloren, die keinen „Joinpartner“ gefunden haben. Bei den *äußeren* Join-Operatoren (engl. *outer joins*) werden – je nach Typ des Joins – auch partnerlose Tupel der linken, der rechten bzw. beider Argumentrelation(en) „gerettet“:

- linker äußerer Join, left outer join ($\bowtie$): Die Tupel der linken Argumentrelation bleiben in jedem Fall erhalten.
- rechter äußerer Join, right outer join ($\bowtie$): Die Tupel der rechten Argumentrelation bleiben in jedem Fall erhalten.
- (vollständiger) äußerer Join, (full) outer join ($\bowtie$): Die Tupel beider Argumentrelationen bleiben in jedem Fall erhalten.

- **natürlicher Join**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1

- **linker äußerer Join**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_2	b_2	c_2	–	–

- **rechter äußerer Join**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
–	–	c_3	d_2	e_2

- **äußerer Join**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\bowtie$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat				
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

- **Semi-Join von L mit R**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\ltimes$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat		
A	B	C
a_1	b_1	c_1

- **Semi-Join von R mit L**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\rtimes$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat		
C	D	E
c_1	d_1	e_1

- **Anti Semi-Join von L mit R**

L		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2

$\triangleright$

R		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2

$=$

Resultat		
A	B	C
a_2	b_2	c_2

Abbildung 3.9: Beispielanwendungen der verschiedenen Join-Operatoren

In Abbildung 3.9 sind die drei unterschiedlichen äußeren Join-Operatoren auf die beiden abstrakten Argumentrelationen L und R angewendet. Das Ergebnis in der Relation *Resultat* hat dann für den linken äußeren Join folgende Form: Die beiden zueinander passenden Tupel $[a_1, b_1, c_1]$ der Relation L und $[c_1, d_1, e_1]$ der Relation R werden zum Ergebnistupel $[a_1, b_1, c_1, d_1, e_1]$ kombiniert. Das „linke" Tupel $[a_2, b_2, c_2]$ ohne Joinpartner in der rechten Relation R wird mit Null-Werten – hier als '–' dargestellt – aufgefüllt und in das Ergebnis übernommen.

Beim rechten äußeren Join wird entsprechend das Tupel $[c_3, d_2, e_2]$ aus R, dem der Joinpartner in L fehlt, nach links mit Null-Werten aufgefüllt.

Beim vollständigen äußeren Join werden Tupel ohne Joinpartner sowohl aus der linken als auch aus der rechten Argumentrelation gerettet und nach links bzw. nach rechts mit Null-Werten aufgefüllt.

Die äußeren Join-Operatoren sind natürlich durch Kombinationen anderer Relationenalgebra-Ausdrücke ersetzbar (siehe Übungsaufgabe 3.7).

Weiterhin sind in Abbildung 3.9 die sogenannten Semi-Join-Operatoren $\rtimes$ und $\ltimes$ am Beispiel gezeigt. Der Semi-Join von L mit R – in Zeichen $L \ltimes R$ – ist definiert als ($\mathcal{L}$ bezeichne die Menge der Attribute von L):

$$L \ltimes R = \Pi_{\mathcal{L}}(L \bowtie R)$$

Das Ergebnis enthält also all die Tupel aus L in unveränderter Form, die einen potentiellen Joinpartner in R haben.

Der Semi-Join von R mit L – in Zeichen $L \rtimes R$ – ist analog definiert. Es gilt natürlich folgende triviale Äquivalenz:

$$L \rtimes R = R \ltimes L$$

Andererseits sind die linken und rechten äußeren Joins sowie die beiden Semi-Joins nicht kommutativ.

Das Komplement des Semijoins bezeichnet man als Anti-Semi-Join – in Zeichen $L \triangleleft R$ bzw. $L \triangleright R$. Aufbauend auf dem korrespondierenden Semi-Join ist der Anti-Semi-Join als Mengendifferenz wie folgt definiert:

$$L \triangleright R = L - (L \ltimes R)$$

Der Anti-Semi-Join kommt, wie wir später sehen werden, bei Anfragen der Art „Finde die Elemente aus L, für die gilt: es existiert kein Pendant (also kein Join-Partner) in R" zum Einsatz.

3.4.9 Mengendurchschnitt

Als Beispielanwendung für den Mengendurchschnitt (Operatorsymbol $\cap$) betrachten wir folgende Anfrage: Finde die *PersNr* aller C4-Professoren, die mindestens eine Vorlesung halten.

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

Man beachte, dass der Mengendurchschnitt nur auf zwei Argumentrelationen mit gleichem Schema anwendbar ist. Deshalb ist die Umbenennung des Attributs *gelesenVon* in *PersNr* in der Relation *Vorlesungen* notwendig.

Der Mengendurchschnitt zweier Relationen $R \cap S$ kann durch die Mengendifferenz wie folgt ausgedrückt werden:

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

3.4.10 Die relationale Division

Der Divisionsoperator $\div$ wird für Anfragen eingesetzt, in denen eine Allquantifizierung vorkommt. Als Beispiel betrachten wir die Anfrage, in der die *MatrNr* derjenigen *Studenten* gesucht wird, die *alle* vierstündigen *Vorlesungen* belegt haben. Zunächst könnten wir die *VorlNr* der vierstündigen *Vorlesungen* bestimmen:

$$L := \Pi_{\text{VorlNr}}(\sigma_{\text{SWS}=4}(\text{Vorlesungen}))$$

Danach können wir aus der Relation *hören* die *MatrNr* der Studenten ermitteln, die alle in L enthaltenen *Vorlesungen* gehört haben. Dazu bedienen wir uns des Divisionsoperators:

$$\text{hören} \div \overbrace{\Pi_{\text{VorlNr}}(\sigma_{\text{SWS}=4}(\text{Vorlesungen}))}^{L}$$

Das Schema der Ergebnisrelation besteht aus nur einem Attribut, nämlich *MatrNr*.

Formal ist der Divisionsoperator für zwei Argumentrelationen R und S mit den Schemata $\mathcal{R}$ und $\mathcal{S}$ wie folgt definiert. Für die Durchführung der Division $R \div S$ muss gelten, dass $\mathcal{S}$ eine Teilmenge von $\mathcal{R}$ ist, also: $\mathcal{S} \subseteq \mathcal{R}$. Dann ist das Schema der Ergebnisrelation als die Mengendifferenz $\mathcal{R} - \mathcal{S}$ definiert. Das Ergebnis enthält also nur die Attribute der Argumentrelation R, die nicht auch in S enthalten sind.

Ein Tupel t ist in $R \div S$ enthalten, falls es für jedes Tupel t_s aus S ein Tupel t_r aus R gibt, so dass die beiden folgenden Bedingungen erfüllt sind:

$$\begin{aligned} t_r.\mathcal{S} &= t_s.\mathcal{S} \\ t_r.(\mathcal{R} - \mathcal{S}) &= t \end{aligned}$$

Hierbei ist $t_r.\mathcal{S} = t_s.\mathcal{S}$ eine Kurzform für:

$$\forall A \in \mathcal{S} : t_r.A = t_s.A$$

Wir wollen den Divisions-Operator an einer abstrakten Ausprägung der Relationen H und L demonstrieren:

H	
M	V
m_1	v_1
m_1	v_2
m_1	v_3
m_2	v_2
m_2	v_3

$\div$

L
V
v_1
v_2

$=$

$H \div L$
M
m_1

Die Division ergibt für dieses Beispiel also eine einstellige Relation mit nur einem Tupel, nämlich $[m_1]$. Für jedes Tupel $[v_i]$ der Relation L existiert nämlich ein Tupel der Form $[m_1, v_i]$ in der Relation H – wobei hier $(i \in \{1, 2\})$ gilt.

Es ist möglich, den Divisionsoperator durch andere (in Abschnitt 3.4.7 eingeführte) Operatoren auszudrücken. Es gilt nämlich:

$$R \div S = \Pi_{(\mathcal{R}-\mathcal{S})}(R) - \Pi_{(\mathcal{R}-\mathcal{S})}((\Pi_{(\mathcal{R}-\mathcal{S})}(R) \times S) - R)$$

Der Beweis dieser Äquivalenz wird den Lesern als Übungsaufgabe (siehe Aufgabe 3.6) überlassen.

3.4.11 Gruppierung und Aggregation

Bei der Gruppierung werden Tupel mit gleichen Attributwerten (für eine Liste von Attributnamen) gruppiert. Auf jede Gruppe wird dann eine Aggregatfunktion angewendet, die für die gesamte Gruppe einen einzelnen Wert berechnet. Typische Aggregatfunktionen sind **count**, um die Anzahl der Elemente der Gruppe zu zählen; **sum**, um die Werte eines spezifizierten Attributs zu summieren; **max**, **min** und **avg**, um das Maximum bzw. das Minimum oder den Durchschnitt eines Attributs pro Gruppe zu bestimmen. Die Gruppierung und Aggregation geht über die Standard-Operatoren der relationen Algebra hinaus und wird mit dem Operatorsymbol Γ bezeichnet. Wenn wir beispielsweise pro Semester-Wert die Anzahl der Studenten zählen wollen, geht das mit diesem Ausdruck:

$$\Gamma_{\text{Semester};\mathbf{count}(*)}(\text{Studenten})$$

Als Ergebnis erhält man eine Relation dieser Form (für unsere sehr kleine Universitäts-Datenbank):

$\Gamma_{\text{Semester};\mathbf{count}(*)}(\text{Studenten})$	
Semester	**count(*)**
18	1
12	1
10	1
8	1
6	1
3	1
2	2

Die Reihenfolge der Ergebnistupel ist nicht vorgegeben. Allgemein kann man eine Liste von Gruppierungsattributen angeben und eine Liste von Funktionen, die jeweils auf jede resultierende Gruppe angewendet werden. Wichtig ist, dass das Ergebnis pro Gruppe nur ein Ergebnistupel enthält. Die Stelligkeit der Ergebnisrelation ergibt sich aus der Anzahl der Gruppierungsattribute plus der Anzahl der Aggregatfunktionen. Die Anfrage

$$\Gamma_{\text{gelesenVon};\mathbf{count}(*),\mathbf{sum}(\text{SWS})}(\text{Vorlesungen})$$

liefert also die dreistellige Ergebnisrelation:

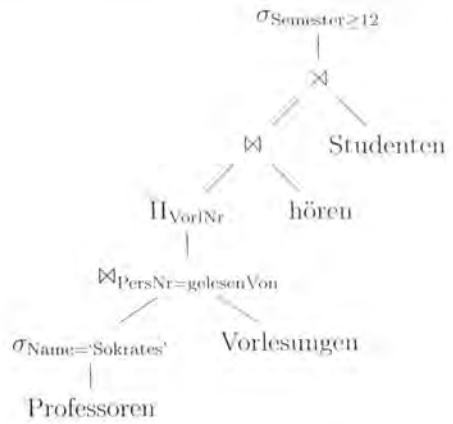


Abbildung 3.10: Baumdarstellung des Algebra-Ausdrucks

$\Gamma_{\text{gelesenVon};\textbf{count}(*),\textbf{sum}(\text{SWS})}(\text{Vorlesungen})$		
gelesenVon	**count**(*)	**sum**(SWS)
2125	3	10
2126	3	8
2133	1	2
2134	1	2
2137	2	8

3.4.12 Operatorbaum-Darstellung

Bislang haben wir Relationenalgebra-Ausdrücke immer „in-line" dargestellt. Bei komplizierteren Anfragen ist aber eine sogenannte Operatorbaum-Darstellung übersichtlicher.

Wir wollen dies an folgender Anfrage illustrieren: Finde Sokrates' Dauerstudenten, also die Studenten, die mindestens eine Vorlesung von Sokrates gehört haben und schon im 12. oder noch höherem Semester sind.

Als Operatorbaum ist die Anfrage in Abbildung 3.10 gezeigt. Die Leser mögen zum Vergleich die „in-line"-Darstellung dieser Anfrage aus dem Operatorbaum ableiten.

3.5 Der Relationenkalkül

Ausdrücke in der Relationenalgebra spezifizieren, wie das Ergebnis der Anfrage zu berechnen ist. Diese prozedurale Berechnungsvorschrift ergibt sich aus den Algebraoperatoren. Besonders deutlich wird das an der Operatorbaum-Darstellung, wo man sich veranschaulichen kann, dass die Zwischenergebnis-Tupel von unteren zu weiter oben angeordneten Operatoren weitergeleitet werden.

Demgegenüber ist der *Relationenkalkül* stärker *deklarativ* orientiert, d.h. es werden die qualifizierenden Ergebnistupel beschrieben, ohne dass eine Herleitungsvor-

schrift angegeben wird. In anderer Hinsicht sind der Relationenkalkül und die relationale Algebra aber nahe verwandt: Sie sind gleich mächtig. D.h. eine Anfrage, die in der Relationenalgebra formuliert ist, kann auch im Relationenkalkül ausgedrückt werden und umgekehrt.

Der Relationenkalkül basiert auf dem mathematischen Prädikatenkalkül erster Stufe, der quantifizierte Variablen und Werte zulässt. Es gibt zwei unterschiedliche, aber gleich mächtige Ausprägungen des Relationenkalküls:

1. Der relationale Tupelkalkül und
2. der relationale Domänenkalkül.

Der Unterschied besteht darin, dass Variablen des Kalküls im ersten Fall an Tupel einer Relation gebunden werden und im zweiten Fall an Domänen, die als Wertemengen von Attributen vorkommen. Hinsichtlich ihrer Ausdruckskraft sind die beiden Kalküle, wie oben gesagt, gleich mächtig, so dass sich jede Anfrage des Tupelkalküls in eine äquivalente Anfrage des Domänenkalküls umformulieren lässt und umgekehrt.

3.5.1 Beispielanfrage im relationalen Tupelkalkül

Anfragen im relationalen Tupelkalkül haben folgende generische Form:

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

Hierbei ist t eine sogenannte Tupelvariable und P ist ein *Prädikat*, das erfüllt sein muss, damit t in das Ergebnis aufgenommen wird. Die Variable t ist eine sogenannte *freie* Variable des Prädikats P, d.h. t darf nicht durch einen Existenz- oder Allquantor quantifiziert sein.

Als konkretes Beispiel formulieren wir die Anfrage nach allen C4-Professoren im Tupelkalkül:

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

In dieser Anfrage werden zwei Bedingungen an die aktuelle Belegung von p gestellt:

1. Das Tupel p muss in der Relation *Professoren* enthalten sein.
2. Das Tupel p muss für das Attribut *Rang* den Wert 'C4' besitzen.

Man kann sich die Auswertung dieser Anfrage so vorstellen, dass p gemäß Bedingung 1. sukzessive an alle Tupel der Relation *Professoren* gebunden wird und dann Bedingung 2. ausgewertet wird.

Es ist auch möglich, neue noch nicht in der Datenbank existierende Tupel aufzubauen. Dazu wird in der Kalkülanfrage links vom „|"-Zeichen der Tupelkonstruktor [...] verwendet. Eine solche Anfrage hat dann folgende Struktur:

$$\{[t_1.A_1, \ldots, t_n.A_n] \mid P(t_1, \ldots, t_n)\}$$

Hierbei sind $t_1, \ldots, t_n$ Tupelvariablen und $A_1, \ldots, A_n$ Attributnamen. Das so erzeugte Ergebnis ist also eine n-stellige Relation. Die Attribute $A_1, \ldots, A_n$ müssen

natürlich im Schema der Relationen enthalten sein, an die $t_1, \ldots, t_n$ gebunden werden. Die Tupelvariablen können durchaus mehrfach vorkommen (also $t_i = t_j$ für $i \neq j$), damit man aus einer Relation mehrere Attribute ausgeben kann.

Als konkretes Beispiel formulieren wir die Anfrage, in der die Paare von *Professoren* (*Name*) und den ihnen zugeordneten *Assistenten* (*PersNr*) gebildet werden. Hierzu ist also der Join dieser beiden Relationen notwendig.

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

Bei dieser Anfrage wird also p an die Tupel der Relation *Professoren* und a an die Tupel der Relation *Assistenten* gebunden. Konzeptuell werden dann alle möglichen Kombinationen (Bindungen) für a und p gebildet um dafür die weitere Bedingung $p.PersNr = a.Boss$ zu überprüfen. Diese Bedingung ist demnach ein Joinprädikat, da es sich auf zwei Tupelvariablen bezieht. Aus den qualifizierenden Bindungen für a und p werden dann jeweils die beiden interessierenden Attribute projiziert und als neues 2-stelliges Tupel in das Ergebnis aufgenommen.

3.5.2 Quantifizierung von Tupelvariablen

Der Tupelkalkül erlaubt die *Existenz-* und *Allquantifizierung* von Tupelvariablen. Mit der Existenzquantifizierung wird das umgangssprachliche „es existiert ein ..." und mit der Allquantifizierung (oft auch Universalquantifizierung genannt) das umgangssprachliche „für alle ..." ausgedrückt. Die Quantifizierungen werden für ein Prädikat $Q(t)$ wie folgt notiert:

$$\exists t \in R(Q(t)) \quad \text{bzw.} \quad \forall t \in R(Q(t))$$

Die erste Form ist die Existenzquantifizierung; die zweite die Universalquantifizierung. Hierbei wird vereinfachend angenommen, dass die Tupelvariable t in $Q(t)$ nicht schon anderweitig quantifiziert war – d.h. t ist *frei* in $Q(t)$.

Die hier betrachtete Quantifizierung bindet die Tupelvariable t gleichzeitig an eine Relation R. Demnach bedeutet die erste Form, dass es ein Tupel t in der Relation R gibt, für das $Q(t)$ erfüllt (wahr) ist. Die zweite Variante verlangt, dass $Q(t)$ für alle Tupel der Relation R erfüllt ist.

Als konkretes Beispiel betrachten wir folgende Anfrage, in der die *Studenten* ermittelt werden, die mindestens eine Vorlesung bei der Professorin namens Curie gehört haben.

$$\begin{aligned}\{\text{s} \mid \text{s} \in \text{Studenten}\\ \wedge\ \exists \text{h} \in \text{hören}(\text{s.MatrNr} = \text{h.MatrNr}\\ \wedge\ \exists \text{v} \in \text{Vorlesungen}(\text{h.VorlNr} = \text{v.VorlNr}\\ \wedge\ \exists \text{p} \in \text{Professoren}(\text{p.PersNr} = \text{v.gelesenVon}\\ \wedge\ \text{p.Name} = \text{'Curie'}))\}\end{aligned}$$

In dieser Anfrage sind die Tupelvariablen h, v und p existenzquantifiziert und jeweils an die Relationen *hören*, *Vorlesungen* und *Professoren* gebunden. Die an *Studenten* gebundene Tupelvariable s kommt – als Ergebnisvariable – in dem Prädikat frei vor.

Als Beispiel für die Allquantifizierung (oder Universalquantifizierung) wollen wir die Anfrage aus Abschnitt 3.4.10 hier im relationalen Tupelkalkül formulieren. Es

geht also darum, diejenigen *Studenten* zu finden, die *alle* vierstündigen *Vorlesungen* gehört haben.

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

Hierbei wird also verlangt, dass für *alle* Elemente (Tupel) v der Relation *Vorlesungen*, deren $v.SWS$-Attribut den Wert 4 hat, ein Tupel h in *hören* existiert, aus dem hervorgeht, dass die aktuell betrachtete Belegung für s diese vierstündige Vorlesung hört.

3.5.3 Formale Definition des Tupelkalküls

Ein Ausdruck des Tupelkalküls der Form[5]

$$\{v \mid F(v)\}$$

besteht aus einer Ergebnisspezifikation links vom „|"-Zeichen und einer *Formel* $F(v)$ (Prädikat) mit freier Tupelvariablen v. In der Formel F können weitere Tupelvariablen vorkommen, die aber dann nicht *frei* sein können. Wie bereits angedeutet, bezeichnet man eine Variable als *frei*, falls sie nicht durch $\exists$ oder $\forall$ quantifiziert ist.

Eine Formel wird aus *Atomen*, den Grundbausteinen, zusammengebaut, wobei ein Atom folgende Form hat:

- $s \in R$, wobei s eine Tupelvariable und R ein Relationenname ist.
- $s.A \, \phi \, t.B$, wobei s und t Tupelvariablen, A und B Attributnamen und ϕ ein Vergleichsoperator sind. Als Vergleichsoperatoren sind $=$, $\neq$, $<$, $\leq$, $>$ und $\geq$ erlaubt, wobei die Wertebereiche von $s.A$ und $t.B$ entsprechend definiert sein müssen, damit der Vergleichsoperator anwendbar ist (z.B. ist auf den Wertebereich Boolean nur $=$ und $\neq$ anwendbar).
- $s.A \, \phi \, c$, wobei $s.A$ die gleiche Bedeutung wie oben hat und c eine Konstante darstellt. Hierbei muss c ein Element des Wertebereichs von $s.A$ sein.

Formeln werden nun nach folgenden Regeln aufgebaut:

- Alle Atome sind Formeln.
- Falls P eine Formel ist, dann sind auch $\neg P$ und (P) Formeln.
- Falls P_1 und P_2 Formeln sind, dann sind auch $P_1 \wedge P_2$, $P_1 \vee P_2$ und $P_1 \Rightarrow P_2$ Formeln.
- Falls $P(t)$ eine Formel mit freier Variablen t ist, dann sind auch

  $$\forall t \in R(P(t)) \quad \text{und} \quad \exists t \in R(P(t))$$

  Formeln.

[5] Hier nehmen wir vereinfachend an, dass in der Ergebnisspezifikation nur eine Tupelvariable vorkommt. Die Definition lässt sich aber leicht verallgemeinern.

Eigentlich wäre es ausreichend gewesen, nur den Existenz- oder den Allquantor einzuführen, da folgende Äquivalenzen gelten:

$$\begin{aligned} \forall t \in R(P(t)) &= \neg(\exists t \in R(\neg P(t))) \\ \exists t \in R(P(t)) &= \neg(\forall t \in R(\neg P(t))) \end{aligned}$$

3.5.4 Sichere Ausdrücke des Tupelkalküls

Ausdrücke des Tupelkalküls können in einigen Fällen unendliche Ergebnisse spezifizieren. Als Beispiel betrachten wir folgende Anfrage:

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

Natürlich kann man sich unendlich viele Tupel vorstellen, die nicht in der Relation *Professoren* enthalten sind – von denen die meisten auch gar nicht in der (endlichen) Datenbank enthalten sind.

Um diesem unerwünschten Effekt entgegenzuwirken, wird eine Einschränkung bei der Formulierung von Anfragen im Tupelkalkül auf sogenannte *sichere* Anfragen vollzogen. Für die Definition dieser Einschränkung benötigen wir als neues Konzept die *Domäne* einer Formel. Sie enthält alle Werte, die als Konstante in der Formel vorkommen und alle Werte (d.h. Attributwerte in Tupeln) der Relationen, die in der Formel referenziert (d.h. namentlich erwähnt) werden. Zum Beispiel enthält die Domäne der in der folgenden Anfrage

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

spezifizierten Formel den Wert 'C4' und alle Attributwerte der Relation *Professoren* – also z.B. 2125, 'Curie', 'C4', etc.

Ein Ausdruck des Tupelkalküls heißt *sicher*, wenn das Ergebnis des Ausdrucks eine Teilmenge der Domäne ist. Für sichere Ausdrücke ist natürlich garantiert, dass das Ergebnis endlich ist. Warum?

Mit Ausnahme der Anfrage

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

sind alle in diesem Abschnitt formulierten Anfragen sicher. Die obige Anfrage ist deshalb nicht *sicher*, weil die Domäne nur Tupel aus der Relation *Professoren* enthält: das Ergebnis aber gerade andere Werte (Tupel) spezifiziert.

3.5.5 Der relationale Domänenkalkül

Im Unterschied zum Tupelkalkül werden Variablen im Domänenkalkül an Domänen, d.h. Wertemengen von Attributen, gebunden. Eine Anfrage im Domänenkalkül hat folgende generische Struktur:

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

Hierbei sind die v_i $(1 \leq i \leq n)$ Variablen, genauer gesagt Domänenvariablen, die einen Attributwert repräsentieren. P ist ein Prädikat (bzw. eine Formel) mit den freien Variablen $v_1, \ldots, v_n$.

Formeln im Domänenkalkül werden genauso aus Atomen zusammengesetzt wie im Tupelkalkül. Bei den Atomen gibt es einen Unterschied, der darin besteht, dass man jetzt keine Bindung einer einzelnen Variable an eine Relation mehr hat, sondern eine Sequenz von Domänenvariablen an eine Relation bindet.

- $[w_1, w_2, \ldots, w_m] \in R$ ist ein Atom, wobei R eine m-stellige Relation ist. Die Zuordnung der m Domänenvariablen $w_1, \ldots, w_m$ zu den Attributen der Relation R erfolgt nach der Reihenfolge der Attribute im Schema.
- $x \, \phi \, y$ ist ein Atom, wobei x und y Domänenvariablen sind und ϕ ein Vergleichsoperator ($=$, $\neq$, $<$, $\leq$, $>$ oder $\geq$) ist, der auf die Domäne anwendbar ist.
- $x \, \phi \, c$ ist ein Atom mit der Domänenvariable x, der Konstanten c und dem Vergleichsoperator ϕ. Hierbei muss der Vergleichsoperator ϕ auf diese Domäne – in der c enthalten sein muss – anwendbar sein.

Formeln werden aus den oben beschriebenen Atomen – als „Grundbausteine" – wie folgt zusammengesetzt:

- Ein Atom ist eine Formel.
- Falls P eine Formel ist, dann sind $\neg P$ und (P) auch Formeln.
- Falls P_1 und P_2 Formeln sind, dann sind auch $P_1 \vee P_2$, $P_1 \wedge P_2$ und $P_1 \Rightarrow P_2$ Formeln.
- Falls $P(v)$ eine Formel mit freier Variablen v ist, dann sind auch $\exists v(P(v))$ und $\forall v(P(v))$ Formeln.

Als Abkürzung schreiben wir z.B. $\exists v_1, v_2, v_3(P(v_1, v_2, v_3))$ anstatt der formal korrekten, aber umständlicheren Form $\exists v_1(\exists v_2(\exists v_3(P(v_1, v_2, v_3))))$.

3.5.6 Beispielanfragen im Domänenkalkül

Als Beispielanfrage wollen wir die *MatrNr* und *Namen* der Studenten finden, die schon mindestens eine Prüfung bei Curie abgelegt haben:

$$\{[\mathrm{m,\ n}] \mid \exists \mathrm{s}([\mathrm{m,\ n,\ s}] \in \mathrm{Studenten} \wedge \exists \mathrm{v,\ p,\ g}([\mathrm{m,\ v,\ p,\ g}] \in \mathrm{prüfen} \wedge \exists \mathrm{a,\ r,\ b}([\mathrm{p,\ a,\ r,\ b}] \in \mathrm{Professoren} \wedge \mathrm{a} = \text{'Curie'})))\}$$

Im Domänenkalkül werden Joinbedingungen i.A. implizit durch die Verwendung derselben Domänenvariablen spezifiziert. In der obigen Anfrage wird z.B. die Variable m dafür verwendet, den Join zwischen *Studenten* und *prüfen* zu vollziehen: Es wird nämlich implizit gefordert, dass die *MatrNr* – die durch m repräsentiert wird – in beiden Tupeln $[m, n, s] \in$ *Studenten* und $[m, v, p, g] \in$ *prüfen* dieselbe ist.

Man könnte diese Joinbedingungen natürlich auch explizit formulieren. Dies führt zu der folgenden äquivalenten Anfrage:

$$\{[\mathrm{m,\ n}] \mid \exists \mathrm{s}([\mathrm{m,\ n,\ s}] \in \mathrm{Studenten} \wedge \exists \mathrm{m'},\ \mathrm{v,\ p,\ g}([\mathrm{m'},\ \mathrm{v,\ p,\ g}] \in \mathrm{prüfen} \wedge \mathrm{m} = \mathrm{m'} \wedge \exists \mathrm{p'},\ \mathrm{a,\ r,\ b}([\mathrm{p'},\ \mathrm{a,\ r,\ b}] \in \mathrm{Professoren} \wedge \mathrm{p} = \mathrm{p'} \wedge \mathrm{a} = \text{'Curie'})))\}$$

3.5.7 Sichere Ausdrücke des Domänenkalküls

Analog zum Tupelkalkül ist man auch bei der Formulierung von Anfragen im Domänenkalkül prinzipiell in der Lage, unendliche Ergebnisse zu spezifizieren. Als Beispiel diene uns wieder folgende Anfrage:

$$\{[\mathrm{p}, \mathrm{n}, \mathrm{r}, \mathrm{o}] \mid \neg([\mathrm{p}, \mathrm{n}, \mathrm{r}, \mathrm{o}] \in \text{Professoren})\}$$

Hier werden wiederum alle Tupel gesucht, die *nicht* in der Relation *Professoren* enthalten sind – davon gibt es natürlich immer noch unendlich viele.

Wiederum benötigt man den Begriff der Domäne einer Formel, der analog zum Tupelkalkül definiert ist. Die Domäne einer Formel besteht also aus der Menge aller Konstanten, die in der Formel vorkommen, und aller Attributwerte von Relationen, die in der Formel referenziert werden. Die Klasse der *sicheren* Domänenkalkül-Ausdrücke wird dann wie folgt definiert. Ein Ausdruck

$$\{[x_1, x_2, \ldots, x_n] \mid P(x_1, x_2, \ldots, x_n)\}$$

ist sicher, falls folgende drei Bedingungen gelten:

1. Falls das Tupel $[c_1, c_2, \ldots, c_n]$ mit Konstante c_i im Ergebnis enthalten ist, so muss c_i $(1 \leq i \leq n)$ in der Domäne von P enthalten sein.

2. Für jede existenz-quantifizierte Teilformel $\exists x(P_1(x))$ muss gelten, dass P_1 nur für Elemente aus der Domäne von P_1 erfüllbar sein kann – oder evtl. für gar keine. Mit anderen Worten, wenn für eine Konstante c das Prädikat $P_1(c)$ erfüllt ist, so muss c in der Domäne von P_1 enthalten sein.

3. Für jede universal-quantifizierte Teilformel $\forall x(P_1(x))$ muss gelten, dass sie dann und nur dann erfüllt ist, wenn $P_1(x)$ für alle Werte der Domäne von P_1 erfüllt ist. Mit anderen Worten, $P_1(d)$ muss für alle d, die *nicht* in der Domäne von P_1 enthalten sind, auf jeden Fall erfüllt sein.

Diese Bedingungen 2. und 3. konnten in Abschnitt 3.5.4 bei der Definition des sicheren Tupelkalküls weggelassen werden, da wir alle existenz- und universalquantifizierten Tupelvariablen immer an eine existierende (d.h. abgespeicherte) Relation gebunden hatten. Daher waren diese Tupelvariablen automatisch immer an endliche Mengen gebunden. Dies ist beim Domänenkalkül nicht der Fall, da die Variablen an Domänen (also Wertebereiche von Attributen) gebunden werden. Diese können aber i.A. unendlich viele Elemente enthalten – man denke etwa an die Domäne *integer*. Bedingungen 2. und 3. sind eingeführt worden, um zu verhindern, dass man (konzeptuell) unendlich viele Werte „ausprobieren" muss, um die Erfüllbarkeit von $\exists x(P_1(x))$ bzw. $\forall x(P_1(x))$ zu bestimmen. In beiden Fällen kann man sich jetzt wegen Bedingung 2. bzw. 3. – auf die endliche Anzahl von Werten aus der Domäne von $P_1(x)$ beschränken, da die anderen nicht in der Domäne enthaltenen Werte keinen Einfluss auf das Ergebnis eines *sicheren* Domänenkalkül-Ausdrucks haben können. Warum? – siehe Aufgabe 3.14.

3.6 Ausdruckskraft der Anfragesprachen

Codd (1972b) hat die Ausdruckskraft von relationalen Anfragesprachen definiert. In seiner Terminologie heißt eine Anfragesprache *relational vollständig*, wenn sie mindestens so mächtig ist wie die relationale Algebra bzw. der Relationenkalkül. Man braucht nämlich hinsichtlich der Ausdruckskraft zwischen der relationalen Algebra und dem Relationenkalkül nicht zu unterscheiden, da die folgenden drei Sprachen gleiche Ausdruckskraft besitzen:

1. die relationale Algebra,
2. der relationale Tupelkalkül, eingeschränkt auf *sichere* Ausdrücke und
3. der relationale Domänenkalkül, eingeschränkt auf *sichere* Ausdrücke.

Der Beweis lässt sich so vollziehen, dass man zunächst induktiv zeigt, dass jeder Ausdruck der Relationenalgebra in einen äquivalenten Ausdruck des Tupelkalküls transformiert werden kann. Dazu reicht es, zu den sechs Basisoperatoren $\sigma, \Pi, \times, \cup, -, \rho$ äquivalente Ausdrücke des relationalen Tupelkalküls zu spezifizieren.

Dann zeigt man, dass jeder Tupelkalkülausdruck in einen äquivalenten Domänenkalkülausdruck überführt werden kann. Schließlich muss noch – wiederum induktiv – bewiesen werden, dass es zu einem gegebenen Ausdruck des Domänenkalküls einen äquivalenten Algebraausdruck gibt. Wir verweisen auf die Übungsaufgabe 3.8 für den vollständigen Beweis.

3.7 Übungen

3.1 Gegeben sei die ER-Modellierung von Zugverbindungen in Abbildung 3.11.

a) Fügen Sie bei den Beziehungen Kardinalitäten in der (min, max)-Notation hinzu.

b) Übertragen Sie das ER-Modell in ein relationales Schema.

c) Verfeinern Sie das relationale Schema soweit möglich durch Eliminierung von Relationen.

3.2 Man überführe den konzeptuellen Entwurf der Beziehung *betreuen* zwischen *Professoren*, *Studenten* und *Seminarthemen* aus Abbildung 2.5 in ein relationales Schema. Zu diesem Zweck sei angenommen, dass der *Titel* ein Seminarthema eindeutig identifiziere.

Diskutieren Sie, welche Schlüssel Ihre Relationen haben. Inwieweit werden die in Abschnitt 2.7.2 diskutierten Konsistenzbedingungen, die durch die Funktionalitätsangaben spezifiziert wurden, durch das relationale Schema abgedeckt.

3.3 Übertragen Sie das in Aufgabe 2.14 entwickelte ER-Modell für ein Informationssystem zur Bundestagswahl in das zugehörige relationale Schema und verfeinern Sie dieses.

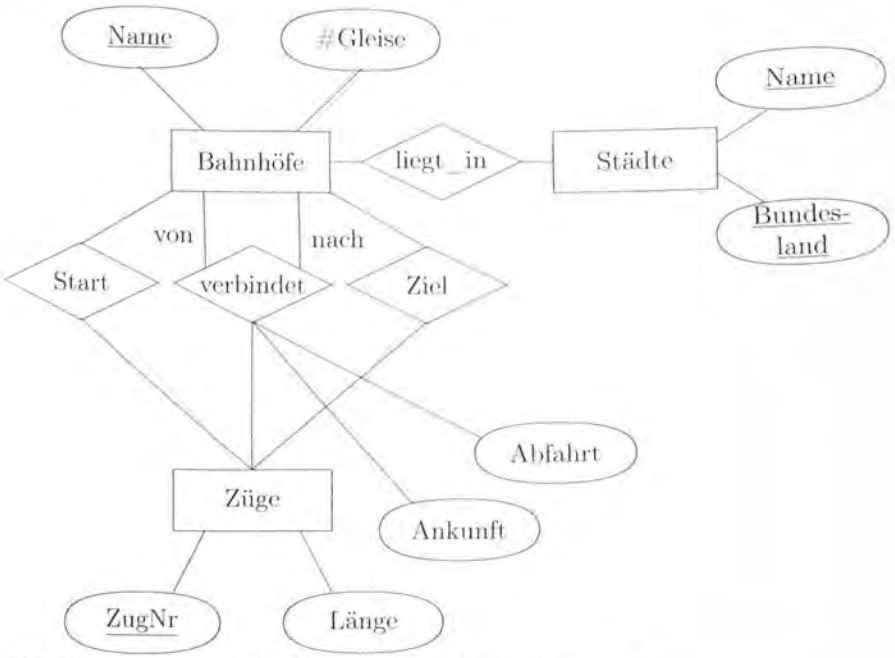


Abbildung 3.11: ER-Modellierung von Zugverbindungen

3.4 Formulieren Sie für das in Aufgabe 3.1 entwickelte relationale Schema folgende Anfragen:

- Finde die direkten Verbindungen von Passau nach Karlsruhe.
- Finde die Verbindungen mit genau einmaligem Umsteigen von Passau nach Aachen – der Umsteigebahnhof ist frei wählbar; aber der Anschlusszug sollte noch am selben Tag fahren.
- Gibt es eine Verbindung mit höchstens dreimaligem Umsteigen von Passau nach Westerland?

Formulieren Sie die Anfragen jeweils

- in der Relationenalgebra,
- im relationalen Tupelkalkül und
- im relationalen Domänenkalkül.

3.5 Eine 1:1-Beziehung der Art

kann man sowohl durch Übernahme des Primärschlüssels von E_2 (als Fremdschlüssel) in E_1 als auch umgekehrt modellieren. Wenn die Beziehung aber

nur für wenige Elemente von E_1 definiert ist, enthält die Relation viele Tupel mit Null-Werten für diesen Fremdschlüssel.

Geben Sie Beispiele aus der realen Welt, wo dies der Fall ist und man die Beziehungen deshalb besser in E_2 repräsentiert.

Geben Sie Beispiele, wo es sowohl für E_1 als auch für E_2 viele Elemente gibt, die die Beziehung R nicht „eingehen". Diskutieren Sie für diesen Fall die Vor- und Nachteile einer separaten Repräsentation der Beziehung als eigenständige Relation.

3.6 Es gelte $\mathcal{S} \subseteq \mathcal{R}$. Beweisen Sie die folgende Äquivalenz:

$$R \div S = \Pi_{(\mathcal{R}-\mathcal{S})}(R) - \Pi_{(\mathcal{R}-\mathcal{S})}((\Pi_{(\mathcal{R}-\mathcal{S})}(R) \times S) - R)$$

Es wird hierdurch also bewiesen, dass der Divisionsoperator die Ausdruckskraft der Relationenalgebra nicht erhöht, sondern nur zur Vereinfachung der Anfrageformulierung eingeführt wurde.

3.7 In Abbildung 3.9 sind die Join-Operatoren $\bowtie$, $\bowtie$ und $\bowtie$ an abstrakten Beispielen eingeführt worden. Geben Sie andere Relationenalgebra-Ausdrücke (ohne Verwendung dieser drei Operatoren) an, die dieselbe Wirkung haben.

Hinweis: $\{|-,-,-|\}$ bezeichnet eine konstante Relation mit einem Tupel, das nur drei NULL-Attributwerte besitzt.

3.8 Beweisen Sie, dass die folgenden drei Sprachen die gleiche Ausdruckskraft besitzen:

a) die relationale Algebra,

b) der relationale Tupelkalkül, eingeschränkt auf *sichere* Ausdrücke und

c) der relationale Domänenkalkül, eingeschränkt auf *sichere* Ausdrücke.

3.9 Finden Sie die *Studenten*, die *Vorlesungen* hören (bzw. gehört haben), für die ihnen die direkten Voraussetzungen fehlen. Formulieren Sie die Anfrage

- in der Relationenalgebra,
- im relationalen Tupelkalkül und
- im relationalen Domänenkalkül.

Erweitern Sie die oben gefundene Menge von Studenten um diejenigen, denen für eine gehörte Vorlesung die indirekten Grundlagen 2. Stufe (also die Vorgänger der Vorgänger der Vorlesung) fehlen. Kommt da was anderes heraus?

Illustrieren Sie die Auswertung am Beispiel der Universitätsdatenbank (Abbildung 3.8).

Für die Relationenalgebra sollten Sie die Anfrage auch als Operatorbaum aufzeichnen.

3.10 Finden Sie die *Professoren*, deren sämtliche *Vorlesungen* nur auf selbst gelesenen (direkten) Vorgängern aufbauen. Formulieren Sie die Anfrage in der

- Relationenalgebra, im
- relationalen Tupelkalkül und im
- relationalen Domänenkalkül.

3.11 Der Allquantor ist durch den Existenzquantor ausdrückbar – und umgekehrt. Formulieren Sie die Anfrage aus Abschnitt 3.5.2 so um, dass nur Existenzquantoren in dem Anfrageprädikat vorkommen. Bei der Anfrage geht es darum, die Studenten zu finden, die *alle* vierstündigen Vorlesungen gehört haben. Begründen Sie die Äquivalenz der beiden alternativen Anfrageformulierungen.

3.12 Finden Sie die *Assistenten* von *Professoren*, die den Studenten Fichte unterrichtet haben – z.B. als potentielle Betreuer ihrer Diplomarbeit.

3.13 Beweisen Sie, dass der natürliche Verbundoperator $\bowtie$ assoziativ ist.

Gilt das auch für den linken (bzw. rechten) äußeren Join und die Semi-Joins?

3.14 Weisen Sie nach, dass bei Einhaltung der drei Bedingungen für sichere Domänenkalkül-Anfragen immer ein endliches Ergebnis spezifiziert wird.

Weisen Sie nach, dass dieses (endliche) Ergebnis, durch Überprüfung endlich vieler Werte gewonnen werden kann.

3.15 Basierend auf der ER-Modellierung eines Verwaltungssystems für eine Leichtathletik-Weltmeisterschaft:

1. Übertragen Sie das ER-Modell in ein relationales Schema. Sie können dabei auf die Berücksichtigung der Generalisierung verzichten, d.h. es reicht, wenn Sie nur das Entity *Helfer* übertragen und die Beziehungen entsprechend anpassen.
2. Verfeinern Sie das relationale Schema soweit möglich durch Eliminierung von Relationen.

3.16 Unser Schema für ein Informationssystem einer Universitätsverwaltung modelliert unter anderem, dass Professoren Vorlesungen anbieten, die von Studenten gehört werden. Nicht berücksichtigt ist jedoch, dass dieselbe Vorlesung in unterschiedlichen Semestern von unterschiedlichen Dozenten gehalten werden kann.

Erweitern Sie den bestehenden Entwurf um diesen Sachverhalt. Ihr neuer Entwurf sollte redundante Datenspeicherung möglichst vermeiden. Überführen Sie anschließend Ihre Modellierung in das zugehörige relationale Schema. Gehen Sie dabei insbesondere auf Fremdschlüsselbeziehungen ein.

3.8 Literatur

Das Relationenmodell wurde in einem bahnbrechenden Aufsatz von Codd (1970) eingeführt – er hat dafür auch den Turing-Preis (die höchste Auszeichnung für

Informatiker) zuerkannt bekommen. Die Grundlagen der Relationenalgebra waren schon in diesem frühen Aufsatz enthalten; der relationale Tupelkalkül wurde von Codd (1972b)] nachgereicht. Die Klassifikation des Relationenkalküls in die tupelbasierte und die domänen-basierte Form stammt von Pirotte (1978).

Die ersten Forschungsprojekte, in denen das relationale Datenmodell realisiert wurde, waren:

- System R, das am IBM Forschungslabor San Jose (jetzt Almaden) entwickelt wurde. Eine Übersicht zu diesem Projekt wurde von Astrahan et al. (1976) verfasst.
- Ingres, das an der University of California, Berkeley unter der Leitung von M. Stonebraker und E. Wong entwickelt wurde. Eine Übersicht wurde von Stonebraker et al. (1976) geschrieben. Eine Sammlung der wichtigsten Forschungspapiere zu Ingres wurde von Stonebraker (1985) zusammengestellt.

Beide Projekte waren Vorläufer von kommerziellen Produkten: SQL/DS ist ein IBM-Produkt aus dem Jahre 1982, das auf System R aufbaut. DB2 ist ein etwas später auf den Markt gekommenes Produkt von IBM, das zwar von den Erfahrungen mit System R profitiert hat, aber vollständig neu realisiert wurde. Die historische Entwicklung relationaler Datenbanksysteme der Firma IBM kann man in dem Bericht von McJones (1995) nachlesen.

Ingres wurde kommerziell von einer Firma namens RTI (Relational Technology, Inc.), die von den universitären Projektleitern gegründet wurde, auf den Markt gebracht.

Interessanterweise war Oracle das erste kommerziell vertriebene relationale Datenbanksystem, obwohl dessen Entwicklung losgelöst von den beiden Forschungsprojekten Ingres und System R erfolgte. Die Hintergründe der Oracle-Entwicklung und des Erfolgs dieses heute führenden Datnbanksystem-Herstellers kann man in dem Buch von Wilson (2003) kurzweilig erfahren.

Weitere relationale Datenbankprodukte sind: Adabas von der Software AG (mittlerweile von SAP unter dem Namen MaxDB vertrieben), Informix (mittlerweile von IBM übernommen), Microsoft SQL Server, NonStopSQL, Sesam von Siemens, Sybase, um nur einige zu nennen.

Das Buch von Maier (1983) widmet sich ausschließlich der relationalen Datenbanktheorie. Es ist eine sehr gute Lektüre zur Vertiefung der formalen Grundlagen. Das Buch ist aber leider nicht mehr über den Verlag lieferbar – es findet sich aber möglicherweise in der jeweiligen Hochschulbibliothek. Ein noch umfangreicheres Buch zur Theorie der relationalen Datenbanken wurde von Abiteboul, Hull und Vianu (1995) verfasst.

Kandzia und Klein (1993) haben ein deutschsprachiges Lehrbuch über die formalen Aspekte des Relationenmodells geschrieben.

4. Relationale Anfragesprachen

Im vorhergehenden Kapitel wurden Anfragen mit Hilfe formaler Anfragesprachen spezifiziert. Die relationale Algebra und der Relationenkalkül bilden die theoretische Grundlage für die Anfragesprache SQL, die von praktisch allen relationalen Datenbanksystemen zur Verfügung gestellt wird.

Im Unterschied zum theoretischen Modell werden in der Praxis einige Vereinfachungen gemacht, die die Benutzer entlasten und eine effizientere Abarbeitung ermöglichen.

Anfragesprachen,[1] wie SQL, sind im Allgemeinen *deklarativ*. Die Benutzer geben nur an, *welche* Daten sie interessieren, und nicht, *wie* die Auswertung der Daten vorgenommen wird. Die oft sehr komplexen, zur Festlegung der Auswertung nötigen Entscheidungen werden vom Anfrageoptimierer des Datenbanksystems übernommen. Dies hat den zusätzlichen Vorteil, dass die eingangs erwähnte physische Datenunabhängigkeit größtenteils gewährleistet werden kann.

Weiterhin realisieren Datenbanksysteme nicht Relationen im eigentlichen mathematischen Sinne, sondern Tabellen, die auch doppelte Einträge enthalten können. Dementsprechend werden in diesem Kapitel auch häufig die Begriffe Zeile und Spalte anstelle von Tupel und Attribut verwendet.

Zusätzlich zur Manipulation von Tabellen beinhalten Anfragesprachen auch Möglichkeiten zur Definition von Integritätsbedingungen für die Daten, zur Vergabe von Zugriffsrechten und zur Transaktionskontrolle. Diese Möglichkeiten sind Thema der nachfolgenden Kapitel.

4.1 Geschichte

Nach der Einführung des relationalen Modells Anfang der 70er Jahre wurde von IBM ein DBMS-Prototyp namens „System R" entwickelt. Die Anfragesprache, die System R bereitstellte, wurde „SEQUEL" genannt (Structured English Query Language) und später in SQL umbenannt. Anfang der 80er Jahre erschien das aus diesem Prototypen entwickelte kommerzielle System SQL/DS. Seither sind eine Vielzahl von anderen relationalen DBMS auf den Markt gebracht worden, unter anderem DB2 von IBM, Oracle von Oracle Corporation und der SQL Server von der Firma Microsoft – um die drei derzeitigen Marktführer zu nennen.

Durch die wachsende Popularität relationaler Systeme wurde bald die Notwendigkeit einer Standardisierung deutlich. Die erste SQL-Norm wurde 1986 von der ANSI-Kommission (American National Standards Institute) verabschiedet. 1989 wurde der Standard das erste Mal revidiert, 1992 entstand das stark erweiterte SQL-92, auch SQL 2 genannt. Seit Ende der neunziger Jahre ist eine Erweiterung unter der Bezeichnung SQL 3 oder SQL-99 als Standard verabschiedet, der aber

[1] Der Begriff Anfragesprache ist historisch geprägt, aber leider etwas verwirrend: Anfragesprachen beinhalten normalerweise auch Befehle zur Datendefinition und Datenmanipulation.

noch nicht von allen Datenbank-Herstellern vollständig umgesetzt wurde. Mittlerweile gibt es nochmals eine Spracherweiterung unter dem Namen SQL-2003, in der insbesondere die XML-Integration festgelegt wird.

Alle in diesem Buch verwendeten Beispiele orientieren sich an den aktuellen Versionen der drei marktführenden Produkte Oracle, IBM DB2 sowie dem Microsoft SQL Server, die den SQL-92 Standard weitgehend unterstützen.

4.2 Datentypen

Relationale Datenbanken stellen in der Hauptsache drei fundamentale Datentypen als Attribut-Domänen zur Verfügung: Zahlen, Zeichenketten und einen Datumstyp **date**. Für jeden dieser Datentypen existieren viele unterschiedliche, historisch geprägte Varianten. Wir werden hier aber nur die wichtigsten durch die ANSI-Kommission festgelegten vorstellen.

Zeichenketten können entweder den Typ **character**(n) oder **char varying**(n) haben. Zeichenketten vom Typ **character** werden im Gegensatz zu **character varying** immer fest mit der angegebenen Größe n abgespeichert und mit Leerzeichen aufgefüllt. **character** kann in beiden Fällen zu **char** abgekürzt werden, äquivalent zu **char varying** kann **varchar** verwendet werden.

Der allgemeinste Zahlentyp ist **numeric**(p, s). Die Angabe von (p, s) ist optional: p gibt die Gesamtzahl der gespeicherten Stellen an, davon werden s als Nachkommastellen reserviert. Zahlen ohne Nachkommastellen können auch als **integer** oder **int** bezeichnet werden. Innerhalb der gegebenen Präzision ist **numeric** exakt. Zusätzlich gibt es noch Datentypen, die einen weiteren Bereich von Zahlen angenähert darstellen können, unter anderem **float**.

In vielen kommerziellen DBMS-Produkten gibt es heute auch schon einen Datentyp namens **blob** oder **raw** für (sehr) große binäre Daten (engl. *binary large object*). Solche Datentypen kann man verwenden, um vom Datenbanksystem nicht zu interpretierende Daten eines externen Anwendungssystems (z.B. eines CAD-Systems) abzuspeichern.

Seit kurzem bieten die relationalen Datenbanksysteme auch Unterstützung für XML-formatierte Daten. Dazu wurde der Datentyp **xml** eingeführt, so dass ein Attribut des Typs **xml** ein XML-Dokument als Wert hat (siehe Abschnitt 20.3.2).

4.3 Schemadefinition

Mit dem Wissen über die Datentypen können jetzt die ersten Tabellen definiert werden. Die zu einer Datenbank gehörenden Tabellendefinitionen werden, wie in Kapitel 3 bereits beschrieben, als das *Schema* der Datenbank bezeichnet. Sie werden automatisch im *Datenwörterbuch* gespeichert. Das Datenwörterbuch beschreibt den Zustand der Datenbank, enthält also *Metadaten*. Es ist eine Sammlung normaler Tabellen, die auch mit den normalen SQL-Anfragebefehlen abgefragt werden können. Es können jedoch im Allgemeinen vom Benutzer keine Änderungen vorgenommen werden.

Eine neue Tabelle wird mit dem **create table** Befehl erzeugt:

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

Nach dem Namen der Tabelle folgt in Klammern eine Liste der Attribute und ihrer Typen, jeweils durch Komma getrennt. Nach einer Typangabe kann zusätzlich noch die Einschränkung **not null** folgen. Dadurch wird erzwungen, dass alle in die Tabelle eingetragenen Tupel an dieser Stelle einen definierten Wert haben. Im Beispiel ist es also nicht möglich, Professoren ohne Namen oder ohne Personalnummer einzutragen. Die Spezifikation **not null** ist eine sogenannte *Integritätsbedingung* und sollte zumindest für alle Primärschlüsselattribute angegeben werden. Integritätsbedingungen werden in Kapitel 5 noch näher erläutert. Dort befindet sich auch in Abbildung 5.4 die vollständige Schemadefinition unserer Universitäts-Datenbank.

Wir haben hier vereinfachend den Typ **integer** als Wertebereich für das Attribut **PersNr** gewählt. Es obliegt somit den Benutzern, eindeutige Personalnummern zu vergeben. Die kommerziellen Datenbanksysteme bieten hierfür aber auch Unterstützung. In Oracle gibt es beispielsweise den Zahlengenerator **sequence**, wodurch fortlaufende eindeutige Identifikatoren generiert werden können.

4.4 Schemaveränderung

Sollte uns im nachhinein einfallen, dass zu den Professoren auch die Raumnummern ihrer Büros gespeichert werden müssen, kann das Attribut noch mit

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

hinzugefügt werden. Eine Beschränkung auf zehn Zeichen für einen Namen ist sicherlich auch nicht sinnvoll. Besser wäre eine Länge von dreißig Zeichen:

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

Diese Spaltenmodifikation berührt nicht die **not null**-Angabe aus der ursprünglichen Definition. Es ist also auch weiterhin nicht möglich, Professoren ohne Namen abzuspeichern.

Im Standard SQL-92 wird der **alter**-Befehl anders verwendet. Hier müssten die beiden Beispiele

```
alter table Professoren
    add column Raum integer;

alter table Professoren
    alter column Name varchar(30);
```

lauten. Zusätzlich besteht noch die Möglichkeit, Spalten mit **drop column** wieder zu entfernen. Diese Möglichkeit bietet Oracle V7 noch nicht. Eine nicht mehr benötigte Tabelle kann mit dem Befehl **drop table**, gefolgt vom Tabellennamen, wieder entfernt werden.

4.5 Elementare Datenmanipulation: Einfügen von Tupeln

Um die gerade angelegten Tabellen mit Daten zu füllen, fehlt noch ein Befehl zum Einfügen von Tupeln:

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

In den Klammern werden die Werte der Attribute in der Reihenfolge der Definition eingegeben, entsprechend der normalen Tupelschreibweise. Der **insert**-Befehl hat noch vielfältigere Möglichkeiten, doch dazu muss zunächst das Aussehen von SQL-Anfragen untersucht werden.

4.6 Einfache SQL-Anfragen

Zur Demonstration sollen zuerst die Personalnummern und Namen aller C4-Professoren aus unserer Beispieldatenbasis herausgesucht werden. In SQL ist diese Anfrage aus drei Teilen aufgebaut:

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

Zunächst wird im **select**-Teil bestimmt, welche Spalten (Attribute) im Ergebnis ausgegeben werden sollen. In diesem Fall suchen wir die Namen und die Personalnummern der entsprechenden Professoren. Der **from**-Teil gibt die für die Berechnung des Ergebnisses benötigten Tabellen an. In diesem Fall ist es nur die eine Tabelle *Professoren*. Schließlich kann noch ein Kriterium (Selektionsprädikat) angegeben werden, das jede ausgegebene Zeile erfüllen muss. Für unsere Beispielausprägung aus Abbildung 3.8 lautet das Ergebnis der Anfrage wie folgt:

PersNr	Name
2125	Sokrates
2126	Russel
2136	Curie
2137	Kant

Die Stärke von SQL liegt in der Tatsache, dass es sehr nahe an einer natürlichsprachlichen Formulierung eines Befehls liegt. Um eine SQL-Anfrage zu verstehen, reicht es fast aus, sie aus dem Englischen zu übersetzen: „Wähle Personalnummer und Name der Professoren, deren Rang gleich „C4" ist."

In der obigen Beispielanfrage wurden die Namen und Personalnummern der Professoren in einer willkürlichen Reihenfolge ausgegeben. Es ist aber auch möglich, explizit Attribute anzugeben, nach denen sortiert werden soll, und eine Sortierreihenfolge festzulegen. Die möglichen Sortierreihenfolgen sind **asc** (engl. ascending, aufsteigend) und **desc** (engl. descending, absteigend). Fehlt die Angabe, wird implizit **asc** angenommen. Zur Demonstration stellen wir die Anfrage: „Wähle Personalnummer, Name und Rang aller Professoren; sortiere absteigend nach Rang und aufsteigend nach Namen."

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

PersNr	Name	Rang
2136	Curie	C4
2137	Kant	C4
2126	Russel	C4
2125	Sokrates	C4
2134	Augustinus	C3
2127	Kopernikus	C3
2133	Popper	C3

Im obigen Beispiel ist *Rang* das Hauptkriterium, nach dem absteigend sortiert wird. *Name* ist das Nebenkriterium, nach dem aufsteigend sortiert wird.

Da die Eliminierung von Duplikaten in einer Tabelle aus Effizienzgründen nicht automatisch vorgenommen wird, gibt es für diesen Zweck das Schlüsselwort **distinct**. Um also beispielsweise herauszufinden, welche unterschiedlichen Ränge es bei den Professoren gibt, kann man folgende Anfrage stellen:

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

Rang
C3
C4

4.7 Anfragen über mehrere Relationen

Bisher haben wir bei unseren Anfragen immer nur eine Relation betrachtet. Um aber festzustellen, wer die Vorlesung mit dem Titel „Mäeutik" liest, müssen die Tabellen *Professoren* und *Vorlesungen* miteinander verbunden werden:

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

Diese Anfrage könnte man so übersetzen: „Wähle Name und Titel aus den Kombinationen von Professoren und Vorlesungen, bei denen der Wert von *gelesenVon* mit *PersNr* übereinstimmt und der Titel der Vorlesung „Mäeutik" ist."
Die Abarbeitung der Anfrage kann man sich in drei Schritten vorstellen:

1. Zunächst wird das Kreuzprodukt der beteiligten Tabellen gebildet.
2. Jede Zeile dieses Kreuzprodukts wird auf die Erfüllung der Bedingung aus dem **where**-Teil überprüft, die passenden Zeilen werden ausgewählt.
3. Zuletzt wird die Projektion auf die im **select**-Teil angegebenen Attribute durchgeführt.

Das ist in Abbildung 4.1 am Beispiel vorgeführt. Es sollte betont werden, dass mit dieser Auswertungsstrategie nur die Semantik einer SQL-Anfrage demonstriert wird. Die tatsächlich vom DBMS durchgeführte Auswertung ist im Allgemeinen wesentlich effizienter und wird vom Anfrageoptimierer – siehe Kapitel 8 – festgelegt.

Ein äquivalenter Relationenalgebra-Ausdruck sieht folgendermaßen aus:

$$\Pi_{\text{Name, Titel}}(\sigma_{\text{PersNr=gelesenVon} \wedge \text{Titel='Mäeutik'}}(\text{Professoren} \times \text{Vorlesungen}))$$

Allgemein hat eine SQL-Anfrage die Form:

select $A_1, \ldots, A_n$
from $R_1, \ldots, R_k$
where P;

Das Ergebnis des **from**-Teils entspricht logisch dem kartesischen Produkt $R_1 \times \ldots \times R_k$ der beteiligten Relationen. Der **where**-Teil entspricht der Selektion der relationalen Algebra. Er kann auch fehlen, dann wird als Bedingung implizit „true" eingesetzt und jedes Tupel des Kreuzproduktes in das Ergebnis aufgenommen. Der **select**-Teil projiziert schließlich auf die angegebenen Attribute $A_1, \ldots, A_n$. Die in SQL verwendete **select**-Klausel ist also nicht mit der Selektion der relationalen Algebra zu verwechseln; vielmehr entspricht sie der Projektion. Werden alle Attribute benötigt, kann zur Abkürzung einfach „*" anstelle der Attributnamen angegeben werden. Für den allgemeinen Fall ergibt sich also

$$\Pi_{A_1,\ldots,A_n}(\sigma_P(R_1 \times \ldots \times R_k))$$

Mittlerweile ist auch eine direkte Darstellung verschiedener Joinarten der relationalen Algebra in SQL möglich.[2] Bei einem natürlichen Join werden Tabellen anhand gleicher Werte in Spalten mit gleichem Attributnamen miteinander verbunden. Daher muss eine Möglichkeit bestehen, Attributnamen einer Relation zuzuordnen, um Mehrdeutigkeiten zu vermeiden. Möchte man feststellen, welche Studenten welche Vorlesungen hören, benutzt man daher folgende Joinbedingung:

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

Eine zweite Möglichkeit bilden *Tupelvariablen*, die einer Relation zugeordnet werden. Sie sind in diesem Beispiel noch nicht unbedingt notwendig, werden aber später eine Rolle spielen, wenn wir dieselbe Relation mehrfach in einer Anfrage verwenden müssen.

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

[2] Ab SQL-92 können im **from**-Teil auch Jointypen, wie **join** oder **outer join**, angegeben werden. Wir werden später (Abschnitt 4.15) darauf eingehen, obwohl diese Möglichkeit von vielen SQL-Programmierern „verweigert" wird.

Professoren			
PersNr	Name	Rang	Raum
2125	Sokrates	C4	226
2126	Russel	C4	232
⋮	⋮	⋮	⋮
2137	Kant	C4	7

Vorlesungen			
VorlNr	Titel	SWS	gelesenVon
5001	Grundzüge	4	2137
5041	Ethik	4	2125
⋮	⋮	⋮	⋮
5049	Mäeutik	2	2125
⋮	⋮	⋮	⋮
4630	Die 3 Kritiken	4	2137

↘ Verknüpfung (×) ↙

PersNr	Name	Rang	Raum	VorlNr	Titel	SWS	gelesenVon
2125	Sokrates	C4	226	5001	Grundzüge	4	2137
2125	Sokrates	C4	226	5041	Ethik	4	2125
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2125	Sokrates	C4	226	5049	Mäeutik	2	2125
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2126	Russel	C4	232	5001	Grundzüge	4	2137
2126	Russel	C4	232	5041	Ethik	4	2125
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
2137	Kant	C4	7	4630	Die 3 Kritiken	4	2137

↓ Auswahl (σ)

PersNr	Name	Rang	Raum	VorlNr	Titel	SWS	gelesenVon
2125	Sokrates	C4	226	5049	Mäeutik	2	2125

↓ Projektion (Π)

Name	Titel
Sokrates	Mäeutik

Abbildung 4.1: Ausführung einer Anfrage über mehrere Relationen

An dieser Anfrage erkennt man sehr gut die Verwandtschaft von SQL zum relationalen Tupelkalkül (siehe Abschnitt 3.5). In beiden Fällen werden Variablen an Tupel einer Relation gebunden.

4.8 Aggregatfunktionen und Gruppierung

Aggregatfunktionen führen Operationen auf Tupelmengen durch und komprimieren eine Menge von Werten zu einem einzelnen Wert. Zu ihnen gehören **avg** zur Bestimmung des Durchschnitts einer Menge von Zahlen, **max** und **min** zur Bestimmung des größten bzw. kleinsten Elementes und **sum** zur Bildung der Summe. **count** zählt die Anzahl der Zeilen in einer Tabelle.

Die durchschnittliche Semesterzahl aller Studierenden lässt sich mit

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

bestimmen. Besonders nützlich sind Aggregatfunktionen im Zusammenhang mit der Gruppierung durch **group by**. Nehmen wir an, es soll die Anzahl der Semesterwochenstunden herausgefunden werden, die von den einzelnen Professoren erbracht wird:

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

Hier werden alle Zeilen der Tabelle, die den gleichen Wert im angegebenen Attribut *gelesenVon* haben, zusammengefasst und für jede der so entstandenen Gruppen die Summe der SWS berechnet. Man beachte, dass die Gesamtstunden nur für die Professoren berechnet werden, die mindestens eine Vorlesung halten. Die Erweiterung auf alle Professoren wird in Übungsaufgabe 4.8 behandelt.

Sollen bei der Zählung nur die Professoren berücksichtigt werden, die überwiegend lange Vorlesungen halten, kann an die durch **group by** gebildeten Gruppen noch mit **having** eine zusätzliche Bedingung gestellt werden. In der folgenden Anfrage werden also zunächst alle Vorlesungen nach Professoren gruppiert und in jeder Gruppe der Durchschnitt der Semesterwochenstunden gebildet:

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

Um den Unterschied zwischen **where** und **having** zu verdeutlichen, erweitern wir die Anfrage um eine **where**-Bedingung. Es sollen nur C4-Professoren berücksichtigt werden. Zusätzlich sollen ihre Namen ausgegeben werden. Die entsprechende Anfrage lautet:

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

Eine mögliche Abarbeitung der Anfrage ist in Abbildung 4.2 dargestellt. Zuerst werden diejenigen Tupel aus der temporären Relation *Vorlesungen*×*Professoren* ausgewählt, die die **where**-Bedingung erfüllen. Anschließend findet die Gruppierung statt: Tupel mit gleichem Wert in den Gruppierungsattributen werden zusammen angeordnet. Jetzt wird die **having**-Bedingung für jede Gruppe überprüft. Dafür ist in diesem Fall die Berechnung des Durchschnitts der Semesterwochenstunden pro Gruppe notwendig. Im letzten Schritt werden aus den Gruppen die Ergebnistupel gebildet, hier unter Bildung der Summe der Semesterwochenstunden. Da in der Ausgaberelation jede Gruppe nur durch ein einziges Tupel repräsentiert wird, können in der **select**-Klausel nur Aggregatfunktionen vorkommen oder Attribute, nach denen gruppiert wurde, d.h. die auch in der **group by**-Klausel verwendet wurden. Aus diesem Grund musste in der Beispielanfrage das Attribut *Name* mit in die **group by**-Klausel übernommen werden.

4.9 Geschachtelte Anfragen

In SQL können **select**-Anweisungen auf vielfältige Weisen verknüpft und geschachtelt werden. Dabei werden Anfragen, die höchstens ein Tupel zurückliefern, von denen unterschieden, die beliebig viele Tupel ergeben. Wenn eine Unteranfrage nur ein Tupel mit nur einem Attribut zurückliefert, so kann diese Unteranfrage dort eingesetzt werden, wo ein skalarer Wert gefordert wird. Insbesondere geht dies in der **select**-Klausel und bei Vergleichen in der **where**-Klausel.[3] So könnte man beispielsweise alle Prüfungen suchen, die genau durchschnittlich verlaufen sind:

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

Das Symbol * in der **select**-Klausel gibt, wie bereits gesagt, an, dass alle Attribute der in der **from**-Klausel aufgeführten Relation(en) ausgegeben werden sollen. Die SQL-Syntax schreibt vor, dass Unteranfragen immer geklammert werden – egal wo sie stehen.

Um das Beispiel aus Abschnitt 4.8 nochmals aufzugreifen, wollen wir die Lehrbelastung der Professoren ermitteln:

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

Die beiden obigen Anfragen unterscheiden sich in einem interessanten Gesichtspunkt. Im ersten Beispiel verwendet die Unteranfrage lediglich ihre „eigenen" Attribute. Im zweiten Beispiel hingegen bezieht sich die Unteranfrage auf das Attribut

[3] Das Operieren mit mehrstelligen Tupeln ist eine SQL-92 Erweiterung und noch nicht in allen Produkten implementiert. Das gleiche gilt für Unteranfragen in der **select**-Klausel.

Vorlesungen × Professoren							
VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5001	Grundzüge	4	2137	2125	Sokrates	C4	226
5041	Ethik	4	2125	2125	Sokrates	C4	226
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ **where**-Bedingung

VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5001	Grundzüge	4	2137	2137	Kant	C4	7
5041	Ethik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnistheorie	3	2126	2126	Russel	C4	232
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5052	Wissenschaftstheorie	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ Gruppierung

VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5043	Erkenntnistheorie	3	2126	2126	Russel	C4	232
5052	Wissenschaftstheorie	3	2126	2126	Russel	C4	232
5216	Bioethik	2	2126	2126	Russel	C4	232
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ **having**-Bedingung

VorlNr	Titel	SWS	gelesenVon	PersNr	Name	Rang	Raum
5041	Ethik	4	2125	2125	Sokrates	C4	226
5049	Mäeutik	2	2125	2125	Sokrates	C4	226
4052	Logik	4	2125	2125	Sokrates	C4	226
5001	Grundzüge	4	2137	2137	Kant	C4	7
4630	Die 3 Kritiken	4	2137	2137	Kant	C4	7

⇓ Aggregation (**sum**) und Projektion

gelesenVon	Name	**sum**(SWS)
2125	Sokrates	10
2137	Kant	8

Abbildung 4.2: Ausführung einer Anfrage mit **group by**

PersNr der Tupel der äußeren Anfrage. Die Unteranfrage ist mit der äußeren Anfrage *korreliert*.

Eine nicht-korrelierte Unteranfrage braucht nur einmal ausgewertet zu werden: das Ergebnis ist während der Auswertung der äußeren Anfrage konstant. Korrelierte Anfragen werden jedoch im Allgemeinen für jedes Tupel der umgebenden Anfrage neu berechnet (d.h. für jedes zu überprüfende Tupel, falls die Unteranfrage in der **where**-Bedingung steht bzw. für jedes auszugebende Tupel, falls sie sich in der **select**-Klausel befindet). In dieser Hinsicht legt SQL eine „nested-loops"-Semantik fest, da für jedes Tupel der übergeordneten Anfrage die Unteranfrage auszuwerten ist – wobei Unteranfragen selbst wieder Unteranfragen enthalten können (also beliebige Schachtelungstiefe).

Zur Verdeutlichung des Unterschieds zwischen korrelierten und unkorrelierten Unteranfragen wollen wir noch ein paar Beispielanfragen formulieren. Zu diesem Zweck nehmen wir an, dass die Relationen *Studenten*, *Assistenten* und *Professoren* ein weiteres Attribut *GebDatum* vom Typ **date** enthalten. Folgende Anfrage liefert alle Studenten, die älter als der jüngste Professor bzw. die jüngste Professorin sind:

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

Der **exists**-Operator liefert *true* falls die Unteranfrage mindestens ein Ergebnistupel zurückliefert; sonst *false* – siehe Abschnitt 4.12. Diese Anfrage mit korrelierter Unteranfrage lässt sich leicht in eine äquivalente Anfrage mit unkorrelierter Unteranfrage umformulieren, indem wir mit der Aggregatfunktion **max** das Geburtsdatum des jüngsten Professors bzw. der jüngsten Professorin ermitteln:

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

Die Anfrageauswertungskomponente des DBMS wird diese unkorrelierte Unteranfrage (hoffentlich!) nur einmal auswerten und dann den einen Wert für die Auswertung der übergeordneten Anfrage verwenden. Deshalb ist diese zweite Formulierung natürlich viel effizienter zu bearbeiten. Es ist wünschenswert, dass der Anfrageoptimierer eines DBMS automatisch die günstigste Auswertung für eine gegebene Anfrage ermittelt. Leider sind heutige Anfrageoptimierer davon noch weit entfernt, so dass Datenbankanwender durchaus auf manuelle Umformulierungen von Anfragen angewiesen sind, um die Leistungsfähigkeit ihrer Anwendungen zu steigern.

Nicht immer ist es so einfach, eine korrelierte Unteranfrage in eine unkorrelierte Unteranfrage umzuwandeln. Manchmal kann man durch die Einführung eines Joins eine sogenannte Entschachtelung einer korrelierten Unteranfrage erzielen – wie folgendes Beispiel demonstriert:

```
select a.*
from Assistenten a
where exists
      ( select p.*
        from Professoren p
        where a.Boss = p.PersNr and p.GebDatum > a.GebDatum );
```

Hier werden also die Assistenten ermittelt, die für einen jüngeren Professor bzw. eine jüngere Professorin arbeiten. Die oben verwendete Methode zur „Dekorrelierung" der Unteranfrage ist hier wegen des zusätzlichen Prädikats $a.Boss = p.PersNr$ nicht anwendbar. Man kann diese geschachtelte Anfrage aber in eine äquivalente ungeschachtelte Joinanfrage überführen:

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

Wir überlassen es den Lesern, einige allgemeine Umformungsregeln herzuleiten, um korrelierte Unteranfragen in einer **where**-Klausel zu entschachteln.

Mit dem **exists**-Operator wird eine Unteranfrage, die möglicherweise mehrere Tupel zurückliefert, auf einen atomaren Wert (*true* oder *false*) abgebildet. Es ist aber auch möglich, die von einer Unteranfrage zurückgelieferten Tupel als Kollektion zu „verwerten". Solche Unteranfragen können als Argument einer Mengenoperation oder in der Liste der Relationen im **from**-Teil einer Anfrage auftreten. Im nächsten Beispiel verwenden wir eine in die **from**-Klausel geschachtelte Unteranfrage, um eine komplexere Anfrage modular aufbauen zu können. In der Unteranfrage wird der Join der beiden Relationen *Studenten* und *hören* sowie deren Gruppierung nach *MatrNr* und *Name* durchgeführt. Das Ergebnis der Unteranfrage ist eine dreistellige temporäre Relation, die wir mit *tmp* benennen, mit den Attributen *MatrNr*, *Name* und *VorlAnzahl*. Das letztere Attribut wurde durch eine **count**-Aggregation in der Unteranfrage gebildet. Man beachte, dass man nun in der Lage ist, in der übergeordneten Anfrage das Attribut *VorlAnzahl* in der **where**-Klausel zu verwenden. Ohne die Schachtelung hätte man die Einschränkung auf fleißige Studenten nur über eine **having**-Klausel formulieren können.

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

Als Anfrageergebnis erhält man für unsere Universitätsdatenbank folgende Tabelle:

MatrNr	Name	VorlAnzahl
28106	Carnap	4
29120	Theophrastos	3

Wir wollen noch eine Anfrage formulieren, die in ähnlicher Form in sogenannten Decision Support-Anwendungen häufig vorkommt: Ermittle den Marktanteil der einzelnen Vorlesungen als den Prozentsatz der Studenten, die die Vorlesung hören.

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

In einer „natürlicheren" Formulierung hätte man vielleicht den Einzelwert *GesamtAnz* durch eine geschachtelte Anfrage in der **select**-Klausel bestimmt. Wir haben bei der obigen Formulierung aber bewusst auf die Schachtelung von Anfragen in der **select**-Klausel verzichtet, da einige Systeme dies noch nicht unterstützen.

Als Ergebnis erhält man für unsere Universitätsdatenbank folgende Tabelle:

VorlNr	AnzProVorl	GesamtAnz	Marktanteil
4052	1	8	.125
5001	4	8	.5
5022	2	8	.25
...	...	...	...

Manche „pedantische" Datenbanksysteme würden allerdings anstatt einer Dezimalzahl für den Marktanteil eine „abgeschnittene" oder gerundete Integer-Zahl zurückgeben, da beide Argumente der Division (*GesamtAnz* und *AnzProVorl*) vom Typ Integer sind. Um dennoch die gewünschte Genauigkeit zu erzielen, muss man dazu mindestens einen der beiden Operanden in eine Dezimalzahl umwandeln, was man in SQL mit der **cast**-Klausel wie folgt erzielt:

cast(h.AnzProVorl as **decimal**(8,2))/g.GesamtAnz

Etwas trickreicher, dafür aber kürzer, wäre eine Formulierung, bei der man zuerst *AnzProVorl* durch eine „Dummy"-Multiplikation mit 1.00 in eine Dezimalzahl umwandelt:

cast(h.AnzProVorl * 1.00)/g.GesamtAnz

4.10 Modularisierung von SQL-Anfragen

Zugegebenermaßen ist die obige Anfrage zur Ermittlung des Marktanteils der einzelnen Vorlesungen relativ schwer zu lesen, da zwei **select** ... **from** ... **where** ...-Anfragen in der äußeren **from**-Klausel geschachtelt wurden. De facto wurden hierdurch zwei temporäre Zwischenergebnis-Relationen definiert: die Relation h mit den Attributen *VorlNr* und *AnzProVorl* sowie die Relation g mit dem Attribut *GesamtAnz*. Zum Glück kann man in SQL die Spezifikation derartiger Zwischenergebnisse aus der eigentlichen SQL-Anfrage „herausziehen" um die Anfrage zu modularisieren. Dies geht mit dem **with**-Konstrukt, das in der nachfolgenden Formulierung der Anfrage verwendet wird:

```
with h as (
        select VorlNr, count(*) as AnzProVorl
        from hören
        group by VorlNr ),
     g as ( select count(*) as GesamtAnz from Studenten )

select h.VorlNr, h.AnzProVorl, g.GesamtAnz,
       cast(h.AnzProVorl as decimal(8,2))/g.GesamtAnz as Marktanteil
from h , g
```

In der **with**-Klausel werden hier zwei temporäre Sichten namens h und g – für die Dauer der Anfragebearbeitung – definiert. Man kann h und g wie „normale" Relationen auffassen, die genau die Daten enthalten, die sich aus der zugeordneten Anfrage als Ergebnis ergeben.

4.11 Mengen-Operatoren

Die klassischen Operationen der Mengenlehre, Vereinigung, Durchschnitt und Differenz, heißen in SQL **union**, **intersect** und **except**.[1] Damit ist es z.B. möglich, die Namen aller Angestellten zu bestimmen, also die aller Professoren und aller Assistenten:

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

Da das Ergebnis einer Anfrage wieder eine sinnvolle Tabelle darstellen soll, müssen die Ergebnistypen der Teilanfragen übereinstimmen. In unserem Fall liefern beide Teilanfragen jeweils eine Tabelle zurück, deren einzige Spalte Zeichenketten enthält. Es ist nicht möglich, eine Tabelle aus Zeichenketten beispielsweise mit einer Tabelle aus Zahlen zu vereinigen.

Die **union**-Operation führt automatisch eine Duplikateliminierung durch, die durch Einsatz von **union all** allerdings „abgestellt" werden kann.

Der Operator **in** testet auf Mengenmitgliedschaft. Sollen die Professoren gefunden werden, die sich nicht an der Lehre beteiligen, kann mit **not in** getestet werden, welche Personalnummern im *gelesenVon*-Attribut der Relation *Vorlesungen* enthalten sind:

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

[1] **except** heißt in Oracle **minus**.

In vielen Fällen lässt sich eine geschachtelte Anfrage mit **in** durch eine gleichwertige, nichtgeschachtelte Anfrage mit einem Join ersetzen (siehe Übungsaufgabe 4.2).

in ist äquivalent zur *quantifizierenden Bedingung* = **any**. Eine quantifizierende Bedingung besteht aus einem der Vergleichsoperatoren (=, <, >, ...) und **all** oder **any**.[5]

any testet, ob es mindestens ein Element im Ergebnis der Unteranfrage gibt, für das der Vergleich mit dem linken Argument des Operators erfüllt wird. **all** überprüft, ob alle Ergebnisse der Unteranfrage den Vergleich erfüllen. Die Studenten mit der größten Semesterzahl können mit

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

herausgefunden werden. Effizienter wäre hier natürlich eine Formulierung mit der **max**-Aggregation in der Unteranfrage. Wie und warum?

all sollte nicht mit dem Allquantor ($\forall$) verwechselt werden, der in SQL nicht vorhanden ist. **all** führt lediglich einen Vergleich eines Werts mit einer Menge durch. Auf diese Weise ist es nicht möglich, eine Anfrage wie „Finde die Studenten, die alle vierstündigen Vorlesungen hören" zu formulieren.

4.12 Quantifizierte Anfragen in SQL

Der Existenzquantor ($\exists$) wird in SQL durch **exists** realisiert. **exists** überprüft, wie oben bereits ausgeführt, ob die von einer Unteranfrage bestimmte Menge von Tupeln leer ist. Bei einer leeren Menge liefert **exists** den Wahrheitswert **false** und sonst **true**. Bei dem Operator **not exists** ist es natürlich umgekehrt.

Die Frage nach den Professoren, die keine Vorlesungen halten, kann man mit **not exists** wie folgt formulieren:

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

In SQL gibt es, wie oben schon erwähnt, keinen expliziten Allquantor, so dass Anfragen mit einer logischen Allquantifizierung auch durch den Existenzquantor ausgedrückt werden müssen. Wir wollen dies an der Beispielanfrage aus Kapitel 3.5.2 demonstrieren:

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

[5] Oder, alternativ zu **any**, **some**

In dieser Tupelkalkül-Formel werden also die Studenten ermittelt, die *alle* vierstündigen Vorlesungen hören. Obwohl SQL den relationalen Tupelkalkül als Grundlage hat, lässt sich diese Formel erst in SQL umsetzen, nachdem man den Allquantor ($\forall$) und den Implikationsoperator ($\Rightarrow$) gemäß folgender Äquivalenzen eliminiert hat:

$$\begin{aligned} \forall t \in R(P(t)) &= \neg(\exists t \in R(\neg P(t))) \\ R \Rightarrow T &= \neg R \vee T \end{aligned}$$

Schrittweise ermittelt man also folgende äquivalente Formeln:

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

Durch Anwendung der DeMorgan-Gesetze kann man die Negationen „nach innen ziehen":

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

Diese letzte Formel kann man jetzt sehr einfach in SQL-Syntax überführen und erhält folgende geschachtelte Anfrage:

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

Viele kommerzielle Datenbanksysteme haben Schwierigkeiten, derartig tief geschachtelte Anfragen effizient auszuwerten. Deshalb ist es oft viel effizienter, die Allquantifizierung durch Zählen von Tupeln in SQL auszudrücken. Wir betrachten dazu eine etwas einfachere Anfrage, in der wir die (*MatrNr* der) Studenten ermitteln wollen, die *alle* Vorlesungen hören:

```
select h.MatrNr
from hören h
group by h.MatrNr
   having count(*) = (select count(*) from Vorlesungen);
```

Es wird also gezählt, wieviele Vorlesungen die einzelnen Studenten hören und überprüft, ob diese Anzahl mit der Anzahl der in der Relation *Vorlesungen* gespeicherten Tupel übereinstimmt. Für die Korrektheit dieser Formulierung muss verlangt werden, dass alle *VorlNr*-Werte in der Relation *hören* gültig sind; d.h. für jeden

VorlNr-Wert aus *hören* muss es eine Vorlesung mit diesem *VorlNr*-Schlüsselwert in der Relation *Vorlesungen* geben. Mit anderen Worten muss die referentielle Integrität gewährleistet sein – siehe dazu Kapitel 5. Weiterhin darf die Relation *hören* keine Duplikate enthalten. Wir überlassen es den Lesern in Übungsaufgabe 4.5 eine Formulierung zu finden, die auch bei einer möglichen Verletzung der referentiellen Integrität das korrekte Ergebnis liefert. In Aufgabe 4.6 werden die Leser herausgefordert, die schwierigere Anfrage, in der die Studenten ermittelt werden, die alle vierstündigen Vorlesungen hören, mittels einer **count**-Aggregierung zu formulieren.

4.13 Nullwerte

In SQL gibt es einen speziellen Wert mit dem Namen **null**, der in jedem Datentyp vorhanden ist. Ein **null**-Wert wird z.B. dann als Attributwert gespeichert, wenn der korrekte Wert nicht bekannt ist. Beispielsweise würde man **null** als Wert für das Attribut *gelesenVon* eines Tupels der Relation *Vorlesungen* eintragen, wenn für die zugehörige Vorlesung noch kein Referent bzw. keine Referentin gefunden wurde.

Bei der Anfragebearbeitung können Nullwerte als Ergebnis von Operationen entstehen – selbst wenn die zugrunde liegenden Relationen keine **null**-Werte enthalten. Ein Beispiel sind die äußeren Joins, die in den Abschnitten 3.4.8 und 4.15 behandelt sind. Ein anderes Beispiel ist die Anwendung einer Aggregatfunktion (wie **max**) auf eine leere Tabelle.

Das Ergebnis von Anfragen bei vorliegenden **null**-Werten ist oftmals überraschend. Nehmen wir folgendes Beispiel:

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

Wenn es Studenten gibt, deren *Semester*-Attribut den Wert **null** hat, werden diese nicht mitgezählt. Der Grund liegt in folgenden Regeln für den Umgang mit **null**-Werten begründet:

1. In arithmetischen Ausdrücken werden Nullwerte propagiert, d.h. sobald ein Operand **null** ist, wird auch das Ergebnis **null**. Dementsprechend wird z.B. **null** + 1 zu **null** ausgewertet – aber auch **null** ∗ 0 wird zu **null** ausgewertet.

2. SQL hat eine dreiwertige Logik, die nicht nur **true** und **false** kennt, sondern auch einen dritten Wert **unknown**. Diesen Wert liefern Vergleichsoperationen zurück, wenn mindestens eines ihrer Argumente **null** ist. Beispielsweise wertet SQL das Prädikat (*PersNr* = ...) immer zu **unknown** aus, wenn die *PersNr* des betreffenden Tupels den Wert **null** hat.

3. Logische Ausdrücke werden nach den folgenden Tabellen berechnet:

not	
true	false
unknown	unknown
false	true

and	true	unknown	false
true	true	unknown	false
unknown	unknown	unknown	false
false	false	false	false

or	true	unknown	false
true	true	true	true
unknown	true	unknown	unknown
false	true	unknown	false

Diese Berechnungsvorschriften sind recht intuitiv. **unknown or true** wird z.B. zu **true** – die Disjunktion ist mit dem **true**-Wert des rechten Arguments immer erfüllt, unabhängig von der Belegung des linken Arguments. Analog ist **unknown and false** automatisch **false** – keine Belegung des linken Arguments könnte die Konjunktion mehr erfüllen.

4. In einer **where**-Bedingung werden nur Tupel weitergereicht, für die die Bedingung **true** ist. Insbesondere werden Tupel, für die die Bedingung zu **unknown** auswertet, nicht ins Ergebnis aufgenommen.

5. Bei einer Gruppierung wird **null** als ein eigenständiger Wert aufgefasst und in eine eigene Gruppe eingeordnet.

Betrachten wir jetzt nochmals die Beispielanfrage. Für Studenten mit einem **null**-Wert für das Attribut *Semester* evaluieren beide Terme der Disjunktion, „*Semester* < 13" und „*Semester* >= 13", zu **unknown**. Gemäß obiger Wertetabelle evaluiert **unknown or unknown** zu **unknown**. Somit kann sich das entsprechende Tupel nicht für das Anfrageergebnis qualifizieren, da nur Tupel in die Ergebnismenge aufgenommen werden, deren Anfrageprädikat zu **true** evaluiert.

Das Beispiel demonstriert, dass man soweit möglich auf **null**-Werte verzichten sollte (z.B. durch entsprechende Integritätsbedingungen oder einer Normalisierung, wie in den nächsten beiden Kapiteln behandelt) oder aber deren Existenz bei der Anfrageformulierung berücksichtigen muss. Kann ein Ausdruck zu **null** auswerten, lässt sich das mit der Bedingung **is null** bzw. **is not null** überprüfen. Logische Ausdrücke lassen sich mit **is unknown** bzw. **is not unknown** testen.[6]

4.14 Spezielle Sprachkonstrukte

Es gibt noch einige weitere Bedingungen, die im **where**-Teil benutzt werden können. Zwei davon wollen wir hier untersuchen.

Die erste, **between**, ist nichts anderes als eine Abkürzung. Häufig möchte man nur einen bestimmten Wertebereich testen, z.B. Semesterzahlen zwischen eins und vier. In diesem Fall sind die folgenden beiden Bedingungen gleichwertig:

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

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

[6] Dies ist jedoch SQL-92 und noch nicht überall verfügbar.

Für kleine diskrete Bereiche ist auch die explizite Angabe einer Menge möglich:

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

Sehr nützlich ist der Vergleich von Zeichenketten auf Ähnlichkeit mit **like**. Wenn eine Zeichenkette nicht genau bekannt ist, können „%" und „_" als Platzhalter für unbekannte Teile verwendet werden. „%" steht dabei für beliebig viele und „_" für genau ein unbekanntes Zeichen. Sucht man beispielsweise die Matrikelnummer von Theophrastos, weiß aber nicht mehr genau, ob er mit „h" geschrieben wurde, kann man die „verdächtigen" Stellen durch ein „%" ersetzen:

```
select *
from Studenten
where Name like 'T%eophrastos';
```

Als weiteres Beispiel sollen die Studenten gesucht werden, die mindestens eine Vorlesung über Ethik gehört haben:

```
select distinct s.Name
from Vorlesungen v, hören h, Studenten s
where s.MatrNr = h.MatrNr and h.VorlNr = v.VorlNr and
      v.Titel like '%thik%';
```

Hier wurde bewusst das „E" in Ethik nicht angegeben, da es in zusammengesetzten Wörtern klein geschrieben wird (wie z.B. in „Bioethik").

In SQL-92 ist für die „Dekodierung" von Attributwerten das **case**-Konstrukt vorgesehen. Wir wollen dies an einem einfachen Beispiel vorführen: Wir wollen die Prüfungsnoten, die in der Relation *prüfen* in numerischer Form abgespeichert sind, in entsprechende Prädikatsnoten umwandeln:

```
select MatrNr, ( case when Note < 1.5 then 'sehr gut'
                      when Note < 2.5 then 'gut'
                      when Note < 3.5 then 'befriedigend'
                      when Note <= 4.0 then 'ausreichend'
                      else 'nicht bestanden' end )
from prüfen;
```

Man beachte, dass die Alternativen (**when**-Klauseln) in der Reihenfolge ihres Auftretens evaluiert werden. Die erste Bedingung, die zu *true* auswertet, bestimmt den einzusetzenden Wert.

4.15 Joins in SQL-92

In SQL-92 wurde eine Möglichkeit zur direkten Angabe eines Join-Operators geschaffen. Dort können im **from**-Teil die folgenden Schlüsselwörter verwendet werden:

- **cross join**: Kreuzprodukt,

- **natural join**: natürlicher Join,
- **join** oder **inner join**: Theta-Join und
- **left**, **right** oder **full outer join**: äußerer Join.

Auf diese Weise kann z.B. eine Anfrage der Art

select $*$
from R_1, R_2
where $R_1.A = R_2.B$;

explizit als Join formuliert werden:

select $*$
from R_1 **join** R_2 **on** $R_1.A = R_2.B$;

Hinter dem Join wird mit **on** die Joinbedingung explizit angegeben.

Zu den beiden letzteren Jointypen wollen wir noch je ein Beispiel betrachten. Der äußere Join wurde schon in Abschnitt 3.4.8 als Operator der relationalen Algebra besprochen. Er erhält, je nachdem ob es sich um einen **left**, **right** oder **full outer join** handelt, auch die Zeilen respektive der linken, rechten oder beider Relationen, die nicht die Joinbedingung erfüllen. In Abbildung 4.3 wird das anhand von *Professoren*, *prüfen* und *Studenten* verdeutlicht.

4.16 Rekursion

In der Beispiel-Datenbank aus Abbildung 3.8 wurde mit Hilfe der rekursiven Beziehung *voraussetzen* einigen Vorlesungen Vorgänger zugeordnet. Grafisch kann man diese Beziehung wie in Abbildung 4.4 darstellen.

Nun möchten wir herausfinden, was man alles hören muss, um die Vorlesung „Der Wiener Kreis" verstehen zu können. Dazu stellen wir zunächst die Anfrage:

select Vorgänger
from voraussetzen, Vorlesungen
where Nachfolger = VorlNr **and**
Titel = 'Der Wiener Kreis';

Als Ergebnis erhält man die direkten Vorgänger der Vorlesung „Der Wiener Kreis" – in unserer Beispielausprägung ist das lediglich die eine Vorlesung „Wissenschaftstheorie". Um festzustellen, welche Vorlesungen für die direkten Vorgänger gebraucht werden, stellt man die Anfrage:

select Vorgänger
from voraussetzen
where Nachfolger **in** (**select** Vorgänger
from voraussetzen, Vorlesungen
where Nachfolger = VorlNr **and**
Titel = 'Der Wiener Kreis');

```
select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr, s.MatrNr, s.Name
from Professoren p left outer join
     (prüfen f left outer join Studenten s on f.MatrNr = s.MatrNr)
     on p.PersNr = f.PersNr;
```

p.PersNr	p.Name	f.PersNr	f.Note	f.MatrNr	s.MatrNr	s.Name
2126	Russel	2126	1	28106	28106	Carnap
2125	Sokrates	2125	2	25403	25403	Jonas
2137	Kant	2137	2	27550	27550	Schopenhauer
2136	Curie	–	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr, s.MatrNr, s.Name
from Professoren p right outer join
     (prüfen f right outer join Studenten s on f.MatrNr = s.MatrNr)
     on p.PersNr = f.PersNr;
```

p.PersNr	p.Name	f.PersNr	f.Note	f.MatrNr	s.MatrNr	s.Name
2126	Russel	2126	1	28106	28106	Carnap
2125	Sokrates	2125	2	25403	25403	Jonas
2137	Kant	2137	2	27550	27550	Schopenhauer
–	–	–	–	–	26120	Fichte
⋮	⋮	⋮	⋮	⋮	⋮	⋮

```
select p.PersNr, p.Name, f.PersNr, f.Note, f.MatrNr, s.MatrNr, s.Name
from Professoren p full outer join
     (prüfen f full outer join Studenten s on f.MatrNr = s.MatrNr)
     on p.PersNr = f.PersNr;
```

p.PersNr	p.Name	f.PersNr	f.Note	f.MatrNr	s.MatrNr	s.Name
2126	Russel	2126	1	28106	28106	Carnap
2125	Sokrates	2125	2	25403	25403	Jonas
2137	Kant	2137	2	27550	27550	Schopenhauer
–	–	–	–	–	26120	Fichte
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2136	Curie	–	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Abbildung 4.3: Äußere Joins in SQL-92

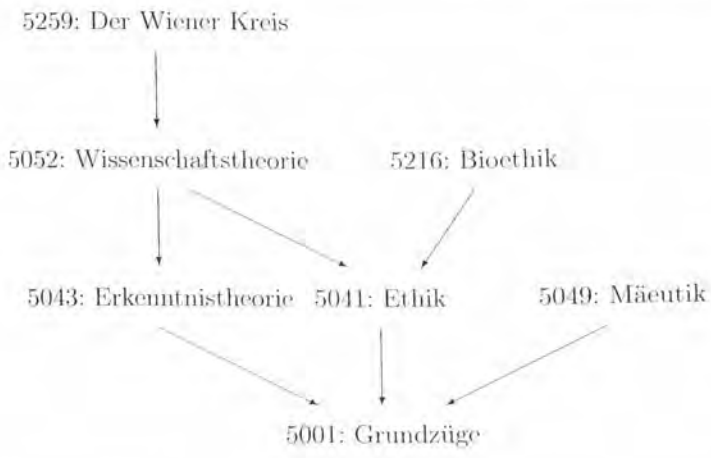


Abbildung 4.4: Grafische Darstellung von *voraussetzen*

Alternativ kann man diese Anfrage mit Hilfe von Tupelvariablen ohne Schachtelung formulieren:

```
select v1.Vorgänger
from voraussetzen v1, voraussetzen v2, Vorlesungen v
where v1.Nachfolger = v2.Vorgänger and
      v2.Nachfolger = v.VorlNr and
      v.Titel = 'Der Wiener Kreis';
```

Aber mit den indirekten Vorgängern erster Stufe ist man noch nicht fertig. Es müssen immer weitere indirekte Vorgänger gesucht werden, bis sich keine weiteren Vorlesungen mehr hinzugesellen. Erst dann ist die Menge aller Voraussetzungen gefunden. Die indirekten Vorgänger n-ter Stufe werden also wie folgt gebildet:

```
select v1.Vorgänger
from voraussetzen v1,
     ⋮
     voraussetzen vn_minus_1
     voraussetzen vn,
     Vorlesungen v
where v1.Nachfolger = v2.Vorgänger and
      ⋮
      vn_minus_1.Nachfolger = vn.Vorgänger and
      vn.Nachfolger = v.VorlNr and
      v.Titel = 'Der Wiener Kreis' ;
```

Das ist sehr umständlich, leider aber im SQL-Standard nicht anders möglich. Es fehlt eine Möglichkeit zur Berechnung von *transitiven Hüllen*. Die transitive Hülle

Abbildung 4.5: Die transitive Hülle der Beziehung *voraussetzen*

einer Relation R mit zwei Attributen A und B gleichen Typs ist definiert als:

$$\begin{aligned} trans_{A,B}(R) = \{(a,b) \mid\ & \exists k \in \mathbb{N}\ (\exists \tau_1, \ldots, \tau_k \in R(\\ & \tau_1.A = \tau_2.B \wedge \\ & \tau_2.A = \tau_3.B \wedge \\ & \vdots \\ & \tau_{k-1}.A = \tau_k.B \wedge \\ & \tau_1.A = a \wedge \\ & \tau_k.B = b))\} \end{aligned}$$

Sie enthält damit alle Tupel (a, b), für die ein Pfad beliebiger Länge k in R existiert. Die transitive Hülle unseres Beispiels ist in Abbildung 4.5 dargestellt.

Damit ist SQL nicht Turing-vollständig. Da sich aber alle Ausdrücke der relationalen Algebra in SQL übertragen lassen und die Ausdruckskraft von relationaler Algebra und Relationenkalkül äquivalent ist, sind auch diese beiden formalen Anfragesprachen nicht Turing-vollständig.

Oracle ermöglicht das Traversieren hierarchischer Beziehungen. Es bietet einen **connect by**-Befehl an, der die Verbindung von Eltern-Objekten zu ihren Kindern angibt.[7] Die folgende Anfrage findet alle Vorgänger der Vorlesung „Der Wiener Kreis".

```
select Titel
from Vorlesungen
where VorlNr in (select Vorgänger
                 from voraussetzen
                 connect by Nachfolger = prior Vorgänger
```

[7] Dieser Befehl ist jedoch, wie gesagt, nicht im SQL-92 Standard enthalten. Rekursive Anfragen wurden erst im SQL-99 Standard in anderer Weise, die wir im Anschluss diskutieren, eingeführt.

```
start with Nachfolger = (select VorlNr
                         from Vorlesungen
                         where Titel = 'Der Wiener Kreis'));
```

Mit **start with** wird der Ausgangspunkt für die Tiefensuche festgelegt, in unserem Fall die Vorlesungsnummer der Vorlesung „Der Wiener Kreis". Die Verbindungsbedingung für **connect by** besagt, dass der *Vorgänger* des Elternknotens mit dem *Nachfolger* des Kindknotens übereinstimmen soll. Attribute des Elternknotens werden durch **prior** markiert. Das Ergebnis lautet:

Titel
Grundzüge
Ethik
Erkenntnistheorie
Wissenschaftstheorie

Die obige Formulierung ist Oracle-spezifisch. Gemäß des SQL-99 Standards könnte man die Anfrage wie folgt formulieren:

```
with recursive TransVorl (Vorg , Nachf)
as ( select Vorgänger, Nachfolger from voraussetzen
     union all
     select t.Vorg, v.Nachfolger
     from TransVorl t, voraussetzen v
     where t.Nachf = v.Vorgänger )

select Titel from Vorlesungen where VorlNr in
    ( select Vorg from TransVorl where Nachf in
        ( select VorlNr from Vorlesungen where Titel = 'Der Wiener Kreis' ) )
```

In der **with**-Klausel wird eine temporäre (rekursive) Sicht namens *TransVorl* für die Dauer der Anfragebearbeitung definiert. Man kann *TransVorl* wie eine „normale" Relation auffassen, die genau die Daten enthält, die sich aus der zugeordneten Anfrage als Ergebnis ergeben. Diese Sicht ist hier rekursiv definiert, da *TransVorl* in der SQL-Definition der Sicht vorkommt. Durch diese Sichtendefinition wird also die transitive Hülle der Relation *voraussetzen* definiert, die dann in der darunter stehenden Anfrage ausgenutzt wird.

Zum Verständnis der Definition der Sicht *TransVorl* trägt – zumindest für Leser mit etwas Erfahrung in logischer Programmierung – die Definition dieser Sicht in der Form von Prolog- oder Datalog-Regeln (siehe auch Kapitel 15) bei:

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

Hierbei gehen wir davon aus, dass die Fakten der Relation *voraussetzen* in der Form *voraussetzen*(5001,5041) und *voraussetzen*(5041,5052) vorgegeben sind. Dann definiert die erste Regel, dass alle direkten *Vorgänger*, *Nachfolger*-Paare der Relation *voraussetzen* in der Sicht *TransVorl* enthalten sind. Die zweite Regel erweitert diese

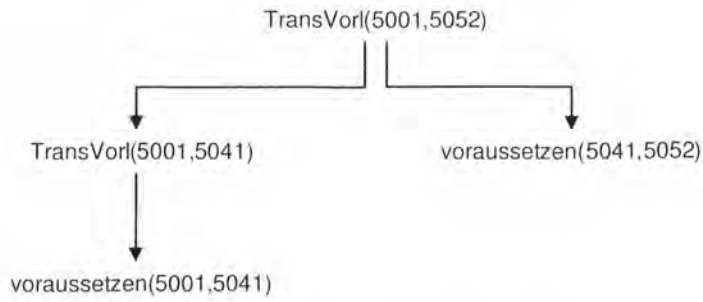


Abbildung 4.6: Herleitung eines Tupels der Sicht TransVorl

Sicht, so dass V ein Vorgänger von N ist, falls V gemäß *TransVorl* ein Vorgänger von Z ist und Z gemäß *voraussetzen* ein direkter Vorgänger von N ist. Gemäß der ersten Regel gilt dann also beispielsweise *TransVorl*(5001,5041). Dann läßt sich mittels der zweiten Regel auch *TransVorl*(5001,5052) herleiten, da *TransVorl*(5001,5041) und *voraussetzen*(5041,5052) gelten. Diese Herleitung geschieht durch folgende Substitution der Variablen: $V \leftarrow 5001, Z \leftarrow 5041, N \leftarrow 5052$. Die obige Herleitung ist in Abbildung 4.6 als Baumstruktur visualisiert. Eine detailliertere Erklärung hierzu findet sich in Kapitel 15.

Diese beiden Regeln zur Definition von *TransVorl* findet man in der SQL-Sichtendefinition – in etwas verboserer Syntax – als die zwei Argumente der Vereinigung **union all** wieder.

Der SQL3- bzw. SQL:1999-Standard enthält die Rekursion in DB2-ähnlicher Form. Trotz dieser Möglichkeiten, rekursive Anfragen stellen zu können, ist SQL dennoch nicht Turing-vollständig – wenn man nur die Kernsprache ohne benutzerdefinierte Funktionen betrachtet. Es wird lediglich ein Spezialfall abgedeckt, der jedoch relativ häufig vorkommt. Man beachte, dass die Ausdruckskraft von SQL mit diesen Möglichkeiten über die Ausdruckskraft der relationalen Algebra – und damit auch des Relationenkalküls – hinausgeht.

4.17 Veränderungen am Datenbestand

In Abschnitt 4.3 wurde ja schon ein Befehl vorgestellt, mit dem Veränderungen am Datenbestand durchgeführt werden konnten: der **insert**-Befehl. Zusätzlich zur direkten Angabe konstanter Werte können Tupel auch durch eine Anfrage generiert werden. Sollte Sokrates beispielsweise der Meinung sein, dass alle Studenten seine Logik-Vorlesung besuchen sollen, kann er das mit folgendem SQL-Befehl – zumindest in der Datenbasis – bewirken:

```
insert into hören
    select MatrNr, VorlNr
    from Studenten, Vorlesungen
    where Titel = 'Logik';
```

Es ist möglich, beim Einfügen nur einen Teil der Attribute anzugeben, falls beispielsweise einige Werte unbekannt sind. Dazu werden die gewünschten Attribute in Klammern hinter dem Tabellennamen angegeben. Die nicht definierten Felder werden vom System mit einem Nullwert aufgefüllt. Im Beispiel wird der Nullwert durch einen Bindestrich angedeutet.

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

Studenten		
MatrNr	Name	Semester
⋮	⋮	⋮
29120	Theophrastos	2
29555	Feuerbach	2
28121	Archimedes	–

Zum Löschen wird der **delete**-Befehl verwendet, bei dem durch die Angabe einer Bedingung eine Auswahl unter den Tupeln getroffen werden kann. Diejenigen Studenten, die bereits länger als 13 Semester studieren, werden mit

delete from Studenten
where Semester > 13;

entfernt. Bestehende Zeilen können mit dem **update**-Befehl verändert werden. Bei Beginn eines neuen Semesters müssen in unserem Beispiel die Semesterzahlen der Studenten erhöht werden:

update Studenten
set Semester = Semester + 1;

Selbstverständlich wäre auch hier eine nähere Qualifizierung der zu ändernden Tupel mit einer **where**-Bedingung möglich.

Es ist noch wichtig zu wissen, dass alle Änderungsoperationen in SQL in zwei Schritten ausgeführt werden. Im ersten Schritt werden die Kandidaten für die Änderungsoperation gebildet, im zweiten Schritt wird dann die Operation auf den Kandidaten ausgeführt. Beim **insert**-Befehl wird zunächst eine temporäre Tabelle mit dem Ergebnis der **select**-Anfrage gebildet, die dann erst *komplett* in die Zieltabelle eingefügt wird. Bei **delete** werden alle zu löschenden Tupel markiert und dann *auf einmal* entfernt. Ein **update** führt die **set**-Operation basierend auf den Werten der Originaltabelle in einer temporären Tabelle durch. Erst dann überschreiben die modifizierten Tupel die Originaltabelle.

Ohne diese aufwendige Verarbeitung könnte das Ergebnis einer Änderungsoperation von der Reihenfolge, in der die Tupel verarbeitet werden, abhängen. Dies würde sicherlich der mengenorientierten Semantik einer deklarativen Sprache widersprechen. Wenn beispielsweise in der Relation *voraussetzen* nur noch direkte Abhängigkeiten von Grundlagenvorlesungen[8] gespeichert werden sollen, kann man alle anderen Tupel mit

[8] Grundlagenvorlesungen seien solche, zu denen es in *voraussetzen* keine *Vorgänger* gibt.

```
delete from voraussetzen
    where Vorgänger in ( select Nachfolger
                         from voraussetzen );
```

entfernen. Ohne einen Markierungsschritt hängt das Ergebnis dieser Anfrage von der Reihenfolge der Tupel in der Relation ab. Eine Abarbeitung in der Reihenfolge der Beispielausprägung in Abbildung 3.8 würde das letzte Tupel (5052, 5259) fälschlicherweise erhalten, da vorher bereits alle Tupel mit 5052 als *Nachfolger* entfernt wurden.

4.18 Sichten

Ein wichtiges Konzept, um ein Datenbanksystem an die Bedürfnisse unterschiedlicher Benutzergruppen anpassen zu können, sind Sichten (engl. *views*). Auf konzeptueller Ebene wurden sie bereits in Kapitel 2 eingeführt. Dort war eine Sicht eine Beschreibung der für eine bestimmte Benutzergruppe interessanten Datenmenge. Es ist aber nicht nur wichtig festzulegen, welche Daten Benutzer sehen wollen, sondern auch, welche sie nicht sehen dürfen. In Kapitel 12 werden Datenschutz-Mechanismen vorgestellt, um bestimmte Daten Benutzern zugänglich bzw. unzugänglich zu machen. Dieses wird im Allgemeinen mit Hilfe von Sichten verfeinert: Sie bieten virtuelle Relationen an, die nur einen Ausschnitt des gesamten Modells zeigen. „Virtuell" heißt in diesem Zusammenhang, dass keine neuen Tabellen angelegt werden, vielmehr werden sie bei jeder Verwendung neu berechnet. Ein Beispiel einer Sicht auf *prüfen* sei die Einschränkung, dass nicht alle Benutzer das Ergebnis einer Prüfung einsehen dürfen. Diese Einschränkung kann realisiert werden durch:

```
create view prüfenSicht as
    select MatrNr, VorlNr, PersNr
    from prüfen;
```

Wird die Sicht *prüfenSicht* nun in einer Anfrage verwendet, berechnet das Datenbanksystem automatisch an deren Stelle den obigen **select**-Ausdruck.

Man kann Daten auch dadurch anonymisieren, dass man sie verdichtet, also aggregiert. Ein Beispiel dafür ist die nachfolgende Sicht *PrüferHärte*:

```
create view PrüferHärte(Name, HärteGrad) as (
    select prof.Name, avg(pruef.Note)
    from Professoren prof join prüfen pruef on
          prof.PersNr = pruef.PersNr
    group by prof.Name, prof.PersNr
          having count(*) >= 50 )
```

Hier wird also eine Sicht definiert, die den Professoren ihre in allen bisher abgenommenen Prüfungen vergebene Durchschnittsnote zuordnet. Durch die **having**-Klausel werden die Professoren ausgenommen, die bislang weniger als 50 Prüfungen abgenommen haben. Dadurch wird die Anonymität einzelner Prüfungen auch dann gewährleistet, wenn sie von Professoren abgenommen wurden, die bislang noch keine oder sehr wenige Prüfungen abgenommen haben. Warum?

Eine andere Einsatzmöglichkeit ist die Vereinfachung von Anfragen. Man kann eine Sicht dabei als eine Art Makro benutzen. Die folgende Sicht assoziiert Studenten mit den Professoren, bei denen sie Vorlesungen gehört haben.

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

Da die Namen der Spalten nicht eindeutig sind, muss eine Neubenennung erfolgen – hier *SName* für die Namen der Studenten und *PName* für die Namen der Professoren. Die neuen Namen werden in Klammern hinter dem Namen der Sicht angegeben. Dieses Konstrukt muss auch benutzt werden, wenn die Werte einer Ergebnisspalte in der Anfrage erst berechnet werden. Man kann die so definierte Sicht dann ganz „normal" in Anfragen verwenden.

Um herauszufinden, in welchen Semestern die Studenten von Sokrates sind, genügt jetzt die sehr einfache Anfrage

```
select distinct Semester
from StudProf
where PName = 'Sokrates';
```

An dieser Anfrage ist gut erkennbar, dass man durch das Sichtenkonzept die Benutzung der Datenbank für bestimmte Benutzergruppen vereinfachen kann.

4.19 Sichten zur Modellierung von Generalisierungen

Bei der Modellierung von Generalisierungen dienen Sichten zur Realisierung von *Inklusion* und *Vererbung*: Objekte (hier Tupel) eines Untertyps einer Generalisierungshierarchie sollen auch automatisch zu ihrem Obertyp gehören und die Attribute des Obertyps erben. Dabei kann entweder der Obertyp oder der Untertyp als Sicht definiert werden. Bild 4.7 demonstriert die beiden Alternativen anhand der Generalisierung von *Professoren* und *Assistenten* zu *Angestellte*.

Die linke Alternative in Abbildung 4.7 (a) zeigt die Modellierung der Untertypen *Professoren* und *Assistenten* als Sicht. Die Relation *Angestellte* mit ihren zwei Attributen *PersNr* und *Name* existiert physisch in der Datenbasis. Zusätzlich werden zwei Relationen *ProfDaten* und *AssiDaten* gebildet. *ProfDaten* ergänzt die Daten der Angestellten, die auch Professoren sind, um die Attribute *Rang* und *Raum*. Analog enthält *AssiDaten* die fehlenden Attribute *Fachgebiet* und *Boss* für Assistenten. Diese beiden Relationen sind allerdings nicht Teil der Benutzerschnittstelle. Für die Benutzer werden zusätzlich zur sichtbaren Basisrelation *Angestellte* die zwei Sichten *Professoren* und *Assistenten* als Verbund (Join) der generellen und speziellen Daten definiert. Sie können wie bisher mit *Professoren* und *Assistenten* arbeiten.

Diese Modellierung bevorzugt Zugriffe auf die Daten des Typs *Angestellte*, aber benachteiligt Zugriffe auf *Professoren* und *Assistenten*. *Angestellte* sind unmittelbar

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

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

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

create view Professoren as
      select *
      from Angestellte a, ProfDaten d
      where a.PersNr = d.PersNr;

create view Assistenten as
      select *
      from Angestellte a, AssiDaten d
      where a.PersNr = d.PersNr;
```

(a) Untertypen als Sicht

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

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

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

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

(b) Obertypen als Sicht

Abbildung 4.7: Modellierungsmöglichkeiten für Generalisierungen

verfügbar, für *Professoren* und *Assistenten* müssen i.A. bei einer Anfrage die Daten durch einen relativ aufwendigen Join verbunden werden. Zusätzlich entstehen die im nächsten Abschnitt beschriebenen Probleme, wenn Sichten verändert werden sollen.

Die rechte Alternative in Abbildung 4.7 (b) realisiert die Generalisierung auf die umgekehrte Weise: *Professoren* und *Assistenten* sind als Relationen physisch in der Datenbasis vorhanden. Zusätzlich existiert eine Basisrelation *AndereAngestellte*, die es ermöglicht, auch Angestellte zu speichern, die weder Professoren noch Assistenten sind. Der Obertyp *Angestellte* wird als Sicht definiert, indem er, nach geeigneter Projektion, die Relationen *Professoren*, *Assistenten* und *AndereAngestellte* vereinigt. In diesem Fall werden Zugriffe auf die Untertypen bevorzugt. Mit Hilfe von Übungsaufgabe 4.23 können die Vor- und Nachteile der beiden Modellierungen anhand eines konkreten Beispiels nachvollzogen werden. Insbesondere sollten sich die Leser über die Möglichkeiten der Änderung von Daten bewusst werden – im nächsten Abschnitt gehen wir darauf ein, dass Sichten im Allgemeinen nicht änderbar sind.

Die obigen Beispiele zeigen letzten Endes, wie Sichten zur Gewährleistung logischer Datenunabhängigkeit eingesetzt werden können. Dies ist nochmals in Abbildung 4.8 skizziert. Die logische Datenunabhängigkeit schützt die Benutzer in gewissen Grenzen vor Veränderungen am Datenbankschema. Unabhängig davon, ob

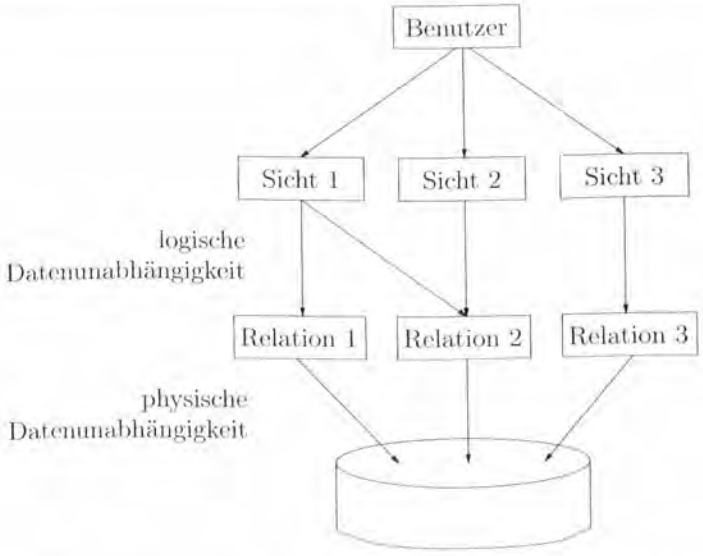


Abbildung 4.8: Sichten zur Gewährleistung von Datenunabhängigkeit

beispielsweise der Ober- oder der Untertyp als Sicht definiert wurde, den Benutzern wird eine einheitliche Schnittstelle geboten.

4.20 Charakterisierung update-fähiger Sichten

Sichten haben das inhärente Problem, dass sie nicht immer änderbar (update-fähig) sind. Ein anschauliches Beispiel ist die folgende Sicht:

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

Diese Sicht ist nicht veränderbar, da sie das berechnete Attribut *Durchschnittsnote* enthält. Eine Änderungsoperation lässt sich nicht mehr auf die ursprüngliche Basisrelation *prüfen* zurückpropagieren. Die folgende Operation würde also vom DBMS zurückgewiesen werden.

```
update WieHartAlsPrüfer
    set Durchschnittsnote = 1.0
    where PersNr = ( select PersNr
                     from Professoren
                     where Name = 'Sokrates' );
```

Nehmen wir an, es wäre eine Sicht zur Vermeidung des expliziten Joins von Vorlesungen und Professoren definiert. Es soll nun eine neue Vorlesung eingefügt werden:

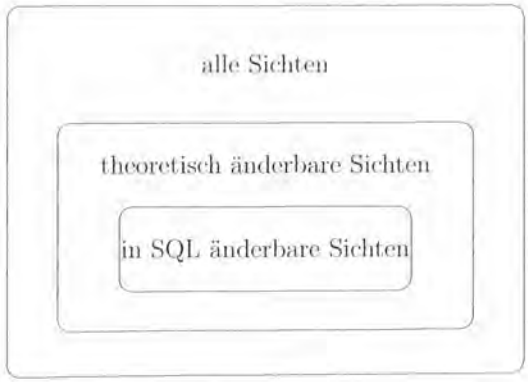


Abbildung 4.9: Änderbarkeit von Sichten

```
create view VorlesungenSicht as
    select Titel, SWS, Name
    from Vorlesungen, Professoren
    where gelesenVon = PersNr;
insert into VorlesungenSicht
    values ('Nihilismus', 2, 'Nobody');
```

Dieser Versuch wird jedoch scheitern, da Veränderungen hier nicht möglich sind. Um das obige Tupel einfügen zu können, müsste das DBMS die eingetragenen Werte den ursprünglichen Relationen zuordnen können. Das ist aber nicht immer möglich, da die Sicht die Schlüssel der ursprünglichen Relationen herausprojiziert hat. Im Allgemeinen sind Sichten veränderbar, wenn

- sie weder Aggregatfunktionen, noch Anweisungen wie **distinct**, **group by** und **having** enthalten,
- in der **select**-Liste nur eindeutige Spaltennamen stehen und ein Schlüssel der Basisrelation enthalten ist und
- sie nur genau eine Tabelle (also Basisrelation oder Sicht) verwenden, die ebenfalls veränderbar sein muss.

Grundsätzlich gilt, dass die gemäß SQL änderbaren Sichten eine Untermenge der theoretisch änderbaren Sichten darstellen. D.h. es gibt Sichtendefinitionen, bei denen theoretisch die eindeutige Propagierung von Änderungen auf die Basisrelationen möglich wäre, SQL diese Änderungen aber dennoch ausschließt. Diese Tatsache ist in Abbildung 4.9 skizziert.

4.21 Einbettung von SQL in Wirtssprachen

Viele Anwendungen erfordern die Einbettung von SQL in eine Wirtssprache (engl. Embedded SQL), z.B. wenn eine benutzerfreundliche Umgebung erstellt oder Turing-Vollständigkeit erreicht werden soll. Exemplarisch wollen wir hier die Einbettung

in C mit Hilfe eines Präcompilers untersuchen. SQL-Anweisungen werden dabei im Quelltext mit dem Präfix **exec sql** markiert und vom Präcompiler in entsprechenden Code umgewandelt. Sogenannte Kommunikationsvariablen bewirken den Datenaustausch zwischen dem C-Programm und dem DBMS. Bild 4.10 zeigt ein Programmbeispiel, das Studenten gemäß eingegebener Matrikelnummer exmatrikuliert.

Zunächst wird die Datei „SQLCA.h" eingebunden. Sie beinhaltet die Definition für eine Statusvariable, mit der Laufzeitfehler und Statusmeldungen des DBMS abgefragt werden können (SQL Communication Area).

Die ersten vier Zeilen des Hauptprogramms definieren Kommunikationsvariablen. Diese Variablen können von dort an sowohl im C-Programm als auch in SQL-Anweisungen verwendet werden. Um sie in SQL-Anweisungen von Datenbank-Objekten zu unterscheiden, müssen sie dort mit einem „:" markiert werden.

Zur Behandlung von Fehlerzuständen bietet der Präcompiler das **whenever**-Konstrukt an, welches die automatische Überprüfung der SQLCA steuert. Sobald der angegebene Fehlerzustand auftritt, hier also **sqlerror**, wird eine bestimmte Aktion ausgeführt. In unserem Fall wird zur Marke „*error:*" gesprungen. Alternativ können auch Funktionen aufgerufen (**do** function()), das Programm abgebrochen (**stop**) oder der Fehler einfach ignoriert werden (**continue**). Um Endlosschleifen zu vermeiden, sollte als erste Handlung bei Fehlern die Fehlerbenachrichtigung abgeschaltet werden.

Die Verbindung zur Datenbank wird durch einen **connect**-Befehl unter Angabe der Datenbank-Kennung durchgeführt. Die Kennung wird im Beispiel nach dem Start vom Benutzer eingegeben.

Im Hauptteil des Programms werden immer wieder Matrikelnummern abgefragt und Studenten mit der entsprechenden Matrikelnummer gelöscht, bis der Benutzer eine Null eingibt.

Die letzten Zeilen sorgen für ein ordnungsgemäßes Beenden der Transaktion und das Abmelden beim Datenbank-System. Näheres dazu in Kapitel 9.[9]

4.22 Anfragen in Anwendungsprogrammen

Bei der Verwendung des **select**-Befehls in Anwendungsprogrammen muss man zwischen zwei Arten von Anfragen unterscheiden: solche, die höchstens ein Tupel zurückliefern und solche, die mehrere Tupel – also Relationen – zurückliefern können. Im ersten Fall genügt eine einfache Angabe, in welche Kommunikationsvariablen die Attribute des Tupels kopiert werden sollen. Die Berechnung des Durchschnitts der Semesterzahlen der Studenten ist ein Beispiel dafür; sei *avgsem* eine entsprechend deklarierte Kommunikationsvariable.

```
exec sql select avg(Semester)
         into :avgsem
         from Studenten;
```

[9]Wenn ein Programm nicht durch explizite Angaben von **commit**- oder **rollback**-Angaben anders aufgeteilt wird, wird es als eine Transaktion behandelt und bei Termination automatisch ein **rollback** ausgeführt. Die **rollback**-Operation stellt den Originalzustand der Datenbasis vor Ausführung des Programms wieder her. **release** sorgt für die Freigabe aller Sperren und das Abmelden von der Datenbank.

```
#include <stdio.h>

/* Kommunikationsvariablen deklarieren */
exec sql begin declare section;
    varchar user_passwd[30];
    int exMatrNr;
exec sql end declare section;

exec sql include SQLCA;

main()
{
    /* Benutzeridentifikation und Authentisierung */
    printf("Name/Password: ");
    scanf("%s", user_passwd.arr);
    user_passwd.len = strlen(user_passwd.arr);

    exec sql whenever sqlerror goto error;
    exec sql connect :user_passwd;

    while (1) {
        printf("Matrikelnummer (0 zum beenden): ");
        scanf("%d", &exMatrNr); /* einlesen der MatrNr */
        if (!exMatrNr) break; /* bei Eingabe von 0 verlasse Schleife */

        exec sql delete from Studenten
                 where MatrNr = :exMatrNr;
    }

    exec sql commit work release;
    exit(0);

error:
    exec sql whenever sqlerror continue;
    exec sql rollback work release;
    printf("Fehler aufgetreten!\n");
    exit(-1);
}
```

Abbildung 4.10: Ein Programmbeispiel mit Embedded SQL

Der zweite Fall, wenn als Ergebnis also Mengen von Tupeln zurückgeliefert werden, gestaltet sich schwieriger: Die traditionellen Programmiersprachen besitzen keine eingebauten Möglichkeiten zur Verwaltung von Mengen. Hier wird das sogenannte *Cursor-Konzept* verwendet. Mit diesem Konzept kann man eine Menge von Tupeln iterativ, eines nach dem anderen, bearbeiten. Der Cursor zeigt dabei jeweils auf das Tupel, das aktuell in Bearbeitung ist.

In Embedded SQL wird ein Cursor in vier Schritten benutzt, die, zusätzlich zur folgenden Beschreibung, in Abbildung 4.11 grafisch dargestellt sind. Zunächst muss der Cursor deklariert und damit die zugehörige Anfrage festgelegt werden:

```
exec sql declare c4profs cursor for
        select Name, Raum
        from Professoren
        where Rang = 'C4';
```

Im zweiten Schritt wird der Cursor geöffnet, wodurch er implizit auf das erste Tupel der Ergebnismenge positioniert wird:

```
exec sql open c4profs;
```

Nun können die Daten Schritt für Schritt zum Anwendungsprogramm übertragen werden. Liegen keine Daten mehr an, wird dies durch entsprechendes Setzen der Statusvariablen angezeigt.

```
exec sql fetch c4profs into :pname, :praum;
```

Im letzten Schritt wird der Cursor geschlossen. Erst durch Schließen und erneutes Öffnen kann der Cursor wiederverwendet werden.

```
exec sql close c4profs;
```

Die Einbettung von SQL in Programmiersprachen hat diverse Nachteile. Wie bereits erwähnt, besitzen die meisten traditionellen Programmiersprachen keine eingebauten Möglichkeiten zur Mengenverarbeitung. Sie bearbeiten Datensätze iterativ (one record at a time), während SQL mengenorientiert arbeitet. Diesen Gegensatz nennt man *Impedance Mismatch*. Das Cursorkonzept ist eine künstliche Angleichung an die tupelorientierte Arbeitsweise. Bei komplexeren Anwendungen entsteht vielfach ein „Reibungsverlust", der durch das wiederholte Schließen und Öffnen eines Cursors oder das Zwischenspeichern von bereits geholten Ergebnissen entsteht. Diese Situation wurde mit SQL-92 zwar etwas gelindert, da dort die Cursor-Steuerung verbessert wurde. Im Endeffekt bleibt der Paradigmenunterschied zwischen satz- und mengenorientierter Verarbeitung allerdings bestehen.

4.23 JDBC: Java Database Connectivity

Bei der Entwicklung von Internet-Anwendungen hat sich die Programmiersprache Java durchgesetzt, deshalb wollen wir nachfolgend die beiden Datenbankanbindungen JDBC und SQLJ für Java-Programme vorstellen. Im vorigen Abschnitt haben wir die Einbettung von SQL in die Programmiersprache C bzw. C++ beschrieben.

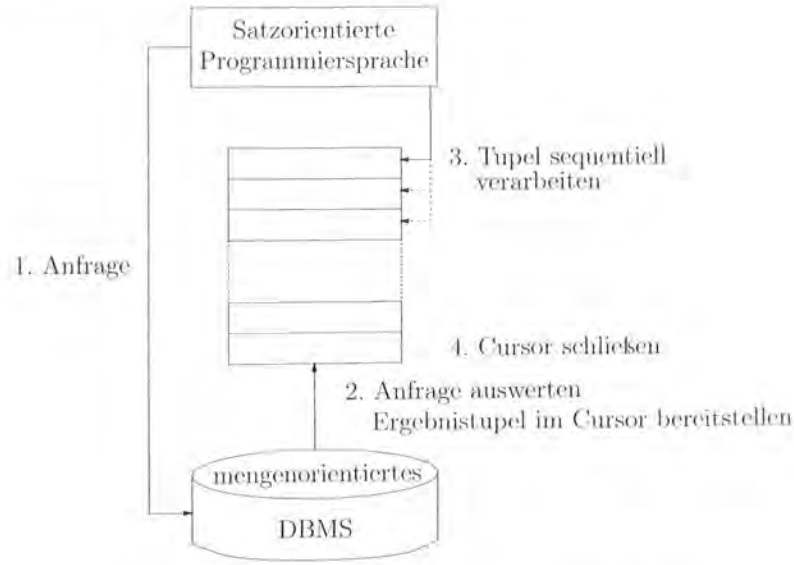


Abbildung 4.11: Grafische Veranschaulichung der Cursor-Schnittstelle

Eine ähnliche Einbettung von SQL in Java existiert mittlerweile auch unter dem Namen SQLJ. Der Vorteil der „echten" Einbettung von SQL in eine Programmiersprache besteht darin, dass man schon zur Übersetzungszeit die syntaktische Korrektheit sowie die Typkonsistenz der SQL-Ausdrücke überprüfen kann. Als Nachteil ist zu sehen, dass man sich auf statische (also fest vorgegebene) SQL-Ausdrücke beschränken muss. Damit wäre z.B. die von uns über die Webseite

```
http://www-db.in.tum.de/research/publications/books/DBMSeinf
```

zur Verfügung gestellte SQL-Schnittstelle, mit der Studierende beliebige SQL-Anfragen formulieren und testen können, sehr schwer zu realisieren. Ein weiterer möglicher Nachteil der Einbettung von SQL besteht darin, dass die Datenbank-Hersteller proprietäre SQL-Erweiterungen realisiert haben, die die Portierung eines Programms mit eingebetteten SQL-Ausdrücken von einem Datenbanksystem auf ein anderes erschweren. Auch ist der Zugriff auf mehrere (heterogene) Datenbanksysteme mittels eingebettetem SQL schwierig zu realisieren.

Für hochgradig dynamische Zugriffe auf Datenbanken wurde ursprünglich die standardisierte Schnittstelle namens Open Database Connectivity (ODBC) entwickelt, die für C bzw. C++ konzipiert war. Nach dem „Siegeszug" der Programmiersprache Java wurde diese Schnittstelle speziell für Java-Programme angepasst und ist unter dem Namen JDBC (Java Database Connectivity) bekannt. Diese Schnittstelle bezeichnet man manchmal auch als Call-Level Interface (CLI). Die SQL-Anweisungen werden als (dynamisch generierbare) Text-Strings an das Datenbanksystem übergeben. Wenn man sich bei der Verwendung der JDBC-Schnittstelle auf standardisierte SQL-Konstrukte beschränkt – also proprietäre SQL-Erweiterungen

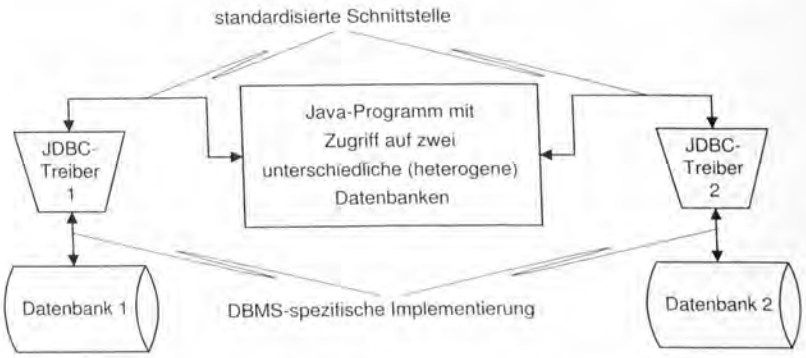


Abbildung 4.12: Zugriff auf mehrere Datenbanken via JDBC

der einzelnen Datenbank-Hersteller vermeidet – erzielt man leicht portierbare Programme. D.h. solche Anwendungen können auf unterschiedlichen Datenbanksystemen laufen und es ist auch möglich, dass man aus einem Java-Programm problemlos auf mehrere unterschiedliche (heterogene) Datenbanken zugreift. Dieser letzte Aspekt ist in Abbildung 4.12 illustriert: Die Programmierschnittstelle ist standardisiert; die Treiber sind Datenbanksystem-spezifisch realisiert.

4.23.1 Verbindungsaufbau zu einer Datenbank

Bevor man von einem Java-Programm mit der Datenbank kommunizieren kann, muss eine Verbindung aufgebaut werden. Die JDBC-Schnittstelle ist in dem Java-Package `java.sql` definiert, das importiert werden muss. Hierin ist u.a. auch ein sogenannter *DriverManager* enthalten, der die JDBC-Treiber verwaltet. Man muss den bzw. die notwendigen JDBC-Treiber laden, damit der *DriverManager* diese für den Verbindungsaufbau verwenden kann. Dies kann durch den Java-Ausdruck `Class.forName("`*`Treiber-Name`*`")` erfolgen. In unseren Beispielen verwenden wir zunächst einen Treiber für eine Oracle-Datenbank und später beim SQLJ-Beispiel (siehe Abbildung 4.14) einen Treiber für eine DB2-Datenbank, die wie folgt geladen werden:

```
Class.forName("oracle.jdbc.driver.OracleDriver");
// ...
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

Danach kann man die Verbindung (ein Objekt vom Typ *Connection*) kreieren. Diese Aufgabe delegiert man an den *DriverManager*, indem man ihm über die Operation `getConnection()` die Adresse der Datenbank (in der Form einer URL) übergibt. Wenn eine Benutzeridentifikation (also Benutzername und Passwort) für die Datenbank notwendig ist, kann man diese dem *DriverManager* mit übergeben. Der Aufruf sieht dann wie folgt aus:

```
Connection conn = DriverManager.getConnection(DB_URL, name, passwd);
```

In unserem lokalen Netz wäre eine Verbindung mit unserem Oracle-Datenbanksystem wie folgt aufzubauen:

```
Connection conn = DriverManager.getConnection(
      "jdbc:oracle:oci8:@lsintern-db", "kemper", "meinPassw");
```

Um SQL-Ausdrücke an die Datenbank absetzen zu können, muss man zuvor noch ein `Statement`-Objekt generieren. Dies erledigt die Operation `createStatement` der Klasse `Connection` wie folgt:

```
Statement sql_stmt = conn.createStatement();
```

Anfragen können jetzt über die Operation `executeQuery()` und Änderungsoperationen über `executeUpdate()` durchgeführt werden. Als Beispiel wollen wir die C4-Professoren der Universität ermitteln:

```
ResultSet rset = sql_stmt.executeQuery(
    "select Name, Raum from Professoren where Rang = 'C4'" );
```

In diesem Programmfragment ist auch schon ein weiteres Konzept, der `ResultSet`, enthalten. Der `ResultSet` ist analog zum *Cursor*-Konzept in Embedded SQL und stellt eine Iterator-Schnittstelle für den Zugriff auf die Menge von Ergebnistupeln der SQL-Anfrage bereit. Von der Schnittstelle der Klasse `ResultSet` verwenden wir hier nur die eine Operation `next()`. Diese Operation schaltet auf das nächste (bzw. beim erstmaligen Aufruf auf das erste Ergebnistupel) und liefert den Boole'schen Wert *true* falls dieses existiert und sonst *false*.[10]

Zur Verarbeitung der Ergebnismenge iteriert man also z.B. in einer `while`-Schleife durch den `ResultSet`:

```
while(rset.next()) {
   System.out.print(rset.getString("Name"));
   // ...
   System.out.println(rset.getInt("Raum"));
}
```

Mit `rset.getString("Name")` greift man auf den String-Wert des Attributs `Name` des derzeit aktuellen Ergebnistupels zu. Man hätte auch über die Position des Attributs zugreifen können, also mit `reset.getString(1)`. Analog dazu extrahiert man den Attributwert von `Raum` mit `getInt("Raum")`. In unserem Fall werden diese Daten einfach nur ausgegeben – natürlich könnten die Daten hier beliebig verarbeitet werden, indem man sie in entsprechende Java-Objekte bzw. Variablen kopiert.

In unserem Beispiel waren die Attribut-Namen und -Typen durch die **select**-Klausel bekannt. Manchmal – insbesondere bei dynamisch generierten Anfragen, die z.B. vom Benutzer interaktiv eingegeben werden könnten – ist diese Information nicht (statisch) bekannt. Deshalb kann man vom jeweiligen `ResultSet` auch Metadaten über die Struktur der Ergebnistupel erfragen. Diese Information wird als `ResultSetMetaData`-Objekt zur Verfügung gestellt:

[10] Falls die SQL-Anfrage ein leeres Ergebnis lieferte, gibt `next()` schon beim ersten Aufruf *false* zurück und ein Zugriff auf ein Ergebnistupel ist dann natürlich nicht möglich.

```
ResultSetMetaData rsm = rset.getMetaData();
```

Über dieses Objekt ist die Anzahl der Attribute als `rsm.getColumnCount()` abzufragen. Der i-te Attributname ist als `rsm.getColumnName(i)` und der Typ des i-ten Attributs als `rsm.getColumnType(i)` ermittelbar. Bei Strings benötigt man für die Formatierung häufig auch deren Darstellungsbreite (Anzahl von Zeichen), die man mit der Operation `rsm.getColumnDisplaySize(i)` erfragen kann. Unter Nutzung dieser Metadaten kann man dann sehr flexible Web-Schnittstellen, über die Benutzer beliebige SQL-Anfragen absetzen können, realisieren (siehe Übungsaufgabe 19.1).

Zum Schluss der Verarbeitung der Ergebnismenge sollte man das `Statement` schließen und am Ende der Anwendung auch die Verbindung zur Datenbank „ordnungsgemäß" wieder schließen:

```
sql_stmt.close();
conn.close();
```

4.23.2 Resultset-Programmbeispiel

In obigen Erläuterungen von JDBC haben wir die Ausnahmenbehandlung, die in Java zwingend vorgeschrieben ist, der Übersichtlichkeit halber weggelassen. Bei allen Interaktionen mit der Datenbank müssen Ausnahmen (Exceptions) abgefangen werden. Dies geschieht in Java, indem man die entsprechenden Aktionen mittels `try{...} catch(...) {...}` wie folgt klammert:

```
try
   {
       // Datenbank-Interaktionen
   }
catch( ... )
   {
       // Ausnahmebehandlung
   }
```

In Abbildung 4.13 findet sich jetzt ein vollständiges (aber sehr kleines) Java-Programm, mit dem zunächst die durchschnittliche Semesterzahl der Studierenden und danach die C4-Professoren der Universität ausgegeben wird. Bei der Ermittlung der durchschnittlichen Semesterzahl erkennt man, dass in JDBC auch ein-elementige Ergebnisse von Anfragen über die `ResultSet`-Schnittstelle übermittelt werden.

4.23.3 Vorübersetzung von SQL-Ausdrücken

Die textuelle Weitergabe von SQL-Anweisungen an das Datenbanksystem erfordert jedesmal die vollständige Übersetzung und Optimierung der Anfrage bzw. der Änderungsoperation durch das Datenbanksystem. Dies kann sehr leicht zu einem Leistungsengpass führen. Zur Leistungssteigerung ist es in JDBC deshalb auch möglich, häufig auszuwertende SQL-Anweisungen nur einmal zu übersetzen und wiederzuverwenden. Dazu dienen die `PreparedStatements`.

```
import java.sql.*;   import java.io.*;

public class ResultSetExample {
  public static void main(String[] argv) {
    Statement sql_stmt = null;
    Connection conn = null;
    try {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      conn = DriverManager.getConnection("jdbc:oracle:oci8:@lsintern-db",
                                         "nobody", "Passwort");
      sql_stmt = conn.createStatement();
    }
    catch (Exception e) {
      System.err.println("Folgender Fehler ist aufgetreten: " + e);
      System.exit(-1);
    }
    try {
      ResultSet rset = sql_stmt.executeQuery(
        "select avg(Semester) from Studenten");
      rset.next();   // eigentlich sollte man noch prüfen, ob Ergebnis leer
      System.out.println("Durchschnittsalter: " + rset.getDouble(1));
      rset.close();
    }
    catch(SQLException se) {
      System.out.println("Error: " + se);
    }
    try {
      ResultSet rset = sql_stmt.executeQuery(
        "select Name, Raum from Professoren where Rang = 'C4'");
      System.out.println("C4-Professoren:");
      while(rset.next())
      {
        System.out.println(rset.getString("Name") + " " +
                           rset.getInt("Raum"));
      }
      rset.close();
    }
    catch(SQLException se) {
      System.out.println("Error: " + se);
    }
    try {
      sql_stmt.close();
      conn.close();
    }
    catch (SQLException e) {
      System.out.println("Fehler beim Schliessen der DB-Verbindung: " + e);
    }
  }
}
```

Abbildung 4.13: Ein JDBC-Programmbeispiel mit ResultSet

```
PreparedStatement sql_exmatrikuliere =
    conn.prepareStatement("delete from Studenten where MatrNr = ?");
```

Hierbei ist das Fragezeichen ein Platzhalter für einen erst später (zur Laufzeit) einzufügenden Parameterwert. Es können auch mehrere Parameter vorkommen, die alle mit einem ? notiert werden. Die Substitution der Parameterwerte erfolgt über die relative Position der ?-Zeichen im SQL-Ausdruck – sie werden also einfach durchnumeriert. Für unser Beispiel erfolgt die Substitution (`setInt`) und die Ausführung (`executeUpdate`) wie folgt:

```
int VomBenutzerEingeleseneMatrNr;
   // zu löschende MatrNr einlesen
sql_exmatrikuliere.setInt(1,VomBenutzerEingeleseneMatrNr);
int rows = sql_exmatrikuliere.executeUpdate();
if (rows == 1) System.out.println("StudentIn gelöscht.");
  else System.out.println("Kein/e StudentIn mit dieser MatrNr.");
```

Die `PreparedStatements` sollte man dann verwenden, wenn immer wieder das gleiche Update oder die gleiche Anfrage – mit unterschiedlichen Parametern – auszuführen ist. In unserem Beispiel der Exmatrikulation von Studenten also dann, wenn die Löschoperation innerhalb einer Schleife ausgeführt wird und die Matrikelnummern nur sukzessive vom Benutzer zu ermitteln sind.

4.24 SQLJ: Eine Einbettung von SQL in Java

In Abschnitt 4.21 haben wir die Einbettung von SQL in die Programmiersprache C bzw. C++ vorgestellt. Eine ähnliche Einbettung von SQL in Java wurde von verschiedenen Datenbank-Herstellern unter dem Namen SQLJ standardisiert. Die Implementierung von SQLJ geschieht auf der Basis von JDBC. Das legt die Frage nahe, warum man denn dann überhaupt SQLJ anstatt JDBC verwenden sollte. Beide Methoden der Datenbank-Anbindung haben ihre Vor- und Nachteile: JDBC ist extrem flexibel und ermöglicht es, dynamisch generierte SQL-Anweisungen an das Datenbanksystem zu übermitteln. Das hat den Vorteil der Flexibilität, aber den Nachteil fehlender Typüberprüfung. D.h. selbst Syntaxfehler können erst zur Laufzeit erkannt werden und müssen entsprechend abgefangen werden. Demgegenüber erlaubt SQLJ eine „echte" Einbettung der SQL-Anweisungen in das Java-Programm. Schon zur Übersetzungszeit werden die SQL-Anweisungen auf Typkonsistenz überprüft. Auch kann schon zur Übersetzungszeit eine Optimierung der SQL-Anfragen stattfinden. Dies war bei der JDBC-Anbindung nur bei den `PreparedStatements` möglich.

Wir wollen SQLJ in einem kurzen Beispielprogramm vorstellen. Es sollen die „Methusaleme" (also solche mit mehr als 13 Semestern) unter den Studenten ausgegeben werden und nachfolgend aus der Datenbank gelöscht werden. Die Datenbankanfrage liefert möglicherweise eine Menge von Ergebnistupeln, die mittels eines `Iterators` verarbeitet werden können. Dazu definiert man vorab den Iterator

```
#sql iterator StudentenItr (String Name, int Semester);
```

Der Präfix # dient dazu, dem Übersetzer des SQLJ-Programms Datenbankbefehle anzuzeigen. Man kann dann beliebig viele Iteratoren dieses `StudentenItr` kreieren. Mit

```
StudentenItr Methusaleme;
```

deklarieren wir einen derartigen Iterator, der dann in folgender Anfrage die Daten aus dem DBMS „aufnimmt":

```
#sql Methusaleme = { select s.Name, s.Semester
                     from Studenten s
                     where s.Semester > 13 };
```

Die Iteration durch die Ergebnismenge geschieht analog zur JDBC-Iteration mittels des Methodenaufrufs `next()`:

```
while (Methusaleme.next()) {
   System.out.println(Methusaleme.Name() + ":" +
                      Methusaleme.Semester()); }
```

Abschließend wird der Iterator durch einen `close()`-Aufruf geschlossen.

In Abbildung 4.14 ist das vollständige Programm mitsamt dem Verbindungsaufbau, der auf einer JDBC-Verbindung basiert, gezeigt. Dieses Programm wurde für eine DB2-Installation geschrieben und müsste für andere Datenbanksysteme entsprechend angepasst werden – andere JDBC-Treiber und Datenbankadressen.

Im Beispiel führen wir auch das Löschen der „Methusaleme" aus der Datenbank durch. Datenmanipulationsbefehle können direkt an das Datenbanksystem abgesetzt werden. Wenn man, so wie wir in dem Beispielprogramm, das `AutoCommit` abgeschaltet hat, sollte man den expliziten `commit`-Befehl nicht vergessen.

4.25 Query by Example

Alternativ zu SQL bieten einige Datenbanksysteme auch das benutzerfreundliche Query-by-Example (QBE) als Anfragesprache an. Es ist Anfang der 70er Jahre von IBM entwickelt worden und wurde später Bestandteil von DB2. Ungewöhnlich ist bei QBE, dass man direkt mit Mustern von Tabellen arbeitet um eine Anfrage zu formulieren. Während SQL dem tupelorientierten Relationenkalkül angelehnt ist, basiert QBE auf dem relationalen Domänenkalkül (siehe Abschnitt 3.5.5). Variablen werden also an Attributdomänen (Wertebereiche) gebunden.

Sollen alle Vorlesungen mit mehr als drei Semesterwochenstunden gefunden werden, wird ein Formular der Tabelle *Vorlesungen* wie folgt ausgefüllt:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
		p._t	>3	

Die Spalten eines Formulares können Bedingungen und Kommandos enthalten. Für die Spalte SWS sollen alle Ergebniszeilen einen Wert größer drei aufweisen. Das Titel-Attribut jeder Ergebniszeile wird der Variable *_t* zugewiesen. Um Variablen

```
import java.io.*;
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator StudentenItr (String Name, int Semester);

public class SQLJExmp {
  public static void main(String[] argv) {
    try {
      Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
      Connection con = DriverManager.getConnection("jdbc:db2:uni");
      con.setAutoCommit(false);
      DefaultContext ctx = new DefaultContext(con);
      DefaultContext.setDefaultContext(ctx);

      StudentenItr Methusaleme;

      #sql Methusaleme = { select s.Name, s.Semester
                           from Studenten s
                           where s.Semester > 13 };

      while (Methusaleme.next()) {
         System.out.println(Methusaleme.Name() + ":" +
                            Methusaleme.Semester());
      }

      Methusaleme.close();

      #sql { delete from Studenten
                   where Semester > 13 };

      #sql { commit };

    }
    catch (SQLException e) {
      System.out.println("Fehler mit der DB-Verbindung: " + e);
    }
    catch (Exception e) {
      System.err.println("Folgender Fehler ist aufgetreten: " + e);
      System.exit(-1);
    }
  }
}
```

Abbildung 4.14: SQLJ-Beispielprogramm

und Zeichenketten unterscheiden zu können, werden Variablen in QBE mit einem Unterstrich markiert. Der Eintrag **p.** ist ein Druck-Befehl (print), der bewirkt, dass die Variable _t jeweils ausgegeben wird.

QBE besitzt, wie bereits angedeutet, eine Analogie zum Domänenkalkül. Die obige Anfrage würde im Domänenkalkül wie folgt formuliert werden:

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

Bei Eingabe mehrerer Musterzeilen werden diese durch ein logisches „oder" verknüpft. Alle Studenten, die Vorlesung 5041 oder Vorlesung 5049 hören, findet man mit

hören	MatrNr	VorlNr
	p. _x	5041
	p. _y	5049

Wollte man die Studenten finden, die beide Vorlesungen hören, müsste man eine einzige Domänenvariable verwenden.

hören	MatrNr	VorlNr
	p. _x	5041
	p. _x	5049

Ein Join mehrerer Tabellen lässt sich durch die Bindung einer Variablen an mehrere Spalten andeuten. Um also einen Join von *Vorlesungen* und *Professoren* zu erzeugen, trägt man z.B. die Variable _x sowohl unter *gelesenVon* als auch unter *PersNr* ein. Die folgende Anfrage findet die Namen der Professoren, die „Mäeutik" lesen:

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
		Mäeutik		_x

Professoren	PersNr	Name	Rang	Raum
	_x	**p.** _n		

In unserem Fall ist dies nur ein Name, nämlich „Sokrates".

Direkt in eine Spalte einer Tabelle eingetragene Bedingungen können nur den Inhalt der Spalte betreffen. Es ist nicht möglich, auf diese Weise zwei Tabellenspalten zu vergleichen. Für komplexere Anfragen verwendet man daher eine sogenannte *Condition Box*, in der beliebige Bedingungen eingetragen werden können. Um beispielsweise einen anderen Studenten als Tutor betreuen zu können, sollte man eine höhere Semesterzahl als dieser haben:

Studenten	MatrNr	Name	Semester
		_s	_a
		_t	_b

conditions
_a > _b

Betreuen	potentiellerTutor	Betreuter
p.	_s	_t

Die Tabelle *Betreuen* ist eine temporäre Relation, die lediglich für den Zweck der Ausgabeprojektion erzeugt wird. Wird ein Kommando, wie hier **p.**, unterhalb des Tabellennamens angegeben, wird es für alle Spalten ausgeführt.

Wie in SQL gibt es auch in QBE ein Kommando zur Gruppierung (**g.**) und Aggregatfunktionen (**sum.**, **avg.**, **min.**, ...). Im Gegensatz zu SQL aber findet bei QBE immer eine Duplikateliminierung statt. Wo sie nicht erwünscht ist, kann sie durch den Befehl **all.** abgeschaltet werden. Im Allgemeinen ist das der Fall, wenn **sum.** und **avg.** benutzt werden. Die Anfrage nach der Summe der Semesterwochen der Professoren, die überwiegend lange Vorlesungen halten, wird auch mit Hilfe einer Condition Box beantwortet.

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

conditions
avg.all. _x>2

Für Veränderungen an der Datenbasis existieren in QBE, ebenfalls wie in SQL, drei Befehle: **i.** entspricht dem **insert**-Befehl, **u.** entspricht **update** und **d.** bewirkt ein Löschen (**delete**). Für die Eingabe neuer Tupel wird der **i.**-Befehl unter den Tabellennamen eingetragen und die Daten in die entsprechenden Spalten geschrieben. Im Falle eines Updates werden, wie bei einer Anfrage, Bedingungen an die zu ändernden Tupel eingetragen. Die Änderungsoperation wird durch Angabe von **u.**, gefolgt von einer Formel, in einer Spalte eingetragen. Auch beim Löschen von Tupeln können Bedingungen an Spalten gestellt werden. Anders als beim **delete**-Befehl in SQL kann nicht nur eine komplette Zeile durch Angabe von **d.** unterhalb des Tabellennamens gelöscht werden, sondern auch einzelne Attribute. Im letzteren Fall wird das **d.** unterhalb des Attributnamens eingetragen und als Ergebnis der Attributwert an dieser Stelle zu einem Nullwert geändert. Auch ein gemeinsames Löschen in mehreren Tabellen ist möglich. Das Austragen von Sokrates, aller seiner Vorlesungen und der Belegungstupel für diese Vorlesungen in *hören* wird, wie bei einem Join, durch die Bindung einer Variablen an mehreren Tabellen vorgenommen.

Professoren	PersNr	Name	Rang	Raum
d.	_x	Sokrates		

Vorlesungen	VorlNr	Titel	SWS	gelesenVon
d.	_y			_x

hören	VorlNr	MatrNr
d.	_y	

In SQL kann diese Form des Löschens von Daten als Integritätsbedingung angegeben werden, wie das nächste Kapitel erläutert.

4.26 Übungen

4.1 Übersetzen Sie die Anfragen aus Aufgabe 3.4 in SQL.

4.2 Welche Bedingung muss gelten, damit eine geschachtelte Anfrage mit **in** in eine gleichwertige, nicht geschachtelte Anfrage umgewandelt werden kann? Geben Sie ein Beispiel an, bei dem eine Übersetzung möglich ist, und eines, bei dem keine Übersetzung möglich ist.

4.3 Bei numerischen Argumenten können Anfragen mit **all** in äquivalente Anfragen ohne die Verwendung von **all** umgeformt werden. Geben Sie zu den drei Vergleichsoperationen >= **all**, = **all** und <= **all** je ein Beispiel und seine Überformung an.

4.4 Suchen Sie unter Verwendung von **any** die Professoren heraus, die Vorlesungen halten. Finden Sie mindestens zwei weitere alternative äquivalente Formulierungen dieser Anfrage.

4.5 Finden Sie die Studenten, die *alle* Vorlesungen gehört haben.

Ihre Anfrage soll aber – anders als die im Text angegebene Formulierung – auch bei einer möglichen Verletzung der referentiellen Integrität das korrekte Ergebnis liefern. Was müssten Sie zusätzlich machen, wenn die Relation *hören* sogar Duplikate enthalten könnte?

Geben Sie zwei Formulierungen an: Einmal mit geschachtelten **not exists**-Unteranfragen und zum anderen unter Verwendung der Aggregatfunktion **count**.

4.6 Geben Sie eine alternative Anfrageformulierung zur Ermittlung der Studenten, die alle vierstündigen Vorlesungen gehört haben. Können Sie immer noch die Aggregatfunktion **count** verwenden, um dadurch auf den Existenzquantor **exists** ganz verzichten zu können? Die Antwort lautet ja; aber wie?

4.7 Finden Sie die Studenten mit der größten Semesterzahl unter Verwendung von Aggregatfunktionen.

4.8 Berechnen Sie die Gesamtzahl der Semesterwochenstunden, die die einzelnen Professoren erbringen. Dabei sollen auch die Professoren berücksichtigt werden, die keine Vorlesungen halten.

4.9 Finden Sie die Namen der Studenten, die in keiner Prüfung eine bessere Note als 3.0 hatten.

4.10 Berechnen Sie mit Hilfe einer SQL-Anfrage den Umfang des Prüfungsstoffes jedes Studenten. Es soll der Name des Studenten und die Summe der Semesterwochenstunden der Prüfungsvorlesungen ausgegeben werden.

4.11 Finden Sie Studenten, deren Namen den eines Professors enthalten. Hinweis: In SQL gibt es einen Operator „||", der zwei Zeichenketten aneinanderhängt.

4.12 Alle Studenten müssen ab sofort alle Vorlesungen von Sokrates hören. Formulieren Sie einen SQL-Befehl, der diese Operation durchführt.

4.13 Ermitteln Sie den Bekanntheitsgrad der Professoren unter den Studenten, wobei wir annehmen, dass Studenten die Professoren nur durch Vorlesungen oder Prüfungen kennen lernen.

4.14 Ermitteln Sie für die einzelnen Vorlesungen die Durchfallquote als die Anzahl der für diese Vorlesung angetretenen Prüflinge relativ zur Anzahl der durchgefallenen Prüflinge.

Als Variation der obigen Anfrage ermitteln Sie die Durchfallquote bei den einzelnen Professoren.

4.15 Ermitteln Sie den Median der Relation *prüfen*. (Die SQL-Formulierung dieser Anfrage ist nicht ganz einfach und wird in dem Buch von Celko (1995) diskutiert.)

4.16 Überlegen Sie sich einige Anfragen, bei denen die erweiterten Joinoperationen sinnvoll eingesetzt werden können.

4.17 Bestimmen Sie für alle Studenten eine gewichtete Durchschnittsnote ihrer Prüfungen. Die Gewichtung der einzelnen Prüfungen erfolgt nach zwei Kriterien: Prüfungen zu langen Vorlesungen sollen eine größere Rolle spielen als Prüfungen zu kurzen Vorlesungen. Prüfer, die im Schnitt sehr gute Noten vergeben, führen zu einer Abwertung des Prüfungsergebnisse, während Prüfer mit im Schnitt sehr schlechten Noten das Ergebnis aufwerten. Hinweis: Komplexere Anfragen lassen sich am besten durch Sichten in einfachere Teilanfragen modularisieren.

4.18 Nehmen wir an, dass in der Relation *Professoren* deren Geburtsdatum gespeichert ist. Der Rektor der Universität möchte nun von Ihnen eine Liste aller Professoren, die in den nächsten 45 Tagen Geburtstag haben. Informieren Sie sich, wie Sie eine entsprechende Anfrage in einer Ihnen zur Verfügung stehenden SQL-Schnittstelle realisieren könnten. Ist das mit Standard-Befehlen möglich? Funktioniert Ihre Anfrage auch, wenn ein Professor am 29. Februar eines Schaltjahres geboren wurde?

4.19 Formulieren Sie, ausgehend von dem in Aufgabe 3.3 eingeführten relationalen Wahlinformationssystem-Schema, folgende Anfragen in SQL:

1. Hat die CSU alle Direktmandate in Bayern im Jahr 2005 holen können?
2. Ermitteln Sie für jede Partei die Anzahl der „gewonnenen" Bundesländer, d.h. Bundesländer, in denen sie die Mehrzahl der (Zweit-)Stimmen erhalten hat.

4.20 **Projektarbeit**: In Abschnitt 22.3 ist der TPC-H/R-Benchmark beschrieben. Das Datenbankschema des Benchmarks modelliert ein (hypothetisches) Handelsunternehmen. Der Benchmark besteht im Wesentlichen aus 22 betriebswirtschaftlichen „Decision Support"-Anfragen, die dort verbal beschrieben sind. Formulieren Sie diese Anfragen in SQL.

4.21 Anfragen liefern beim Auftreten von Nullwerten oft unerwartete Ergebnisse. Folgende Anfragen sollen die Vorlesungen liefern, bei denen sich keiner der Sokrates-Assistenten auskennt:

```
select * from Vorlesungen
where Titel not in
  ( select Fachgebiet from Assistenten
    where Boss = 2125 )
```

```
select * from Vorlesungen
where Titel not exists
  ( select * from Assistenten
    where Boss = 2125 and
          Fachgebiet = Titel )
```

Wenn es nun lediglich einen Sokrates-Assistenten gibt, der sich noch nicht für ein Fachgebiet entschieden hat (dies also **null** ist), dann liefern die beiden Anfragen unterschiedliche Ergebnisse. Warum? Zeigen Sie was passiert.

4.22 Verwenden Sie das in SQL-92 enthaltene **case**-Konstrukt, um folgende Anfrage möglichst kompakt zu formulieren: Ermitteln Sie für jeden Prüfer die Anzahl der Prüfungen, die gut (besser als 2.0), die Anzahl der Prüfungen, die mittelmäßig (zwischen 2.0 und 3.0), die Anzahl der Prüfungen, die knapp bestanden wurden sowie die Anzahl der Prüfungen, die nicht bestanden wurden. Dazu kann man mehrere **case**-Konstrukte in der **select**-Klausel in Verbindung mit der **sum**-Aggregation verwenden.

4.23 Diskutieren Sie die Vor- und Nachteile der beiden relationalen Modellierungsmöglichkeiten der Generalisierung wie sie in Abbildung 4.7 (a) und (b) demonstriert wurden. Arbeiten Sie dazu konkret die Beispielausprägung in Abbildung 3.8 für die beiden Alternativen um.

4.24 Trotz **connect by**-Befehl ist Oracle nicht Turing-vollständig. Geben Sie textuell eine Anfrage an, die sich nicht im SQL-Dialekt von Oracle formulieren lässt. Geben Sie Gründe dafür an.

4.25 Schreiben Sie ein Embedded-SQL Programm, das zu einer eingegebenen Vorlesung alle Vorgänger aus der Datenbank entfernt. Verwenden Sie dabei nicht den **connect by**-Befehl. Hinweis: Benutzen Sie eine temporäre Relation.

4.26 Implementieren Sie obiges Programm in JDBC und SQLJ.

4.27 Führen Sie für eine künstlich generierte Universitäts-Datenbank eine vergleichende Leistungsanalyse von SQLJ und JDBC durch. Bei welcher Art von Anwendung schneidet SQLJ besser ab.

4.28 Falls Sie Zugriff auf zwei unterschiedliche Datenbanken haben, realisieren Sie ein JDBC-Beispielprogramm, das Informationen dieser beiden heterogenen Datenbanken verknüpft.

4.29 Formulieren Sie die Anfrage aus Aufgabe 4.10 in QBE.

4.30 Finden Sie die indirekten Vorgänger zweiter Stufe einer Vorlesung in QBE.

4.31 Bestimmen Sie alle Studenten, die zu jeder Vorlesung, die sie hören (gehört haben), bereits die Prüfung abgelegt haben. Erstellen Sie zwei alternative SQL-Anfragen, einmal mit **not exists** und einmal mittels **count**. Liefern

Ihre Anfragen identische Ergebnismengen, wenn Studenten sich beispielsweise über Vorlesungen prüfen lassen, die sie nicht hören?

4.32 Finden Sie heraus, ob es für Prüfungen von Vorteil ist, die jeweiligen Vorlesungen auch gehört zu haben. Ermitteln Sie dazu die Durchschnittsnote der Prüfungen, zu denen die Studenten die Vorlesungen nicht gehört haben und die Durchschnittsnote der Prüfungen, zu denen sie die Vorlesungen gehört haben.

4.33 Gegeben sei ein erweitertes Universitätsschema mit der folgenden *StudentenGF*-Relation:

StudentenGF : {[MatrNr : integer, Name : varchar(20), Semester : integer, Geschlecht : char, FakName : varchar(20)]}

Ermitteln Sie den Frauenanteil an den verschiedenen Fakultäten in SQL!

Geben Sie auch eine Lösung mit dem **case**-Konstrukt an.

4.34 Gegeben sei ein erweitertes Universitätsschema mit den folgenden *StudentenGF*- und *ProfessorenF*-Relationen:

ProfessorenF : {[PersNr : integer, Name : varchar(20), Rang : char(2), Raum : int, FakName : varchar(20)]}

Ermitteln Sie in Relationenalgebra und SQL die Studenten, die alle Vorlesungen ihrer Fakultät hören.

4.27 Literatur

Sequel wurde von Chamberlin und Boyce (1974) entworfen und als Vorläufer von SQL von Astrahan et al. (1976) beschrieben. Die Standards für SQL sind in ANSI (1986) und ANSI (1992) festgelegt (für SQL-86 bzw. SQL-92). Es empfiehlt sich aber eher, eines der zahlreichen Textbücher zu verwenden, wie sie z.B. von Date (1997) für SQL-86 und von Melton und Simon (1993) für SQL-92 verfasst wurden. Dürr und Radermacher (1990) gehen ausführlich auf SQL ein. Celko (1995) erläutert Fallstricke im Umgang mit SQL-92 und gibt viele praktische Tipps. Der Webserver des National Institute of Standards and Technology (1997) enthält eine Test-Suite, um Datenbanksysteme hinsichtlich der Einhaltung des SQL2-Standards zu testen.

Mittlerweile gibt es auch einen SQL:1999-Standard, der manchmal auch als SQL 3 bezeichnet wird. Zusammenfassungen darüber gibt es von Kulkarni (1994) und Melton (1994). Pistor (1993) beschreibt SQL 3 in einer Ausgabe des Informatik Spektrums. Mattos und DeMichiel (1994) diskutieren Designentscheidungen für SQL 3. Die vollständige Syntax von SQL:1999 ist in dem Buch von Melton und Simon (2001) enthalten. Die objekt-relationalen Erweiterungen von SQL:1999 werden wir in Kapitel 14 behandeln.

QBE wurde von Zloof (1975) auf der National Computer Conference vorgestellt. Scharnofske, Lipeck und Gertz (1997) haben eine orthogonale Erweiterung von QBE vorgeschlagen, um Unteranfragen sauber formulieren zu können.

Eine Konkurrenzsprache zu SQL war QUEL, das innerhalb des INGRES-Projektes entworfen wurde [Stonebraker et al. (1976)]. QUEL konnte sich aber kommerziell nicht gegen SQL durchsetzen, obwohl es von vielen Datenbankforschern als die „konzeptuell sauberere" Sprache angesehen wurde.

Ceri und Gottlob (1985) beschreiben die Übersetzung von SQL in die relationale Algebra. Anfragen mit Allquantoren wurden von Claussen et al. (1997) untersucht. Disjunktive Anfragen werden von Claussen et al. (2000) behandelt. Gottlob, Paolini und Zicari (1988) und Scholl, Laasch und Tresch (1991) untersuchen, wann sich Änderungsoperationen auf Sichten konsistent in die Datenbasis übertragen lassen. Neuhold und Schrefl (1988) beschäftigen sich mit der dynamischen Erzeugung von Sichten.

Moos und Daues (1997) behandeln die Anfrageformulierung schwerpunktmäßig für das Datenbanksystem DB2 von IBM.

Hamilton, Cattell und Fisher (1997) behandeln JDBC sehr detailliert; Saake und Sattler (2000) beschreiben den gesamten Themenkomplex Java und Datenbanken – allerdings ohne auf die Internetanbindung von Datenbanken mittels Java-Schnittstellen einzugehen, die wir in Kapitel 19 diskutieren. Eine sehr umfassende Behandlung von SQL im Zusammenspiel mit Java wird von Melton und Eisenberg (2000) geleistet.

5. Datenintegrität und temporale Daten

Die Aufgabe eines DBMS ist nicht nur die Unterstützung bei der Speicherung und Verarbeitung von großen Datenmengen, sondern auch bei der Gewährleistung der Konsistenz der Daten. Dieses Kapitel beschäftigt sich mit sogenannten *semantischen Integritätsbedingungen*, also solchen, die aus Eigenschaften der modellierten Miniwelt abgeleitet werden können. Die Erhaltung der Konsistenz der Daten bei Systemfehlern und unter Mehrbenutzerzugriff sowie der Schutz vor unerlaubter Manipulation werden in späteren Kapiteln besprochen. Aber auch die funktionalen Abhängigkeiten aus der relationalen Entwurfstheorie – eine Verallgemeinerung des Schlüsselbegriffs (siehe Kapitel 6) – können als semantische Integritätsbedingungen aufgefasst werden.

Die zentrale automatische Überprüfung von Integritätsbedingungen ist ein relativ aktuelles Thema und erst seit neuerer Zeit in kommerziellen relationalen Systemen enthalten. Erste Standardisierungsmaßnahmen dafür wurden in SQL-89, dem Vorläufer von SQL-92, vorgenommen. Der Vorteil eines solchen Mechanismus liegt auf der Hand: Wechselnde oder wachsende Konsistenzanforderungen brauchen nur einmalig dem DBMS in deklarativer Form bekannt gemacht und müssen nicht manuell in alle Anwendungsprogramme eingebaut werden. Damit werden Fehleranfälligkeit und Wartungsaufwand reduziert; außerdem können die oft komplexen Überprüfungsmaßnahmen z.B. zur Beschleunigung von Massendateneingaben kurzfristig zentral ausgeschaltet werden, was bei einer „manuellen" Lösung nicht trivial wäre.

Man unterscheidet statische und dynamische Integritätsbedingungen. Eine statische Bedingung muss von jedem Zustand der Datenbank erfüllt werden. Professoren dürfen z.B. nur entweder den Rang C2, C3 oder C4 haben. Dynamische Bedingungen werden an Zustandsänderungen gestellt: Beispielsweise dürfen Professoren nur befördert, aber nicht degradiert werden. Ihr Rang darf daher z.B. nicht von C4 auf C3 gesetzt werden.

Bisher haben wir schon verschiedene implizite Anforderungen an die Datenintegrität kennengelernt:

- Durch die Definition von Schlüsseln wurde bestimmt, dass keine zwei Tupel mit gleichem Wert in allen Schlüsselattributen existieren dürfen.

- Bei der konzeptuellen Modellierung wurden die Kardinalitäten der Beziehungen festgelegt. Beispielsweise können Professoren mehrere Vorlesungen halten, aber eine Vorlesung wird nicht von mehreren Professoren gehalten. Diese 1:*N*-Beziehung wurde bei der Übertragung ins relationale Modell fest eingebaut: *Vorlesungen* enthält ein Attribut *gelesenVon*, das auf den Primärschlüssel von *Professoren* verweist. Dadurch kann eine Vorlesung nie von mehreren Professoren gelesen werden.

- Bei einer Generalisierungsbeziehung muss jedes Entity eines Untertyps auch in seinen Obertypen enthalten sein.

- Es wurde explizit eine Domäne für jedes Attribut festgelegt. Damit kann z.B. ausgedrückt werden, dass eine Matrikelnummer (*MatrNr*) aus maximal fünf Ziffern besteht. Das Typkonzept in relationalen Datenbanken ist jedoch recht einfach. Es ist beispielsweise durchaus möglich, Personalnummern mit Vorlesungsnummern zu vergleichen, obwohl dieser Vergleich keinen Sinn macht.

5.1 Referentielle Integrität

Die Attributwerte eines Schlüssels identifizieren ein Tupel eindeutig innerhalb einer Relation. Verwendet man den Schlüssel einer Relation als Attribute einer anderen Relation, so spricht man von einem *Fremdschlüssel*. Ein solcher Fremdschlüssel ist beispielsweise das Attribut *gelesenVon* der Relation *Vorlesungen*. Ein Wert des Attributes *gelesenVon verweist* auf einen Datensatz in *Professoren*.

Seien R und S zwei Relationen mit den Schemata R und S. Sei κ Primärschlüssel von R. Dann ist $\alpha \subset \mathcal{S}$ ein Fremdschlüssel, wenn für alle Tupel $s \in S$ gilt:

1. $s.\alpha$ 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 Eigenschaften wird *referentielle Integrität* genannt.

Der Fremdschlüssel (hier α genannt) enthält also die gleiche Anzahl von Attributen wie der Primärschlüssel (κ genannt) der Relation, auf die der Fremdschlüssel verweist. Die Attribute haben auch jeweils dieselbe Bedeutung, obwohl sie oftmals umbenannt werden, um entweder Konflikte zu vermeiden oder den Attributen mnemonischere Namen zu geben. Ein Beispiel dafür ist das Attribut *Boss* in der Relation *Assistenten*, das *PersNr*-Werte von *Professoren* annimmt. Hier hätte man den Originalnamen gar nicht verwenden können, weil *Assistenten* auch eine *PersNr* haben. In der Relation *hören* wurden demgegenüber die Fremdschlüssel genauso benannt wie die Primärschlüssel der referenzierten Relationen – *MatrNr* verweist auf *Studenten* und *VorlNr* auf *Vorlesungen*.

Ohne eine Überprüfung der referentiellen Integrität kann man leicht einen inkonsistenten Zustand der Datenbasis erzeugen:

insert into Vorlesungen
values (5100, 'Nihilismus', 40, 007);

Die Vorlesung „Nihilismus" wird dann von jemandem mit der nicht existenten Personalnummer 007 gehalten. Einen solchen Verweis auf ein undefiniertes Objekt wird „Dangling Reference" genannt. In der konzeptuellen Modellierung spielte referentielle Integrität noch keine Rolle, da davon ausgegangen wurde, dass eine Beziehung grundsätzlich ihre zugehörigen Entities verband.

5.2 Gewährleistung referentieller Integrität

Für jede Veränderung der Datenbasis soll sichergestellt sein, dass nicht versehentlich „Dangling References" eingebaut werden. Wenn R und S Relationen, r und s Tupel, κ Primärschlüssel von R und α Fremdschlüssel auf R in S ist, muss also die folgende Bedingung gelten:

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

Erlaubte Änderungen sind also:

1. Einfügen von s in S, wenn $s.\alpha \in \Pi_\kappa(R)$, d.h. der Fremdschlüssel α verweist auf ein existierendes Tupel in R
2. Verändern eines Attributwertes $w = s.\alpha$ zu w', wenn $w' \in \Pi_\kappa(R)$ (wie bei 1)
3. Verändern von $r.\kappa$ in R, wenn $\sigma_{\alpha=r.\kappa}(S) = \emptyset$, d.h. es existieren keine Verweise auf r
4. Löschen von r in R, wenn $\sigma_{\alpha=r.\kappa}(S) = \emptyset$ (wie bei 3)

Sollten die Bedingungen nicht erfüllt sein, muss die Änderungsoperation (zumindest bei Transaktionsende, siehe Kapitel 9) rückgängig gemacht werden.

5.3 Referentielle Integrität in SQL

Zur Einhaltung der referentiellen Integrität gibt es für jeden der drei Schlüsselbegriffe eine Beschreibungsmöglichkeit:

- Ein Schlüssel(-kandidat) wird durch die Angabe von **unique** gekennzeichnet.
- Der Primärschlüssel wird mit **primary key** markiert. Die Attribute des Primärschlüssels sind automatisch als **not null** spezifiziert und müssen daher alle einen Wert haben.
- Ein Fremdschlüssel heißt **foreign key**. Fremdschlüssel können auch undefiniert, d.h. **null** sein, falls nicht explizit **not null** angegeben wurde. Ein **unique foreign key** modelliert eine 1:1-Beziehung. Wird ein Tupel verändert oder eingefügt, müssen die darin enthaltenen Fremdschlüssel gemäß Abschnitt 5.2 definiert sein.

Zusätzlich kann noch das Verhalten bei Änderungen an Verweisen oder referenzierten Daten festgelegt werden.[1] Es gibt drei Möglichkeiten. Zu deren Demonstration gehen wir wieder, wie im letzten Abschnitt, von den abstrakten Relationen R und S aus. R enthält den Primärschlüssel κ, S den Fremdschlüssel α. Zur Vereinfachung gehen wir davon aus, dass der Primärschlüssel nur aus einem Attribut vom Typ **integer** besteht. Eine entsprechende Tabellendefinition in SQL hätte folgende Form:

[1] Eine Schlüsselbedingung wird meistens durch das Anlegen einer Indexstruktur auf das Attribut erzwungen. Indexstrukturen werden in Kapitel 7 vorgestellt. Mit ihnen kann effizient festgestellt werden, ob ein Schlüsselwert vorhanden ist und nicht nochmal eingefügt werden darf (für **unique**) oder referenziert werden kann (als Fremdschlüssel).

create table R
(κ **integer primary key**,
...);

create table S
(...,
α **integer references** R);

In diesem Fall, wo außer den Schlüsselbeziehungen keine weiteren Angaben gemacht werden, ist es nicht möglich, noch von S referenzierte Tupel in R zu löschen oder zu verändern. Änderungsoperationen der Art, wie sie in Abbildung 5.1 angegeben sind, werden zurückgewiesen.

Dort wird auch die zweite Möglichkeit demonstriert. Wird ein Fremdschlüssel mit einer **cascade**-Angabe angelegt, werden Veränderungen des zugehörigen Primärschlüssels propagiert. Abbildung 5.1a) zeigt den **update**-Fall. Wird in der Tabelle R der Wert κ_1 zu κ_1' geändert, verursacht das Kaskadieren die gleiche Änderung in der Tabelle S. Auf diese Weise referenzieren die Fremdschlüssel in S auch nach der Operation noch dieselben Tupel in R. Analog demonstriert Abbildung 5.1b) das kaskadierende Löschen: κ_1 wird vom Benutzer durch die **delete**-Anweisung in R gelöscht und anschließend vom DBMS aufgrund der Integritätsbedingung auch in S.

Kaskadierendes Löschen ist mit Vorsicht zu genießen: Nehmen wir an, wir hätten unklugerweise festgelegt, dass im Universitätsbeispiel der Fremdschlüssel *gelesenVon* die Tupel in *Professoren* mit **on delete cascade** referenziert. Weiterhin soll auch *VorlNr* in *hören* kaskadierend gelöscht werden. Abbildung 5.3 zeigt, wie eine einzige Löschoperation viele weitere nach sich zieht. Die Linien stellen dabei die Beziehungen zwischen den Tupeln dar, hier repräsentiert durch die entsprechenden Namen. Die ganze Information im umrahmten Bereich wäre nach Ausführung des **delete**-Befehls, der das Tupel namens „Sokrates" aus der Relation *Professoren* löscht, verloren.

Alternativ kann als dritte Möglichkeit der Fremdschlüssel auf einen Nullwert gesetzt werden. Das wird in Abbildung 5.2 vorgeführt. Ist der Fremdschlüssel α mit **on update set null** definiert, wird nach Ausführung des **update**-Befehls der vorher bestehende Verweis auf κ_1 auf einen Nullwert gesetzt. Analog arbeitet **on delete set null**.

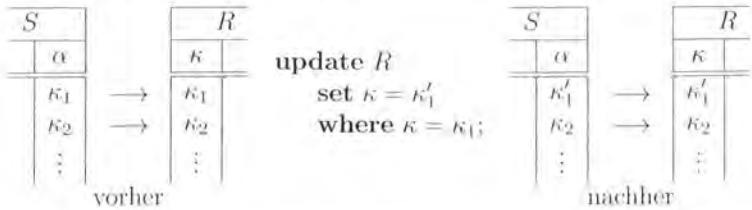
5.4 Überprüfung statischer Integritätsbedingungen

Statische Integritätsbedingungen werden in SQL durch eine **check**-Anweisung gefolgt von einer Bedingung implementiert. Dabei werden Änderungsoperationen an einer Tabelle zurückgewiesen, wenn die Bedingung zu **false** auswertet. Die typischsten Anwendungen für **check** sind Bereichseinschränkungen und die Realisierung von Aufzählungstypen. Da beliebige Bedingungen erlaubt sind – auch Unteranfragen – sind aber auch komplexere Anwendungen denkbar.

Ein Beispiel für eine Bereichseinschränkung wäre z.B. die Bedingung, dass Studenten maximal 13 Semester studieren dürfen:

... **check** Semester **between** 1 **and** 13 ...

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



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

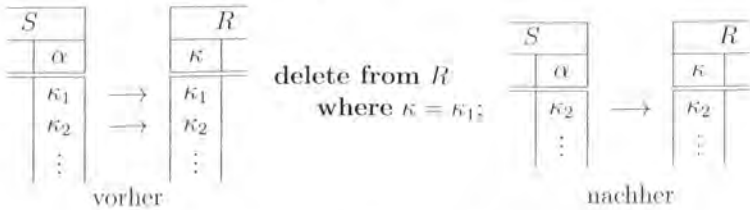
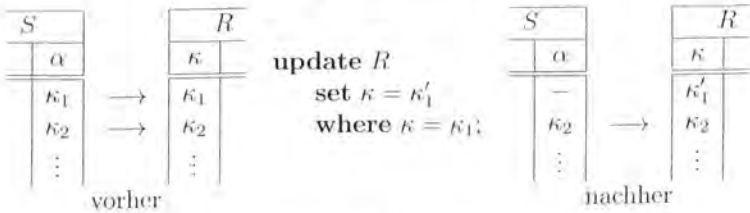


Abbildung 5.1: Referentielle Integrität durch Kaskadieren

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



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

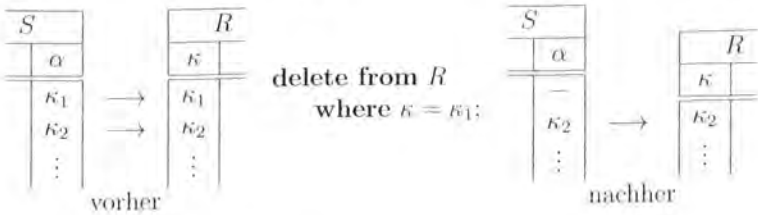
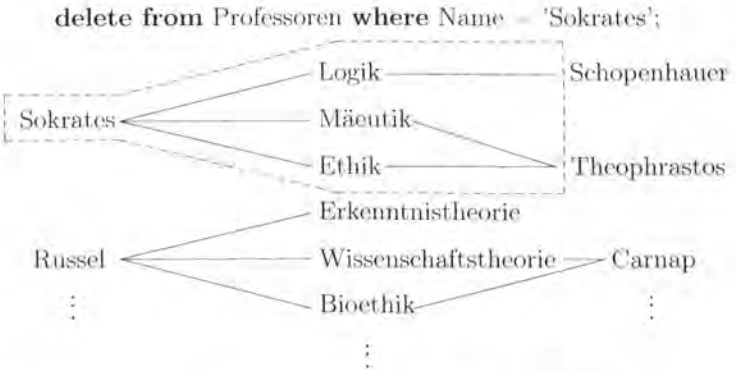


Abbildung 5.2: Referentielle Integrität durch Nullsetzen




```
create table Vorlesungen                create table hören
    ( ...,                                  ( ...,
      gelesenVon integer                      VorlNr integer
                 references Professoren                references Vorlesungen
                 on delete cascade);                   on delete cascade);
```

Abbildung 5.3: Kaskadierende Löschoperationen

Das Universitätsschema enthält auch Kandidaten für Aufzählungstypen, nämlich die Prüfungsnoten und die Ränge der Professoren. Der Rang kann nur drei unterschiedliche Werte annehmen:

... **check** Rang **in** ('C2', 'C3', 'C4') ...

Um die eingangs angegebene Definition der referentiellen Integrität zu erfüllen, muss ein Fremdschlüssel aus mehreren Attributen $S_1, S_2, \ldots$ entweder nur komplett **null** oder vollständig definiert sein. Mit **check** lässt sich das wie folgt erreichen:

... **check** ((S_1 **is null and** S_2 **is null and** ...) **or**
(S_1 **is not null and** S_2 **is not null and** ...))

Im Gegensatz zu **where**-Bedingungen gelten **check**-Bedingungen auch als erfüllt, wenn sie nach den Regeln aus Abschnitt 4.13, z.B. durch einen Nullwert, zu **unknown** auswerten. Hier ist also Vorsicht geboten.

5.5 Das Universitätsschema mit Integritätsbedingungen

Abbildung 5.4 zeigt das Schema mit der Erweiterung um statische Integritätsbedingungen. Integritätsbedingungen werden in der Tabellendefinition angegeben. Bezieht sich die Integritätsbedingung nur auf ein Attribut, kann sie direkt hinter ihrer Definition stehen. Da *MatrNr* z.B. das einzige Attribut ist, das zum Primärschlüssel gehört, kann **primary key** direkt hinter den Attributtyp geschrieben werden. Ebenso kann die Fremdschlüsseleigenschaft von *gelesenVon* direkt durch Angabe des

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

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

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

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

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

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

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

Abbildung 5.4: Das vollständige Universitätsschema mit Integritätsbedingungen

referenzierten Attributes festgelegt werden. Alternativ können Integritätsbedingungen jedoch immer auch unterhalb der Attributdefinitionen angefügt werden. Das ist bei den zusammengesetzten Schlüsseln notwendig. Da das Universitätsschema keine zusammengesetzten Fremdschlüssel besitzt, ist zur Demonstration aber auch die Fremdschlüsseleigenschaft von *Boss* separat aufgeführt.

Wird keine zusätzliche Angabe gemacht, darf ein Tupel nicht gelöscht werden, wenn noch ein Fremdschlüssel auf dieses Tupel verweist. Ein Tupel aus *Professoren* könnte so z.B. nicht gelöscht werden, solange noch zugehörige Tupel in *Assistenten* existieren.

Wird ein(e) Professor(in) gelöscht, dann sorgt die **set null** Bedingung dafür, dass *Boss* bei den zugehörigen Assistenten auf „unbekannt" gesetzt wird. Besteht noch ein Eintrag in *prüfen* zu einer Vorlesung, kann diese nicht entfernt werden. Beim Löschen von Studenten hingegen werden alle zugehörigen Einträge in *prüfen* und *hören* entfernt.

5.6 Komplexere Integritätsbedingungen

Gemäß dem SQL-Standard sind auch komplexere Integritätsbedingungen, die sich auf mehrere Relationen beziehen, möglich. In gewisser Weise stellen natürlich die *foreign key*-Klauseln schon solche Integritätsbedingungen dar, da sie sich immer auf zwei Relationen beziehen. Ein allgemeineres Beispiel einer Integritätsbedingung über mehrere Relationen ist nachfolgend für die Relation *prüfen* formuliert:

```
create table prüfen
    ( MatrNr      integer references Studenten on delete cascade,
      VorlNr      integer references Vorlesungen,
      PersNr      integer references Professoren on delete set null,
      Note        numeric(2,1) check (Note between 0.7 and 5.0),
      primary key (MatrNr, VorlNr)
      constraint VorherHören
          check (exists (select *
                         from hören h
                         where h.VorlNr = prüfen.VorlNr and
                               h.MatrNr = prüfen.MatrNr))
    );
```

Die Integritätsbedingung *VorherHören*[2] garantiert, dass *Studenten* sich nur über solche *Vorlesungen* prüfen lassen können, die sie auch gehört haben. Bei jeder Änderungs- und Einfügeoperation wird der **check** der Integritätsbedingung *VorherHören* „angeworfen" und die Operation nur dann ausgeführt, wenn der **check** den Wert *true* liefert. Das ist in unserem Beispiel dann der Fall, wenn man in der *hören*-Relation ein korrespondierendes Tupel findet.

Man hätte die Integritätsbedingung *VorherHören* auch als Fremdschlüssel-Eigenschaft formulieren können:

[2] Man kann Integritätsbedingungen generell einen Namen, wie hier geschehen, geben. Dies hat den Vorteil, dass man solche Integritätsbedingungen auch nachträglich wieder löschen kann – falls sie sich als zu stringent oder zu ineffizient herausstellen.

... **foreign key** (MatrNr, VorlNr) **references** hören ...

Man könnte auch die Prüfer einschränken, so dass *Professoren* nur selbst gelesene *Vorlesungen* prüfen. Dies sei den Lesern als Übung empfohlen. Außerdem sei den Lesern empfohlen, sich klar zu machen, welche Integritätsbedingungen man auch als *foreign key*-Bedingungen formulieren kann und welche Unterschiede (hinsichtlich späterer Änderungsoperationen) existieren.

Leider werden diese ausdruckskräftigen Integritätsklauseln, die sich über mehrere Relationen erstrecken, von den kommerziellen Datenbanksystemen dezeit kaum unterstützt. Das mag auch daran liegen, dass die Überprüfung u.U. sehr aufwendig ist. Deshalb muss man sich mit Triggern, die als nächstes behandelt werden, „behelfen".

5.7 Trigger

Der allgemeinste Konsistenzsicherungsmechanismus ist der sogenannte *Trigger*. Trigger sind leider noch nicht im SQL-92 Standard, sondern erst im SQL:1999-Standard enthalten, so dass die Datenbanksysteme unterschiedliche Syntax haben. Die Notation des ersten Beispiels ist an Oracle angelehnt. Ein Trigger ist eine benutzerdefinierte Prozedur, die automatisch bei Erfüllung einer bestimmten Bedingung vom DBMS gestartet wird. Sie kann nicht nur Überprüfungs-, sondern auch Berechnungsfunktionen übernehmen. Denkbar sind z.B. Trigger, die Statistiken aktuell halten oder die Werte abgeleiteter Spalten berechnen.

Durch den folgenden Trigger soll beispielsweise verhindert werden, dass Professoren einen Rang degradiert werden können:

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

Dieser Trigger besteht aus vier Teilen:

1. der **create trigger** Anweisung, gefolgt von einem Namen,
2. der Definition des Auslösers, in diesem Fall bevor eine Änderungsoperation (**before update on**) auf einer Zeile (**for each row**) der Tabelle *Professoren* ausgeführt werden kann,

3. einer einschränkenden Bedingung (**when**) und

4. einer Prozedurdefinition in der Oracle-proprietären Syntax.

In der Prozedurdefinition bezieht sich *old* auf das noch unveränderte Tupel (den Originalzustand), *new* enthält bereits die Veränderungen durch die Operation.

Die Triggersyntax des IBM-Datenbanksystem DB2 ähnelt der des SQL:1999-Standards. Der obige Trigger könnte in dieser Syntax wie folgt formuliert werden:

```
create trigger keineDegradierung
no cascade
before update of Rang on Professoren
referencing old as alterZustand
            new as neuerZustand
for each row
mode DB2SQL
when (alterZustand.Rang is not null)
set neuerZustand.Rang = case
        when neuerZustand.Rang is null then alterZustand.Rang
        when neuerZustand.Rang < 'C2' then alterZustand.Rang
        when neuerZustand.Rang > 'C4' then alterZustand.Rang
        when neuerZustand.Rang < alterZustand.Rang then alterZustand.Rang
        else neuerZustand.Rang
        end;
```

Das Schlüsselwort **no cascade** schließt das mehrfache „Feuern" des Triggers aus. Bei den **before**-Triggern ist es zwingend vorgeschrieben, da der Trigger *vor* der Operation ausgeführt wird. Trigger können aber auch als **after**-Trigger angegeben werden, so dass sie nach der auslösenden Operation angestoßen werden. Die möglichen Ereignisse zur Aktivierung eines Triggers sind ein Update eines Attributs (wie in unserem Beispiel), das Einfügen eines neuen Tupels (**insert**) oder das Löschen (**delete**). Ein Trigger bezieht sich immer auf *eine* Relation, die in der **on**-Klausel angegeben wird. Bei **update**-Triggern kann man dem alten bzw. dem neuen Zustand des Tupels einen Variablennamen (im Beispiel *neuerZustand* bzw. *alterZustand*) zuweisen. Bei **insert**-Triggern kann man sich natürlich nur auf das neue Tupel und bei **delete**-Triggern nur auf das alte Tupel beziehen. In unserem Beispiel haben wir einen **row**-Level-Trigger definiert, der für jedes geänderte Tupel separat angestoßen wird. Man kann auch **statement**-Level-Trigger definieren, die für den gesamten ausgeführten SQL-Ausdruck nur einmal angestoßen werden. Die Bedingung zur Ausführung des Triggers wird in der **when**-Klausel angegeben. Danach folgt ein SQL-Ausdruck, der in unserem Fall aus einer **set**-Zuweisung besteht. Der zugewiesene Wert wird mittels eines **case**-Ausdrucks bestimmt.[3]

In unserem (sehr einfachen) Beispiel haben wir Trigger für die Konsistenzerhaltung einer Relation angewendet. Trigger sind aber vielseitig einsetzbar, um z.B. Benutzer oder andere Systeme über bestimmte Ereignisse zu informieren. Man denke

[3]Dieser **case**-Ausdruck ist etwas „übermotiviert", da er auch die Zuweisung ungültiger Werte (wie 'C0' oder 'C5') abfängt. Diese werden ja eigentlich durch die **check**-Bedingung (siehe Abbildung 5.4) schon ausgeschlossen – wir haben eine derartige Konsistenzverletzung der Vollständigkeit halber auch im Trigger abgefangen.

etwa an einen Trigger, der dafür sorgt, dass Produkte, deren Lagerbestand sich dem Ende neigen, automatisch nachbestellt werden. Diese Anwendungen bezeichnet man auch als *aktive Datenbanken*, da ein Teil der Anwendungslogik im Datenbanksystem als Trigger enthalten ist. Bei der Realisierung solcher Systeme sollte man aber sehr sorgfältig abwägen, welche Funktionen man als Trigger „im Hintergrund" ablaufen lassen will. Das Zusammenspiel vieler Trigger, die sich gegenseitig aktivieren, wird sehr schnell unübersichtlich und ist daher recht fehleranfällig.

5.8 Temporale Daten

Mittlerweile enthält der SQL-Standard auch Unterstützung für temporale Daten, d.h., es gibt explizite Datenmodell-Konzepte, um das zeitliche Verhalten der Daten zu erfassen. Dabei unterscheidet man zwei Funktionalitäten: (1) die automatische Versionierung einer Datenbank-Relation, so dass alle Änderungen automatisch als Versionen protokolliert werden. (2) Zum anderen gibt es auch Unterstützung, um anwendungsspezifische Gültigkeits-Zeiträume zu erfassen.

5.8.1 System-versionierte Relationen

Hierbei wird jede Änderung eines Tupels der temporalen Relation dazu führen, dass eine neue Version dieses Tupels erzeugt wird. Man spricht in diesem Zusammenhang auch von *append only*-Datenbanken, da Updates nicht „*in-place*" im Original-Tupel, sondern in einem neu generierten Tupel durchgeführt werden. Bei einer Änderung wird also automatisch das derzeit gültige Tupel zu einer alten Version „degradiert", dessen Gültigkeit mit dem Änderungszeitpunkt, also der sogenannten Transaktionszeit (*transaction time*), endet. Daraus resultiert auch der häufig verwendete Name dieser Versionierung: *transaction time-Versionierung*.

Wollen wir beispielsweise die Kontinuität politischer Grundsatzentscheidungen bezüglich der Studiengebühren in den einzelnen Bundesländern nachvollziehbar protokollieren, so könnte man folgende Relation definieren:

```
create table Studiengebühren
  ( Bundesland varchar(30) not null,
    Beitrag integer not null,
    Beginn date not null generated always as row start,
    Ende date not null generated always as row end,
    period for system_time (Beginn, Ende),
    primary key(Bundesland)
  ) with system versioning;
```

Bei jeder Änderung eines Tupels in dieser Relation wird also eine neue Version erzeugt, wie geschehen als Ministerpräsident Stoiber am 1.4.2007 die Gebühr in Bayern einführte und als MP Seehofer sie am 1.10.2013 wieder aufhob. Die Gültigkeitsperiode eines Tupels wird durch ein halboffenes Intervall angegeben, so dass es ab und inklusive dem *Beginn*-Zeitpunkt bis exklusive dem *Ende*-Zeitpunkt gültig

war. Derzeit gültige Tupel haben ein fiktives (maximal weit in der Zukunft liegendes) Gültigkeitsende, beispielsweise im Jahr 9999. Der Zustand der System-versionierten Relation sieht dann wie folgt aus:

Studiengebühren			
Bundesland	Beitrag	Beginn	Ende
Thüringen	0	1990.10.03	9999.12.31
Bayern	0	1990.10.03	2007.04.01
Bayern	500	2007.04.01	2013.10.01
Bayern	0	2013.10.01	9999.12.31
...	...	...	...

In der Schemadefinition wurde *Bundesland* als Primärschlüssel angegeben, was angesichts der drei Tupel für Bayern ungewöhnlich erscheint. Hierbei ist aber zu beachten, dass es zu jedem Zeitpunkt nur ein gültiges Tupel pro Bundesland gibt daraus resultiert die Korrektheit der Schlüsseldefinition.

„Normale" SQL-Anfragen beziehen sich immer nur auf die derzeit *gültigen* Tupel der Relation, so dass die Anfrage

select Beitrag
from Studiengebühren
where Bundesland = 'Bayern'

den Wert 0 ergibt. Bayerische Studenten könnten ihre im Sommersemester 2011 entrichteten Beiträge aus dem Ergebnis der folgenden Anfrage ermitteln:

select Beitrag
from Studiengebühren
where Bundesland = 'Bayern' **and system_time as of date**('2011.04.01')

Es ist auch über die **system_time between ... and ...**-Klausel möglich, auf das bzw. die während einer bestimmten Zeitperiode gültig gewesene/n Tupel zuzugreifen.

5.8.2 Temporale Daten nach Anwendungszeit

In manchen Anwendungen will man die Gültigkeitsintervalle explizit verwalten evtl. auch rückwirkend noch ändern. Deshalb spricht man von temporalen Daten gemäß der Anwendungszeit (*application time* oder auch *valid time*). Als Beispiel definieren wir für die Universitäts-Datenbank eine Relation *TutorFürVorlesung*, in der die Assistenten temporär als Tutoren für Vorlesungen eingeplant werden:

create table TutorFürVorlesung
(AssiPersNr **integer not null references Assistenten**,
BetreuteVorlNr **integer not null references Vorlesungen**,
Von **date not null**,
Bis **date not null**,
period for Zeitraum(Von,Bis),
primary key(AssiPersNr,Zeitraum **without overlaps**)
);

Eine Ausprägung dieser Relation könnte dann wie folgt aussehen:

TutorFürVorlesung			
AssiPersNr	BetreuteVorlNr	Von	Bis
3002	5049	2012.10.01	2013.04.01
3002	4052	2013.04.01	2013.10.01
3003	5049	2012.04.01	2013.10.01
...	...	...	...

Hier ist also modelliert, dass der Assistent mit der PersNr 3002 (Platon) im Wintersemester 2012/13 die Vorlesung 5049 (Mäeutik) betreute. Da Mäeutik so beliebt ist, hat auch sein Kollege Aristoteles (PersNr 3003) diese Vorlesung im Sommersemester 2012, im Wintersemester 2012/13 und im Sommersemester 2013 als Tutor betreut, wohingegen Platon im Sommersemester 2013 für die Vorlesung 4052 (Logik) als Tutor im Einsatz war.

Der angegebene Primärschlüssel besteht aus *AssiPersNr* und *Zeitraum*, so dass ein Assistent zu jedem Zeitpunkt maximal eine Vorlesung betreuen darf – was durch die nicht-überlappende (**without overlaps**) *Zeitraum*-Definition angegeben wird.

Wenn man diese Tutoren-Zuordnung für Aristoteles dergestalt ändern will, dass er während des Wintersemesters 2012/13 doch die Vorlesung 5041 (Ethik) statt der Mäeutik betreuen soll, so geschieht das mit dieser Update-Anweisung:

```
update TutorFürVorlesung for portion of Zeitraum
    from date('2012.10.01') to date('2013.04.01')
  set VorlNr = 5041
where AssiPersNr = 3003
```

Der resultierende Zustand der Relation *TutorFürVorlesung* weist jetzt zwei zusätzliche Tupel auf, da der ursprünglich über drei Semester gehende Zeitraum für Aristoteles automatisch drei-geteilt wurde:

TutorFürVorlesung			
AssiPersNr	BetreuteVorlNr	Von	Bis
3002	5049	2012.10.01	2013.04.01
3002	4052	2013.04.01	2013.10.01
3003	5049	2012.04.01	2012.10.01
3003	5049	2012.10.01	2013.04.01
3003	5049	2013.04.01	2013.10.01
...	...	...	...

In umgekehrter Richtung kann es auch vorkommen, dass nach einem Update oder einer Einfügung nahtlos aneinander anschließende Gütligkeitsintervalle wieder zusammengefasst werden können, was man im Englischen als *coalescing* bezeichnet.

Für SQL-Anfragen auf den temporalen Daten stehen u.a. folgende Prädikate zur Verfügung: **contains**, **precedes**, **succeeds**, **immediately precedes/succeeds**, **overlaps**.

Wenn man in einer Relation sowohl die Sytemzeit-Versionierung als auch die Anwendungs-spezifischen Zeitintervalle verwendet, spricht man von *bitemporalen Daten*.

5.9 Übungen

5.1 Vollziehen Sie konkret am Universitätsbeispiel nach, welche Integritätsbedingungen bereits in der ER-Modellierung (Abbildung 2.7) vorhanden sind und welche erst später, in Abbildung 5.4, festgelegt wurden.

5.2 Beschreiben Sie die Auswirkungen der folgenden Operationen auf der Beispielausprägung aus Abbildung 3.8 mit dem Schema aus Abbildung 5.4.

- **delete from** Vorlesungen **where** Titel = 'Ethik';
- **insert into** prüfen **values** (24002, 5001, 2138, 2.0);
- **insert into** prüfen **values** (28106, 5001, 2127, 4.3);
- **drop table** Studenten;

5.3 Welches Modellierungskonzept würden Sie mit Hilfe von kaskadierendem Löschen realisieren?

5.4 Geben Sie die **create table**-Befehle inklusive Integritätsbedingungen an, um das in Aufgabe 3.1 gewonnene relationale Schema zu implementieren.

5.5 Geben Sie die **create table**-Befehle an, um das in Aufgabe 3.3 gewonnene relationale Schema für ein Wahlinformationssystem zu implementieren. Setzen Sie notwendige Integritätsbedingungen mit um.

5.6 Da die Generalisierung in den meisten relationalen Systemen nicht unterstützt wird, könnte man auf die Idee kommen, die Vererbungshierarchie von *Angestellte* zu *Professoren* und *Assistenten* gemäß Smith und Smith (1977) durch Redundanz zu modellieren:

Angestellte			
PersNr	Name	Gehalt	Typ
2125	Sokrates	90000	Professoren
3002	Platon	50000	Assistenten
1001	Maier	130000	
...	...	...	...

Professoren				
PersNr	Name	Gehalt	Rang	Raum
2125	Sokrates	90000	C4	226
...	...	...	...	...

Assistenten				
PersNr	Name	Gehalt	Fachgebiet	Boss
3002	Platon	50000	Ideenlehre	2125
...	...	...	...	...

Hierbei werden also beispielsweise Professoren sowohl in der Relation *Professoren* als auch in der Relation *Angestellte* eingetragen. Die Attribute der

Relation *Angestellte* werden redundant auch in der Relation *Professoren* gespeichert.

Es gilt nun, diese Redundanz zu kontrollieren. Können Sie Trigger schreiben, die Updates entsprechend propagieren? Wenn also beispielsweise das Gehalt von Sokrates in der Relation *Professoren* geändert wird, soll diese Änderung automatisch (über einen Trigger) auf die Relation *Angestellte* propagiert werden. Analog muss aber eine Gehaltsänderung von Sokrates, die auf der Relation *Angestellte* durchgeführt wurde, auf die Relation *Professoren* propagiert werden.

Achten Sie darauf, dass Ihre Trigger terminieren!

Nach dem Kenntnisstand der Autoren, ist es in Oracle – aufgrund einer Einschränkung der Triggerfunktionalität – nicht möglich, diese Trigger zu realisieren. Wenn Sie es doch schaffen, lassen Sie es uns bitte wissen. In DB2, beispielsweise, ist es möglich; aber wie?

5.7 Trigger werden oft auch eingesetzt, um replizierte Daten konsistent zu halten. Man denke etwa an ein Attribut *AnzahlHörer*, das der Relation *Vorlesungen* zugeordnet ist. Der Wert dieses Attributs könnte mittels eines Triggers auf der Relation *hören* aktuell gehalten werden. Realisieren Sie die notwendigen Trigger im Datenbanksystem „Ihrer Wahl".

5.10 Literatur

Die Bedeutung von referentieller Integrität speziell bei relationalen Datenbanksystemen wurde von Date (1981) beschrieben. Melton und Simon (1993) beschreiben die Integritätsbedingungen in SQL 2. Das Triggerkonzept von Oracle wird von Bobrowski (1992) erläutert.

Für dynamische Integritätsbedingungen wurden hier nur Trigger vorgestellt. Eine formale Beschreibungsmöglichkeit bietet die temporale Logik, wie es beispielsweise von Lipeck und Saake (1987) diskutiert wird. Auch May und Ludäscher (2002) verwenden einen logikbasierten Formalismus für die Beschreibung referentieller Integritätsaktionen. Türker und Gertz (2001) beschreiben die erweiterten Konzepte zur Modellierung semantischer Integritätsregeln in SQL:99.

Casanova und Tucherman (1988) beschreiben, wie man referentielle Integrität mit Hilfe eines Monitors überwacht.

Das Trigger-Konzept ist eine Vorform der sogenannten *aktiven* Datenbanken. Projekte in diesem Bereich sind beispielsweise SAMOS [Gatziu, Geppert und Dittrich (1991)] und REACH [Buchmann et al. (1995)]. Gertz und Lipeck (1996) haben die Nutzung von Triggern zur Gewährleistung dynamischer Integritätsbedingungen untersucht. Behrend, Manthey und Pieper (2001) untersuchen die erweiterten Integritätsbedingungen von SQL-3 (bzw. SQL:1999), die aber bedauerlicherweise noch nicht in den kommerziellen Produkten enthalten sind.

Temporale Datenbankunterstützung wurde im SQL Standard 2011 eingeführt. Die hier vorgestellten Konzepte wurden aber noch nicht von allen kommerziellen Systemen vollständig umgesetzt. Petkovic (2013) gibt einen schönen Überblick über

die Funktionalität und ergänzt, dass die Unterstützung im IBM Datenbanksystem DB2 am weitesten fortgeschritten ist. Eine weitere englischsprachige Abhandlung der Modellierungskonzepte wurde von Kulkarni und Michels (2012) verfasst. Auch Böhlen et al. (2009) geben eine Übersicht über die SQL-Unterstützung – Koautor dieses Beitrag ist R. Snodgrass, der die Forschung im Bereich temporaler Daten geprägt hat.

Finis et al. (2013) haben ein Verfahren für die effiziente Versionierung hierarchischer Datenstrukturen in Hauptspeicher-Datenbanken entwickelt – u.a. für das SAP Hauptspeicher-Datenbanksystem HANA. Im Kontext von SAP HANA wurde auch der sogenannte Timeline-Index von Kaufmann et al. (2013) entwickelt, der die zeitliche Evolution der Daten entlang der Transaktionszeit indexiert.

6. Relationale Entwurfstheorie

In den vorangegangenen Kapiteln haben wir uns schon mit dem methodischen Entwurf einer Datenbankanwendung beschäftigt. Wir haben dabei den schrittweisen top-down-Entwurf kennengelernt, wobei zunächst ein Pflichtenheft, dann ein konzeptueller Entity-Relationship-Entwurf und schließlich ein relationales Schema erstellt wurden.

In diesem Kapitel beschäftigen wir uns sozusagen mit der konzeptuellen Feinabstimmung des erstellten relationalen Schemas auf der Grundlage formaler Methoden. Die Basis für diesen Feinentwurf bilden funktionale Abhängigkeiten, die eine Verallgemeinerung des – zumindest informell – schon eingeführten Schlüsselbegriffs darstellen. Weiterhin werden mehrwertige Abhängigkeiten untersucht, die ihrerseits eine Verallgemeinerung der funktionalen Abhängigkeiten darstellen.

Basierend auf diesen Abhängigkeiten werden Normalformen für Relationenschemata definiert. Die Normalformen dienen dazu, die „Güte" eines Relationenschemas zu bewerten. Wenn für ein Relationenschema diese Normalformen nicht erfüllt sind, kann man es durch Anwendung entsprechender Normalisierungsalgorithmen in mehrere Schemata zerlegen, die dann die entsprechende Normalform erfüllen.

6.1 Funktionale Abhängigkeiten

Die Diskussion in diesem Kapitel bezieht sich (meistens) auf ein abstraktes relationales Datenbankschema bestehend aus n Relationenschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ mit möglichen – nicht näher bestimmten – Ausprägungen $R_1, \ldots, R_n$ oder ein Schema $\mathcal{R}$ mit 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 wie folgt dargestellt:

$$\alpha \to \beta$$

Hierbei repräsentieren die griechischen Buchstaben α und β jeweils Mengen von Attributen. Betrachten wir zunächst den Fall, dass die FD $\alpha \to \beta$ auf dem Relationenschema $\mathcal{R}$ definiert ist, d.h. α und β seien Teilmengen von $\mathcal{R}$. Dann sind nur solche Ausprägungen R zulässig, für die folgendes gilt: Für alle Paare von Tupeln $r, t \in R$ mit $r.\alpha = t.\alpha$ muss auch gelten $r.\beta = t.\beta$. Hierbei stellt $r.\alpha = t.\alpha$ eine Kurzform für $\forall A \in \alpha : r.A = t.A$ dar. Mit anderen Worten drückt die FD $\alpha \to \beta$ aus, dass wenn zwei Tupel gleiche Werte für alle Attribute in α haben, dann müssen auch ihre β-Werte (d.h. die Werte der Attribute in β) übereinstimmen. Wir sagen dann auch, dass die α-Werte die β-Werte funktional (d.h. eindeutig) bestimmen. Oder anders herum, dass die β-Werte funktional abhängig von den α-Werten sind. Man bezeichnet α auch als *Determinante* von β.

Wir wollen dieses sehr wichtige – und für die relationale Entwurfstheorie zentrale – Konzept der funktionalen Abhängigkeiten an einem abstrakten Beispiel erläutern. Dazu betrachten wir die Relation R mit dem Schema $\mathcal{R} = \{A, B, C, D\}$ und der funktionalen Abhängigkeit $\{A\} \rightarrow \{B\}$.

	R			
	A	B	C	D
t	a_4	b_2	c_4	d_3
p	a_1	b_1	c_1	d_1
q	a_1	b_1	c_1	d_2
r	a_2	b_2	c_3	d_2
s	a_3	b_2	c_4	d_3

Diese Relation erfüllt die FD $\{A\} \rightarrow \{B\}$, da es nur zwei Tupel p und q mit gleichem A-Attributwert gibt, nämlich $p.A = q.A = a_1$. Bei diesen beiden Tupeln p und q stimmt auch der Wert des Attributs B – nämlich $p.B = q.B = b_1$ – überein.

Weiterhin erfüllt die gezeigte Ausprägung R die funktionale Abhängigkeit $\{A\} \rightarrow \{C\}$, wie die Leser auf analoge Weise nachvollziehen können. Außerdem ist die funktionale Abhängigkeit $\{C, D\} \rightarrow \{B\}$ in der Relation R erfüllt. Nur die beiden Tupel s und t haben gleiche Werte für C *und* D – deshalb erfüllen alle anderen Tupel die funktionale Abhängigkeit automatisch. Da s und t auch den gleichen Wert für B haben, folgt daraus, dass $\{C, D\} \rightarrow \{B\}$ erfüllt ist.

Andererseits ist die funktionale Abhängigkeit $\{B\} \rightarrow \{C\}$ in der Relation R nicht erfüllt. Dazu betrachte man die beiden Tupel r und s mit $r.B = s.B$. Offensichtlich haben diese beiden Tupel unterschiedliche C-Werte, nämlich $r.C = c_3 \neq c_4 = s.C$.

Es soll an dieser Stelle nochmals betont werden, dass funktionale Abhängigkeiten eine semantische Konsistenzbedingung darstellen, die sich aus der jeweiligen Anwendungssemantik und *nicht* aus der derzeitigen zufälligen Relationenausprägung ergeben. Mit anderen Worten: Funktionale Abhängigkeiten stellen Konsistenzbedingungen dar, die zu allen Zeiten in jedem (gültigen) Datenbankzustand eingehalten werden müssen.

6.1.1 Konventionen zur Notation

In der Datenbank-Literatur hat sich vielfach die etwas „saloppe" Notation $CD \rightarrow A$ oder $C, D \rightarrow A$ anstatt der formal präzisen Notation $\{C, D\} \rightarrow \{A\}$ eingebürgert. Weiterhin steht $\alpha - A$ für $\alpha - \{A\}$, wenn α eine Attributmenge und A ein Attribut aus dieser Menge repräsentieren. Die Vereinigung zweier Attributmengen α und β wird einfach als $\alpha\beta$ notiert. Die abstrakten Attribute einer Attributmenge wie z.B. $\{A, B, C\}$ werden als ABC notiert.

6.1.2 Einhaltung einer funktionalen Abhängigkeit

Eine andere Charakterisierung für eine funktionale Abhängigkeit $\alpha \rightarrow \beta$ ist die folgende: Die FD $\alpha \rightarrow \beta$ ist in R erfüllt, wenn für jeden möglichen Wert c von α gilt, dass

$$\Pi_\beta(\sigma_{\alpha=c}(R))$$

höchstens ein Element enthält. Das obige ist eine etwas informelle aber anschauliche Formulierung: Unter einem Wert c von α verstehen wir natürlich ein Tupel $[c_1, \ldots, c_i] \in \mathbf{dom}(A_1) \times \cdots \times \mathbf{dom}(A_i)$, wenn $\alpha = \{A_1, \ldots, A_i\}$ gilt. Weiterhin steht dann der Ausdruck $\sigma_{\alpha=c}(R)$ für

$$\sigma_{A_1=c_1}(\cdots(\sigma_{A_i=c_i}(R))\cdots).$$

Die eben diskutierte Charakterisierung einer funktionalen Abhängigkeit liefert einen einfachen Algorithmus, mit dem festgestellt wird, ob eine gegebene Relation R die FD $\alpha \to \beta$ erfüllt:

- Eingabe: eine Relation R und eine FD $\alpha \to \beta$
- Ausgabe: *ja*, falls $\alpha \to \beta$ in R erfüllt ist; *nein* sonst
- *Einhaltung*$(R, \alpha \to \beta)$
  - sortiere R nach α-Werten
  - falls alle Gruppen bestehend aus Tupeln mit gleichen α-Werten auch gleiche β-Werte aufweisen: Ausgabe *ja*; sonst: Ausgabe *nein*

Die Laufzeit dieses Algorithmus wird natürlich durch die Sortierung dominiert. Somit hat der Algorithmus *Einhaltung* die Komplexität $O(n \log n)$.

Die Leser mögen ihn auf die oben angegebene Relation R anwenden um (nochmals) nachzuweisen, dass z.B. die FD $\{C, D\} \to \{B\}$ erfüllt ist.

Die funktionalen Abhängigkeiten, die von *jeder* Relationenausprägung automatisch immer erfüllt sind, nennt man *triviale* FDs. Man kann zeigen, dass nur FDs der Art

$$\alpha \to \beta \quad \text{mit} \quad \beta \subseteq \alpha$$

trivial sind (siehe Übungsaufgabe 6.5).

6.2 Schlüssel

Wie oben bereits erwähnt, stellen die funktionalen Abhängigkeiten eine Verallgemeinerung des Schlüsselbegriffs dar. Das wollen wir jetzt präzisieren.

In der Relation $\mathcal{R}$ ist $\alpha \subseteq \mathcal{R}$ ein *Superschlüssel*, falls gilt:

$$\alpha \to \mathcal{R}$$

D.h. α bestimmt alle anderen Attributwerte innerhalb der Relation $\mathcal{R}$. Wir nennen α in diesem Fall *Superschlüssel*, weil noch nichts darüber ausgesagt ist, ob α eine minimale Menge von Attributen enthält. Z.B. folgt aus der mengentheoretischen Definition des relationalen Modells automatisch (Mengen enthalten keine Duplikate):

$$\mathcal{R} \to \mathcal{R}$$

Also bildet die Menge aller Attribute einer Relation einen Superschlüssel.

Wir benötigen das Konzept der *vollen* funktionalen Abhängigkeiten um Schlüssel von Superschlüsseln abzugrenzen. β ist *voll funktional abhängig* von α – in Zeichen $\alpha \overset{\bullet}{\to} \beta$ – falls beide nachfolgenden Kriterien gelten:

1. $\alpha \rightarrow \beta$, d.h. β ist funktional abhängig von α und
2. α kann nicht mehr „verkleinert" werden, d.h.

$$\forall A \in \alpha : \alpha - \{A\} \not\rightarrow \beta$$

Es kann also kein Attribut mehr aus α entfernt werden, ohne die FD zu „zerstören".

Falls $\alpha \xrightarrow{\bullet} \mathcal{R}$ gilt, bezeichnet man α als Kandidatenschlüssel von $\mathcal{R}$. Im Allgemeinen wird einer der Kandidatenschlüssel als sogenannter *Primärschlüssel* ausgewählt. Diese Auswahl ist notwendig, da im relationalen Modell Verweise zwischen Tupeln unterschiedlicher Relationen über Fremdschlüssel realisiert werden. Man sollte darauf achten, dass für Fremdschlüssel immer derselbe Schlüssel verwendet wird – deshalb ist die Auszeichnung eines Kandidatenschlüssels als Primärschlüssel unbedingt notwendig.

Als Beispiel für die Bestimmung von Kandidatenschlüsseln betrachten wir folgende Relation (*EW* stehe für Einwohnerzahl und *BLand* für Bundesland):

Städte			
Name	BLand	Vorwahl	EW
Frankfurt	Hessen	069	650000
Frankfurt	Brandenburg	0335	84000
München	Bayern	089	1200000
Passau	Bayern	0851	50000
...	...	...	...

Wir gehen davon aus, dass Wohnorte innerhalb von Bundesländern eindeutig benannt sind. Die Kandidatenschlüssel für die Relation *Städte* sind:

- {Name, BLand}
- {Name, Vorwahl}

Man beachte, dass zwei (kleinere) Städte dieselbe Vorwahl haben können; deshalb bildet {*Vorwahl*} alleine keinen Schlüssel. Anders wäre das in einer Relation *Großstädte*, in der nur Großstädte mit exklusiver Vorwahl-Nummer abgespeichert wären.

6.3 Bestimmung funktionaler Abhängigkeiten

Es ist Aufgabe der Datenbankentwerfer, die funktionalen Abhängigkeiten aus der Anwendungssemantik zu bestimmen. Als Beispiel möge uns folgendes Relationenschema[1] dienen:

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

[1] Dieses Schema wird hier nur für die Demonstration funktionaler Abhängigkeiten verwendet. Es stellt in keiner Weise einen guten relationalen Entwurf dar – wie wir im nachfolgenden noch sehen werden.

Hierbei verstehen wir unter *Ort* den eindeutigen Erstwohnsitz der Professoren. Die *Landesregierung* ist die eine „tonangebende" Partei, die also den Ministerpräsidenten bzw. die Ministerpräsidentin stellt, so dass *Landesregierung* funktional abhängig von *BLand* ist. Weiterhin machen wir einige vereinfachende Annahmen: Orte sind innerhalb der Bundesländer (nach wie vor) eindeutig benannt. Die Postleitzahl (*PLZ*) ändert sich nicht innerhalb einer Straße, Städte und Straßen gehen nicht über Bundeslandgrenzen hinweg.

Beim Datenbankentwurf könnten dann folgende FDs bestimmt worden sein:

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

Die erste aufgeführte FD besagt, dass *PersNr* ein Kandidatenschlüssel der Relation *ProfessorenAdr* ist. In der vierten FD gehen wir von der oben beschriebenen vereinfachenden Annahme aus, dass die PLZ sich innerhalb einer Straße eines Orts nicht ändert.

Aus dieser vorgegebenen Menge von funktionalen Abhängigkeiten ergeben sich weitere Abhängigkeiten, die von jeder gültigen Relationenausprägung auch immer erfüllt sind. Beispiele hierfür sind:

- {Raum} → {PersNr, Name, Rang, Raum, Ort, Straße, PLZ, Vorwahl, BLand, EW, Landesregierung}
- {PLZ} → {Landesregierung}

Wir sagen, dass diese weiteren Abhängigkeiten aus den vorgegebenen FDs herleitbar sind. Im Allgemeinen sind wir bei einer gegebenen Menge F von FDs daran interessiert, die Menge F^+ aller daraus herleitbaren funktionalen Abhängigkeiten zu bestimmen. Diese Menge F^+ bezeichnet man als die *Hülle* (engl. *closure*) der Menge F. Die *Hülle* einer Menge von FDs kann durch Anwendung von Herleitungsregeln – auch *Inferenzregeln* genannt – bestimmt werden. Für die Herleitung der vollständigen Hülle reichen die drei nachfolgend aufgeführten *Armstrong-Axiome* als Inferenzregeln aus.

Entsprechend unserer Konvention bezeichnen α, β, γ und δ Teilmengen der Attribute aus $\mathcal{R}$.

- *Reflexivität*: Falls β eine Teilmenge von α ist ($\beta \subseteq \alpha$) dann gilt immer $\alpha \rightarrow \beta$. Insbesondere gilt also immer $\alpha \rightarrow \alpha$.
- *Verstärkung*: Falls $\alpha \rightarrow \beta$ gilt, dann gilt auch $\alpha\gamma \rightarrow \beta\gamma$. Hierbei stehe z.B. $\alpha\gamma$ für $\alpha \cup \gamma$.

- *Transitivität*: Falls $\alpha \to \beta$ und $\beta \to \gamma$ gilt, dann gilt auch $\alpha \to \gamma$.

Die Armstrong-Axiome sind *korrekt* (engl. *sound*) und *vollständig*. Die Korrektheit der Axiome besagt, dass sich mit Hilfe der Armstrong-Axiome aus einer Menge F von FDs nur solche weiteren FDs ableiten lassen, die von *jeder* Relationenausprägung erfüllt sind, für die F erfüllt ist. Die Vollständigkeit der Axiome besagt, dass sich *alle* FDs ableiten lassen, die durch F logisch impliziert werden. Man ist also in der Lage F^+ vollständig mittels der Armstrong-Axiome zu bestimmen.

Obwohl die Armstrong-Axiome vollständig sind, ist es für Herleitungsprozesse komfortabel, noch drei weitere Axiome hinzuzunehmen.

- *Vereinigungsregel*: Wenn $\alpha \to \beta$ und $\alpha \to \gamma$ gelten, dann gilt auch $\alpha \to \beta\gamma$.
- *Dekompositionsregel*: Wenn $\alpha \to \beta\gamma$ gilt, dann gelten auch $\alpha \to \beta$ und $\alpha \to \gamma$.
- *Pseudotransitivitätsregel*: Wenn $\alpha \to \beta$ und $\gamma\beta \to \delta$, dann gilt auch $\alpha\gamma \to \delta$.

Wir wollen mit Hilfe der Axiome nachweisen, dass die funktionale Abhängigkeit $\{PLZ\} \to \{Landesregierung\}$ in unserem Beispielschema gilt. Dazu wird zunächst die Dekompositonsregel angewendet um $\{PLZ\} \to \{BLand\}$ herzuleiten. Unter Anwendung der Transitivitätsregel ergibt sich dann hieraus und aus der gegebenen FD $\{BLand\} \to \{Landesregierung\}$ die FD $\{PLZ\} \to \{Landesregierung\}$.

Die Leser mögen herleiten, dass *Raum* ein Kandidatenschlüssel ist, d.h. dass $\{Raum\} \to \mathbf{sch}(Professoren)$ gilt.

Oftmals ist man nicht an der gesamten Hülle einer Menge von FDs interessiert, sondern nur an der Menge von Attributen α^+, die von α gemäß der Menge F von FDs funktional bestimmt werden. Diese Menge α^+ kann man mit folgendem Algorithmus herleiten:

- **Eingabe**: eine Menge F von FDs und eine Menge von Attributen α
- **Ausgabe**: die vollständige Menge von Attributen α^+, für die gilt $\alpha \to \alpha^+$
- *AttrHülle*(F, α)

```
Erg := α;
while (Änderungen an Erg) do
    foreach FD β → γ in F do
        if β ⊆ Erg then Erg := Erg ∪ γ;
Ausgabe α⁺ = Erg;
```

Mit Hilfe dieses Algorithmus *AttrHülle* kann man nun sehr einfach bestimmen, ob eine Menge von Attributen κ einen Superschlüssel einer Relation $\mathcal{R}$ bezüglich der FDs F bildet. Dazu wendet man *AttrHülle* (F, κ) an um κ^+ zu ermitteln. Nur falls $\kappa^+ = \mathcal{R}$ ergibt, ist κ ein Superschlüssel von $\mathcal{R}$.

6.3.1 Kanonische Überdeckung

Im Allgemeinen gibt es viele unterschiedliche äquivalente Mengen von funktionalen Abhängigkeiten. Zwei Mengen F und G von funktionalen Abhängigkeiten heißen genau dann *äquivalent* (in Zeichen $F \equiv G$), wenn ihre Hüllen gleich sind, d.h. $F^+ = G^+$. Diese Definition von Äquivalenz ist intuitiv einleuchtend, da die gleiche Hülle der beiden Mengen F und G impliziert, dass dieselben FDs aus F und G ableitbar sind.

Zu einer gegebenen Menge F von FDs gibt es also eine eindeutige Hülle F^+. Diese Menge F^+ enthält aber i.A. sehr viele Abhängigkeiten, so dass der Umgang mit F^+ sehr unübersichtlich ist. Insbesondere nachteilig wirkt sich eine große, redundante Menge von funktionalen Abhängigkeiten im Rahmen der Konsistenzüberprüfung bei Datenbankmodifikationen aus. Man beachte, dass nach einer Änderungsoperation die Einhaltung der spezifizierten FDs überprüft werden muss. Deshalb ist man im Entwurfsprozess und bei der Überprüfung von FDs an einer kleinstmöglichen noch äquivalenten Menge von FDs interessiert. Zu einer gegebenen Menge F von FDs nennt man F_c eine *kanonische Überdeckung*, falls folgende drei Eigenschaften erfüllt sind:

1. $F_c \equiv F$, d.h. $F_c^+ = F^+$
2. In F_c existieren keine FDs $\alpha \to \beta$, bei denen α oder β überflüssige Attribute enthalten. D.h. es muss folgendes gelten:
   (a) $\forall A \in \alpha : (F_c - (\alpha \to \beta) \cup ((\alpha - A) \to \beta)) \not\equiv F_c$
   (b) $\forall B \in \beta : (F_c - (\alpha \to \beta) \cup (\alpha \to (\beta - B))) \not\equiv F_c$
3. Jede linke Seite einer funktionalen Abhängigkeit in F_c ist einzigartig. Dies kann durch sukzessive Anwendung der Vereinigungsregel auf FDs der Art $\alpha \to \beta$ und $\alpha \to \gamma$ erzielt werden, so dass die beiden FDs durch $\alpha \to \beta\gamma$ ersetzt werden.

Zu einer gegebenen Menge F von FDs kann man eine kanonische Überdeckung wie folgt bestimmen:

1. Führe für jede FD $\alpha \to \beta \in F$ die Linksreduktion durch, also:
   - Überprüfe für alle $A \in \alpha$, ob A überflüssig ist, d.h. ob
     $$\beta \subseteq \mathit{AttrHülle}(F, \alpha - A)$$
     gilt. Falls dies der Fall ist, ersetze $\alpha \to \beta$ durch $(\alpha - A) \to \beta$.
2. Führe für jede (verbliebene) FD $\alpha \to \beta$ die Rechtsreduktion durch, also:
   - Überprüfe für alle $B \in \beta$, ob
     $$B \in \mathit{AttrHülle}(F - (\alpha \to \beta) \cup (\alpha \to (\beta - B)), \alpha)$$
     gilt. In diesem Fall ist B auf der rechten Seite überflüssig und kann eliminiert werden, d.h. $\alpha \to \beta$ wird durch $\alpha \to (\beta - B)$ ersetzt.

3. Entferne die FDs der Form $\alpha \rightarrow \emptyset$, die im 2. Schritt möglicherweise entstanden sind.

4. Fasse mittels der Vereinigungsregel FDs der Form $\alpha \rightarrow \beta_1, \ldots, \alpha \rightarrow \beta_n$ zusammen, so dass $\alpha \rightarrow (\beta_1 \cup \cdots \cup \beta_n)$ verbleibt.

Betrachten wir ein ganz kleines Beispiel für die Herleitung der kanonischen Überdeckung. Die Menge F habe folgende Form:

$$F = \{A \rightarrow B, B \rightarrow C, AB \rightarrow C\}$$

In Schritt 1. wird $AB \rightarrow C$ durch $A \rightarrow C$ ersetzt, da B auf der linken Seite überflüssig ist (C ist nämlich schon über die ersten beiden FDs funktional abhängig von A). Im zweiten Schritt, der Rechtsreduktion, wird $A \rightarrow C$ durch $A \rightarrow \emptyset$ ersetzt, da C auf der rechten Seite überflüssig ist. Dies folgt daraus, dass

$$C \in \mathit{AttrHülle}(\{A \rightarrow B, B \rightarrow C, A \rightarrow \emptyset\}, \{A\})$$

gilt. In Schritt 3. wird dann lediglich $A \rightarrow \emptyset$ eliminiert, so dass $F_c = \{A \rightarrow B, B \rightarrow C\}$ übrigbleibt. Hier gibt es natürlich in Schritt 4. nichts mehr zusammenzufassen.

6.4 „Schlechte" Relationenschemata

Schlecht entworfene Relationenschemata können zu sogenannten *Anomalien* führen, die wir im folgenden anhand eines anschaulichen Beispiels illustrieren wollen. Dazu betrachten wir die Relation *ProfVorl*, in der Professoren zusammen mit den von ihnen gelesenen Vorlesungen modelliert sind:

ProfVorl						
PersNr	Name	Rang	Raum	VorlNr	Titel	SWS
2125	Sokrates	C4	226	5041	Ethik	4
2125	Sokrates	C4	226	5049	Mäeutik	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

Es handelt sich hierbei um einen schlechten Entwurf, wie wir auch schon in Kapitel 3.3 festgestellt hatten. Dieser Entwurf führt demzufolge auch zu den (für diese Diskussion gewünschten) Anomalien. Wir unterscheiden drei Arten von Anomalien, die in den nachfolgenden Unterabschnitten behandelt werden.

6.4.1 Die Updateanomalien

Wenn ein Professor, sagen wir „Sokrates", von einem Raum in einen anderen umzieht, muss dies in der Datenbank entsprechend fortgeschrieben werden. Aufgrund des schlechten Schemas existiert diese Information aber mehrfach, also redundant. Deshalb kann leicht der Fall eintreten, dass einige Einträge übersehen werden. Selbst wenn man – durch ein entsprechendes Programm – sicherstellen kann, dass immer alle redundanten Einträge gleichzeitig abgeändert werden, hat der vorliegende Entwurf von *ProfVorl* dennoch zwei schwerwiegende Nachteile:

1. Erhöhter Speicherbedarf wegen der redundant zu speichernden Informationen und
2. Leistungseinbußen bei Änderungen, da mehrere Einträge abgeändert werden müssen.

6.4.2 Einfügeanomalien

Bei diesem schlechten Entwurf wurden Informationen zweier Entitytypen (aus der realen Anwendungswelt) vermischt. Deshalb treten Probleme auf, wenn man Information eintragen will, die zu nur einem Entitytypen gehört.

Will man z.B. die Daten für neu berufene Professoren eintragen, die noch keine Vorlesung halten, so geht dies nur, indem man die Attribute *VorlNr*, *Titel* und *SWS* mit NULL-Werten besetzt.

Ein analoges Problem tritt auf, wenn man eine Vorlesung eintragen will, für die aber noch kein Referent bestimmt wurde.

6.4.3 Löschanomalien

Wenn die Information bezüglich eines der zwei miteinander vermischten Entitytypen gelöscht wird, kann es zum gleichzeitigen und unbeabsichtigten Verlust der Daten des anderen Entitytyps kommen. Betrachten wir als Beispiel das Löschen der Vorlesung „Der Wiener Kreis". Da dies die einzige von „Popper" gehaltene Vorlesung ist, geht durch die Löschung des Vorlesungstupels gleichzeitig auch die Information zum Professor „Popper" verloren. Dies wäre nur zu vermeiden, wenn man die entsprechenden Attribute aus dem Vorlesungskontext mit NULL-Werten besetzt. Andererseits ist so etwas bei Professoren, die mehrere Vorlesungen halten, nicht notwendig. Beispielsweise kann man die Vorlesung „Mäeutik" löschen, ohne dass die Infomation zu „Sokrates" verloren ginge.

6.5 Zerlegung (Dekomposition) von Relationen

Die im vorhergehenden Abschnitt dargestellten Anomalien sind darauf zurückzuführen, dass nicht „zusammenpassende" Informationen in einer Relation gebündelt wurden. Um einen solchen unzulänglichen Entwurf zu revidieren, werden bei der sogenannten *Normalisierung*, die in den nachfolgenden Abschnitten behandelt wird, Relationenschemata aufgespalten. Mit anderen Worten, ein Relationenschema $\mathcal{R}$ wird in die Relationenschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ zerlegt. Dabei enthalten die Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ natürlich jeweils nur eine Teilmenge der Attribute aus $\mathcal{R}$, also $\mathcal{R}_i \subseteq \mathcal{R}$ für $1 \leq i \leq n$.

Es gibt zwei sehr grundlegende Korrektheitskriterien für eine solche Zerlegung von Relationenschemata:

1. *Verlustlosigkeit*: Die in der ursprünglichen Relationenausprägung R des Schemas $\mathcal{R}$ enthaltenen Informationen müssen aus den Ausprägungen $R_1, \ldots, R_n$ der neuen Relationenschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ rekonstruierbar sein.

2. *Abhängigkeitserhaltung*: Die für $\mathcal{R}$ geltenden funktionalen Abhängigkeiten müssen auf die Schemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ übertragbar sein.

Wir werden diese beiden Kriterien in den nachfolgenden Unterabschnitten etwas detaillierter behandeln.

6.5.1 Verlustlosigkeit

Für die Diskussion reicht es aus, sich auf die Zerlegung von $\mathcal{R}$ in zwei Relationenschemata $\mathcal{R}_1$ und $\mathcal{R}_2$ zu beschränken.[2] Ein Relationenschema $\mathcal{R}$ werde also auf zwei Schemata $\mathcal{R}_1$ und $\mathcal{R}_2$ aufgeteilt. Es handelt sich hierbei um eine gültige Zerlegung, wenn alle Attribute aus $\mathcal{R}$ erhalten bleiben, d.h. es muss gelten:

- $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$

Für eine Ausprägung R von $\mathcal{R}$ definieren wir jetzt die Ausprägungen R_1 von $\mathcal{R}_1$ und R_2 von $\mathcal{R}_2$ wie folgt:

$$\begin{aligned} R_1 &:= \Pi_{\mathcal{R}_1}(R) \\ R_2 &:= \Pi_{\mathcal{R}_2}(R) \end{aligned}$$

Die Zerlegung von $\mathcal{R}$ in $\mathcal{R}_1$ und $\mathcal{R}_2$ ist *verlustlos*, falls für jede mögliche (gültige) Ausprägung R von $\mathcal{R}$ gilt:

$$R = R_1 \bowtie R_2$$

Die in R enthaltene Information muss also durch den natürlichen Verbund (Join) der beiden Relationen R_1 und R_2 rekonstruierbar sein.

Wir wollen zunächst ein Beispiel betrachten, in dem die Zerlegung zu einem Verlust von Information führt. Die Relation *Biertrinker* habe folgende Gestalt:

Biertrinker		
Kneipe	Gast	Bier
Kowalski	Kemper	Pils
Kowalski	Eickler	Hefeweizen
Innsteg	Kemper	Hefeweizen

In dieser Relation seien Kneipen, Gäste und Biersorten eindeutig durch die Namen identifiziert. Die Relation enthält die Information, welche Biersorte die Gäste in der jeweiligen Kneipe trinken. Die getrunkene Biersorte kann je nach Kneipe (bzw. beliefernder Brauerei) für denselben Gast variieren – z.B. trinkt Kemper im Kowalski immer Pils aber im Innsteg immer Hefeweizen.

Eine Zerlegung von *Biertrinker* könnte wie folgt durchgeführt werden:

- Besucht: {[Kneipe, Gast]}
- Trinkt: {[Gast, Bier]}

[2] Zur Begründung siehe Aufgabe 6.13.

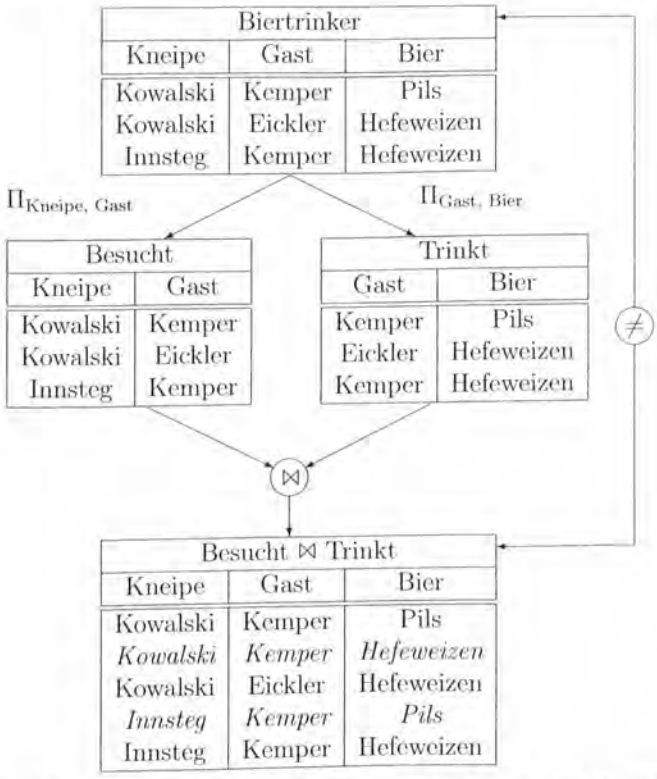


Abbildung 6.1: Illustration der nicht-verlustlosen Zerlegung von *Biertrinker*

Diese Zerlegung ist **nicht** verlustlos! Das erkennt man schon an der obigen Ausprägung, wenn man die Relationenausprägungen für *Besucht* und *Trinkt* bildet:

$$\begin{aligned} \text{Besucht} &:= \Pi_{\text{Kneipe,Gast}}(\text{Biertrinker}) \\ \text{Trinkt} &:= \Pi_{\text{Gast,Bier}}(\text{Biertrinker}) \end{aligned}$$

Die Projektionen resultieren in den beiden in der Mitte von Abbildung 6.1 gezeigten Ausprägungen der Relationen *Besucht* und *Trinkt*. Leider ergibt der natürliche Verbund der beiden Relationen *Besucht* und *Trinkt* nicht die Ausgangsrelation *Biertrinker*, d.h.:

$$\text{Biertrinker} \neq (\text{Besucht} \bowtie \text{Trinkt})$$

Die Relation (*Besucht* $\bowtie$ *Trinkt*) enthält nämlich die Tupel [*Kowalski, Kemper, Hefeweizen*] und [*Innsteg, Kemper, Pils*], die in der Ursprungsrelation *Biertrinker* nicht enthalten waren. Durch die Zerlegung ist die Assoziation von Biersorten und Gästen *relativ* zu der besuchten *Kneipe* verloren gegangen. Auch dies ist in Abbildung 6.1 gezeigt – die kursiv geschriebenen Einträge in der Relation *Besucht* $\bowtie$ *Trinkt* waren in der Ursprungsrelation nicht vorhanden und stellen einen Informationsverlust dar. Es mag seltsam erscheinen, dass zusätzliche Tupel einen Informationsverlust darstellen – das ist aber tatsächlich so, weil die Zuordnungen dadurch verloren gingen.

6.5.2 Kriterien für die Verlustlosigkeit einer Zerlegung

Es ist – wie das vorangegangene Beispiel zeigt – für die Datenbankentwerfer nicht immer auf den ersten Blick ersichtlich, ob eine beabsichtigte Zerlegung verlustlos ist oder nicht. Deshalb ist eine formale Charakterisierung verlustloser Zerlegungen auf der Basis von funktionalen Abhängigkeiten sinnvoll und notwendig.

Eine Zerlegung von $\mathcal{R}$ mit zugehörigen funktionalen Abhängigkeiten $F_{\mathcal{R}}$ in $\mathcal{R}_1$ und $\mathcal{R}_2$ ist verlustlos, wenn mindestens eine der folgenden funktionalen Abhängigkeiten herleitbar ist:

- $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_1 \in F_{\mathcal{R}}^+$
- $(\mathcal{R}_1 \cap \mathcal{R}_2) \rightarrow \mathcal{R}_2 \in F_{\mathcal{R}}^+$

Mit anderen Worten: Es gelte $\mathcal{R} = \alpha \cup \beta \cup \gamma$, $\mathcal{R}_1 = \alpha \cup \beta$ und $\mathcal{R}_2 = \alpha \cup \gamma$ mit paarweise disjunkten Attributmengen α, β und γ. Dann muss mindestens eine von zwei Bedingungen gelten:

- $\beta \subseteq \mathit{AttrHülle}(F_{\mathcal{R}}, \alpha)$ oder
- $\gamma \subseteq \mathit{AttrHülle}(F_{\mathcal{R}}, \alpha)$

Wir werden nachher bei der Diskussion der sogenannten mehrwertigen Abhängigkeiten (siehe Abschnitt 6.10) sehen, dass dies eine hinreichende, aber keine notwendige Bedingung für Verlustlosigkeit ist. Das bedeutet, wenn diese Bedingung erfüllt ist, kann man sicher sein, dass kein Informationsverlust auftreten kann. Aber es gibt auch verlustlose Zerlegungen, bei denen diese Bedingung nicht erfüllt ist, für die also die Bedingung zu „stark" ist. Unser *Biertrinker*-Beispiel war eine „verlustige" Zerlegung und dementsprechend war die Bedingung verletzt. Es gilt nämlich nur die eine nicht-triviale funktionale Abhängigkeit

- {Kneipe, Gast} → {Bier},

wohingegen keine der zwei möglichen, die Verlustlosigkeit garantierenden FDs

- {Gast} → {Bier}
- {Gast} → {Kneipe}

erfüllt sind.

Wir wollen natürlich noch ein anschauliches Beispiel für die verlustlose Zerlegung liefern. Man betrachte die Relation *Eltern* : {[*Vater*, *Mutter*, *Kind*]} und deren Zerlegung in *Väter* : {[*Vater*, *Kind*]} und *Mütter* : {[*Mutter*, *Kind*]}, die in Abbildung 6.2 gezeigt ist. Für dieses (anekdotische) Beispiel gehen wir davon aus, dass Personen eindeutig durch ihre Vornamen identifizierbar sind.

Diese Zerlegung ist verlustlos, da sogar beide funktionalen Abhängigkeiten

- {Kind} → {Mutter}
- {Kind} → {Vater}

gelten.

Allerdings ist diese Zerlegung auch nicht besonders sinnvoll, da dadurch keine Anomalien abgebaut werden – die Relation *Eltern* entspricht schon einem „sinnvollen" Design.

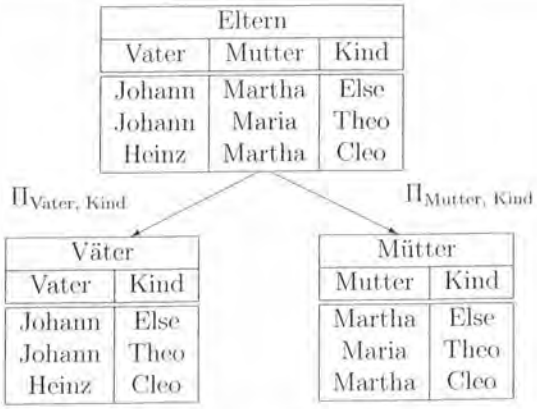


Abbildung 6.2: Verlustlose Zerlegung der Relation *Eltern*

6.5.3 Abhängigkeitsbewahrung

Eine Zerlegung von $\mathcal{R}$ mit zugehörigen funktionalen Abhängigkeiten $F_{\mathcal{R}}$ in die Relationenschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ sollte so erfolgen, dass die Überprüfung aller funktionalen Abhängigkeiten lokal auf den $\mathcal{R}_i$ erfolgen kann, ohne dass Joins notwendig sind. Wie bereits geschildert, stellen die FDs in $F_{\mathcal{R}}$ Konsistenzbedingungen dar, die von jeder aktuellen Ausprägung R von $\mathcal{R}$ erfüllt sein müssen. Dies bedeutet, dass man die Einhaltung der FDs bei Änderungen auf der Datenbank erneut überprüfen muss. Wenn jetzt aber $\mathcal{R}$ in $\mathcal{R}_1, \ldots, \mathcal{R}_n$ zerlegt wird, gibt es keine Ausprägung R mehr, sondern nur noch $R_1, \ldots, R_n$. Theoretisch könnte man R als $R_1 \bowtie \cdots \bowtie R_n$ jeweils neu berechnen und die Abhängigkeiten dann auf der Basis von R überprüfen. Das wäre aber viel zu aufwändig. Deshalb wird durch die *Abhängigkeitsbewahrung* gefordert, dass alle Abhängigkeiten in $F_{\mathcal{R}}$ lokal auf den R_i $(1 \leq i \leq n)$ überprüft werden können. Dazu bestimmt man für jedes $\mathcal{R}_i$ die Einschränkung $F_{\mathcal{R}_i}$ der Abhängigkeiten aus $F_{\mathcal{R}}^+$, d.h. $F_{\mathcal{R}_i}$ enthält die Abhängigkeiten aus der Hülle von $F_{\mathcal{R}}$, deren Attribute alle in $\mathcal{R}_i$ enthalten sind. Bei der Abhängigkeitsbewahrung wird dann folgendes gefordert:

- $F_{\mathcal{R}} \equiv (F_{\mathcal{R}_1} \cup \ldots \cup F_{\mathcal{R}_n})$ bzw. $F_{\mathcal{R}}^+ = (F_{\mathcal{R}_1} \cup \ldots \cup F_{\mathcal{R}_n})^+$

Entsprechend dieser Bedingung nennt man eine abhängigkeitsbewahrende Zerlegung oft auch eine *hüllentreue* Dekomposition.

Wir geben ein Beispiel für eine verlustlose, aber nicht abhängigkeitserhaltende Zerlegung. Die Relation *PLZverzeichnis* stelle ein Postleitzahlenverzeichnis dar (nach fünfstelligem System):

$$\text{PLZverzeichnis} : \{[\text{Straße}, \text{Ort}, \text{BLand}, \text{PLZ}]\}$$

Wir nehmen vereinfachend an:

- Orte werden durch ihren Namen (*Ort*) und das Bundesland (*BLand*) eindeutig identifiziert.

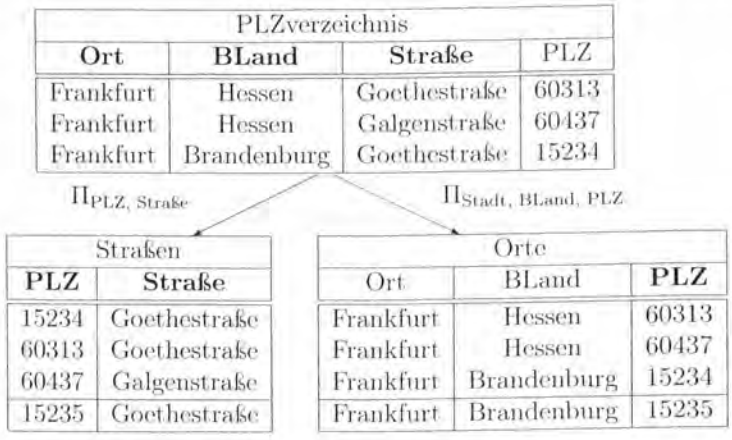


Abbildung 6.3: Zerlegung der Relation PLZverzeichnis

- Innerhalb einer Straße ändert sich die Postleitzahl nicht.
- Postleitzahlengebiete gehen nicht über Ortsgrenzen und Orte nicht über Bundeslandgrenzen hinweg.

Dann gelten folgende funktionale Abhängigkeiten:

- $\{\text{PLZ}\} \rightarrow \{\text{Ort, BLand}\}$
- $\{\text{Straße, Ort, BLand}\} \rightarrow \{\text{PLZ}\}$

Demnach ist die Zerlegung von *PLZverzeichnis* in

$$\begin{aligned} \text{Straßen} &: \{[\text{PLZ, Straße}]\} \\ \text{Orte} &: \{[\text{PLZ, Ort, BLand}]\} \end{aligned}$$

verlustlos, da *PLZ* das einzige gemeinsame Attribut ist und $\{PLZ\} \rightarrow \{Ort, BLand\}$ gilt.

Die funktionale Abhängigkeit $\{Straße, Ort, BLand\} \rightarrow \{PLZ\}$ ist aber jetzt keiner der beiden Relationen *Straßen* oder *Orte* zuzuordnen, so dass die Zerlegung von *PLZverzeichnis* in *Straßen* und *Orte* zwar verlustlos, aber **nicht** abhängigkeitserhaltend ist.

Wir wollen die nachteiligen Auswirkungen dieser nicht abhängigkeitserhaltenden Zerlegung verdeutlichen. In Abbildung 6.3 ist die Zerlegung für eine Beispielausprägung gezeigt. Die Schlüssel der jeweiligen Relationen sind durch Fettdruck markiert. In der Urspungsrelation *PLZverzeichnis* wurde sichergestellt, dass es zu einem (*Ort*, *BLand*, *Straße*)-Tripel nur einen Eintrag, d.h. eine eindeutige Postleitzahl geben kann. Das folgt aus der funktionalen Abhängigkeit

$$\{\text{Ort, BLand, Straße}\} \rightarrow \{\text{PLZ}\}$$

Diese Abhängigkeit, die für *PLZverzeichnis* den Schlüssel festlegt, ging bei der Zerlegung verloren, so dass für die Relation *Straßen* nur noch triviale Abhängigkeiten übrigbleiben.

Demgemäß besteht der Schlüssel von *Straßen* aus der Menge aller Attribute. Es ist jetzt – nach der Zerlegung – ohne weiteres möglich, die unten in der Abbildung 6.3 gezeigten Tupel [15235, *Goethestraße*] in *Straßen* und [*Frankfurt*, *Brandenburg*, 15235] in *Orte* einzufügen. Dadurch bekommt die *Goethestraße* im *Brandenburger Frankfurt* eine zusätzliche Postleitzahl. Die Relationen *Straßen* und *Orte* sind *lokal* konsistent; die Kombination dieser beiden Einfügungen verletzt aber eine *globale* funktionale Abhängigkeit der Relation *PLZverzeichnis*. Die Verletzung der Konsistenzbedingung $\{Straße, Ort, BLand\} \rightarrow \{PLZ\}$ ist nur nach einem Join der Relationen *Straßen* und *Orte* aufzudecken.

Dieses Beispiel sollte verdeutlichen, dass die Abhängigkeitserhaltung bei allen Zerlegungen anzustreben ist.

6.6 Erste Normalform

Die erste Normalform ist bei der von uns benutzten Definition des relationalen Modells automatisch eingehalten. Die erste Normalform verlangt, dass alle Attribute atomare Wertebereiche (Domänen) haben. Demnach wären zusammengesetzte, mengenwertige oder gar relationenwertige Attributdomänen nicht zulässig.

Das folgende ist ein Beispiel für eine Relation mit einem mengenwertigen Attribut:

Eltern		
Vater	Mutter	Kinder
Johann	Martha	{Else, Lucia}
Johann	Maria	{Theo, Josef}
Heinz	Martha	{Cleo}

In der Relation *Eltern* seien Personen eindeutig durch ihren Vornamen identifiziert. Das Attribut *Kinder* ist mengenwertig, wobei die Menge die Namen der Kinder enthält, die dieselben Eltern haben. Diese Relation ist nicht in erster Normalform. Ein gültiges Schema in erster Normalform wird durch „Flachklopfen" erreicht:

Eltern		
Vater	Mutter	Kind
Johann	Martha	Else
Johann	Martha	Lucia
Johann	Maria	Theo
Johann	Maria	Josef
Heinz	Martha	Cleo

Es wird also verlangt, dass Attributwerte nicht weiter zerlegbar sind.

Es gibt neuere Entwicklungen im Datenbankbereich, in denen gerade auf die Einhaltung der ersten Normalform verzichtet wurde. Dieses Modell wird dementsprechend oft als NF^2-Modell (non-first normal form Modell) oder geschachteltes relationales Modell (engl. nested relational model) bezeichnet.

Bei diesen erweiterten relationalen Modellen sind nicht nur mengenwertige Attribute, wie oben gezeigt, sondern sogar relationenwertige Attribute – also geschachtelte Relationen – möglich. Als Beispiel schauen wir uns die folgende geschachtelte

Relation an, in der wir zusätzlich zum Namen auch noch das Alter der Kinder speichern:

Eltern			
Vater	Mutter	Kinder	
		KName	KAlter
Johann	Martha	Else	5
		Lucia	3
Johann	Maria	Theo	3
		Josef	1
Heinz	Martha	Cleo	9

Hier ist also z.B. in dem ersten Tupel der Relation *Eltern* die Relation mit den zwei Tupeln $[Else, 5]$ und $[Lucia, 3]$ geschachtelt.

In der weiteren Diskussion dieses Kapitels setzen wir stillschweigend immer die erste Normalform voraus.

6.7 Zweite Normalform

Intuitiv verletzt ein Relationenschema die zweite Normalform (2NF), wenn in der Relation Informationen über mehr als ein einziges Konzept modelliert werden. Demnach soll jedes Nichtschlüssel-Attribut der Relation, wie Kent (1983) es ausdrückt, einen Fakt zu dem dieses Konzept identifizierenden Schlüssel (und zwar den gesamten Schlüssel und nichts als den Schlüssel) ausdrücken.

Formal ausgedrückt: Eine Relation $\mathcal{R}$ mit zugehörigen FDs F ist in zweiter Normalform, falls jedes Nichtschlüssel-Attribut $A \in \mathcal{R}$ voll funktional abhängig ist von jedem Kandidatenschlüssel der Relation.

Seien also $\kappa_1, \ldots, \kappa_i$ die Kandidatenschlüssel[3] von $\mathcal{R}$ – einschließlich des ausgewählten Primärschlüssels, der ja auch Kandidatenschlüssel sein muss. Sei $A \in \mathcal{R} - (\kappa_1 \cup \cdots \cup \kappa_i)$. Ein solches Attribut A wird auch als *nicht-prim* bezeichnet – im Gegensatz zu den Schlüsselattributen, die man als *prim* bezeichnet. Dann muss also für alle κ_j $(1 \leq j \leq i)$ gelten:

$$\kappa_j \xrightarrow{\bullet} A \in F^+$$

D.h., es muss die FD $\kappa_j \rightarrow A$ gelten und diese FD ist linksreduziert.

Wir wollen uns ein Beispiel einer Relation anschauen, die diese Bedingung verletzt. Die Relation *StudentenBelegung* sei wie folgt gegeben:

[3] Man beachte, dass Kandidatenschlüssel – im Gegensatz zu Superschlüsseln – minimal sein müssen.

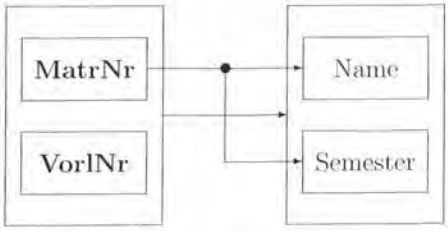


Abbildung 6.4: Schematische Darstellung der funktionalen Abhängigkeiten

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

Die Leser werden bemerkt haben, dass diese Relation gerade dem Join der Relation *hören* und *Studenten* aus unserem Universitätsbeispiel entspricht.

Die Relation *StudentenBelegung* hat den Schlüssel $\{MatrNr, VorlNr\}$. Zusätzlich zu den aus diesem Schlüssel folgenden funktionalen Abhängigkeiten gibt es aber noch die funktionalen Abhängigkeiten

$$\{\text{MatrNr}\} \rightarrow \{\text{Name}\} \quad \text{und} \quad \{\text{MatrNr}\} \rightarrow \{\text{Semester}\},$$

wodurch die zweite Normalform verletzt wird. Grafisch kann man sich das wie in Abbildung 6.4 verdeutlichen.

Diese oben gezeigte Beispielausprägung illustriert (nochmals) die schwerwiegenden Anomalien:

- Einfügeanomalie: Was macht man mit Studenten, die (noch) keine Vorlesungen hören?
- Updateanomalien: Wenn z.B. „Carnap" ins vierte Semester kommt, muss sichergestellt werden, dass alle vier Tupel geändert werden.
- Löschanomalien: Was passiert, wenn „Fichte" ihre einzige Vorlesung absagt?

Die Lösung dieser Probleme ist relativ offensichtlich: Man zerlegt die Relation in mehrere Teilrelationen, die dann die zweite Normalform erfüllen. In unserem Fall wird *StudentenBelegung* in die beiden folgenden Relationen zerlegt:

- hören: {[MatrNr, VorlNr]}
- Studenten: {[MatrNr, Name, Semester]}

Diese beiden Relationen erfüllen beide die zweite Normalform. Weiterhin stellen sie natürlich eine verlustlose Zerlegung dar.

Wir werden hier nicht näher auf den Zerlegungsalgorithmus, der eine gegebene Relation $\mathcal{R}$ in mehrere 2NF-Teilrelationen $\mathcal{R}_1, \ldots, \mathcal{R}_n$ aufspaltet, eingehen. In der Praxis sollte nämlich immer die „schärfere" dritte Normalform angestrebt werden.

6.8 Dritte Normalform

Nach den Ausführungen von Kent (1983) wird die dritte Normalform intuitiv verletzt, wenn ein Nichtschlüssel-Attribut einen Fakt einer Attributmenge darstellt, die keinen Schlüssel bildet. Die Verletzung der Normalform könnte also dazu führen, dass derselbe Fakt mehrfach gespeichert wird.

Ein Relationenschema $\mathcal{R}$ ist in *dritter Normalform*, wenn für jede für $\mathcal{R}$ geltende funktionale Abhängigkeit der Form $\alpha \rightarrow B$ mit $\alpha \subseteq \mathcal{R}$ und $B \in \mathcal{R}$ mindestens *eine* von drei Bedingungen gilt:

- $B \in \alpha$, d.h. die FD ist trivial.
- Das Attribut B ist in einem Kandidatenschlüssel von $\mathcal{R}$ enthalten – also B ist *prim*.
- α ist Superschlüssel von $\mathcal{R}$.

Als Beispiel für eine Relation, die nicht in dritter Normalform ist, betrachten wir nochmals die in Abschnitt 6.3 bereits eingeführte Relation *ProfessorenAdr*:

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

Wir hatten schon folgende, teilweise vereinfachende Annahmen getroffen: Unter *Ort* verstehen wir den eindeutigen Erstwohnsitz der Professoren. Die *Landesregierung* ist die eine „tonangebende" Partei, die also den Ministerpräsidenten bzw. die Ministerpräsidentin stellt, so dass *Landesregierung* funktional abhängig von *BLand* ist. Weiterhin machen wir die Annahmen, dass Orte innerhalb der Bundesländer (nach wie vor) eindeutig benannt seien. Die Postleitzahl (*PLZ*) ändert sich nicht innerhalb einer Straße, Städte und Straßen gehen nicht über Bundeslandgrenzen hinweg.

Die aus diesen Annahmen folgenden funktionalen Abhängigkeiten sind in Abbildung 6.5 grafisch dargestellt. Es ist ersichtlich, dass $\{PersNr\}$ und $\{Raum\}$ jeweils Kandidatenschlüssel von *ProfessorenAdr* sind. Offensichtlich ist die Relation *ProfessorenAdr* nicht in dritter Normalform, da z.B. die FD $\{Ort, BLand\} \rightarrow \{Vorwahl\}$ die Kriterien der 3NF verletzt.

Wir geben jetzt einen sogenannten **Synthesealgorithmus** an, mit dem zu einem gegebenen Relationenschma $\mathcal{R}$ mit funktionalen Abhängigkeiten F eine Zerlegung in $\mathcal{R}_1, \ldots, \mathcal{R}_n$ ermittelt wird, die alle drei folgenden Kriterien erfüllt:

- $\mathcal{R}_1, \ldots, \mathcal{R}_n$ ist eine verlustlose Zerlegung von $\mathcal{R}$.
- Die Zerlegung ist abhängigkeitsbewahrend
- Alle $\mathcal{R}_i$ $(1 \leq i \leq n)$ sind in dritter Normalform.

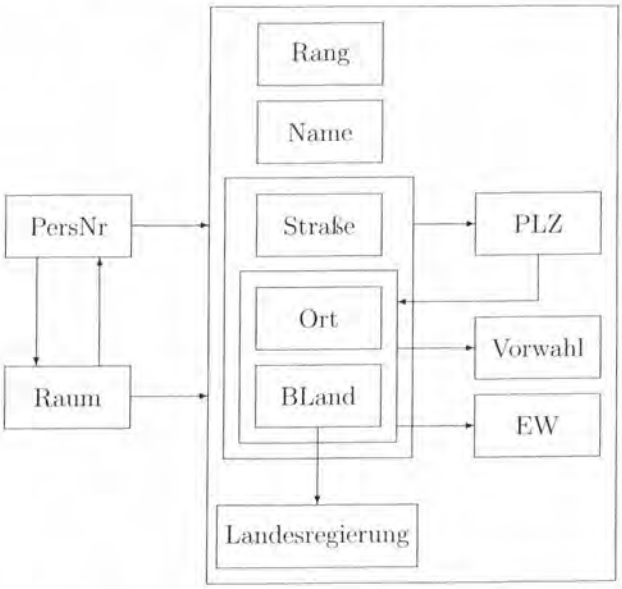


Abbildung 6.5: Grafische Darstellung der funktionalen Abhängigkeiten

Der Synthesealgorithmus berechnet die Zerlegung auf der Basis der funktionalen Abhängigkeiten wie folgt:

1. Bestimme die kanonische Überdeckung F_c zu F. Zur Wiederholung:
   (a) Linksreduktion der FDs
   (b) Rechtsreduktion der FDs
   (c) Entfernung von FDs der Form $\alpha \to \emptyset$
   (d) Zusammenfassung von FDs mit gleichen linken Seiten
2. Für jede funktionale Abhängigkeit $\alpha \to \beta \in F_c$:
   - Kreiere ein Relationenschema $\mathcal{R}_\alpha := \alpha \cup \beta$.
   - Ordne $\mathcal{R}_\alpha$ die FDs $F_\alpha := \{\alpha' \to \beta' \in F_c \mid \alpha' \cup \beta' \subseteq \mathcal{R}_\alpha\}$ zu.
3. Falls eines der in Schritt 2. erzeugten Schemata $\mathcal{R}_\alpha$ einen Kandidatenschlüssel von $\mathcal{R}$ bzgl. F_c enthält, sind wir fertig; sonst wähle einen Kandidatenschlüssel $\kappa \subseteq \mathcal{R}$ aus und definiere folgendes zusätzliche Schema:
   - $\mathcal{R}_\kappa := \kappa$
   - $F_\kappa := \emptyset$
4. Eliminiere diejenigen Schemata $\mathcal{R}_\alpha$, die in einem anderen Relationenschema $\mathcal{R}_{\alpha'}$ enthalten sind, d.h.
   - $\mathcal{R}_\alpha \subseteq \mathcal{R}_{\alpha'}$

Wir wollen den Synthesealgorithmus an unserer Beispielrelation *ProfessorenAdr* demonstrieren. In Schritt 1. wird die kanonische Überdeckung der funktionalen Abhängigkeiten ermittelt, wobei wir uns hier die Herleitung ersparen. Die Leser mögen die Berechnung der kanonischen Überdeckung anhand des in Abschnitt 6.3.1 ausgearbeiteten Algorithmus selbst durchführen.

Die kanonische Überdeckung enthält folgende FDs:

$$
\begin{aligned}
fd_1 &: \{\text{PersNr}\} \rightarrow \{\text{Raum, Name, Rang, Straße, Ort, BLand}\} \\
fd_2 &: \{\text{Raum}\} \rightarrow \{\text{PersNr}\} \\
fd_3 &: \{\text{Straße, Ort, BLand}\} \rightarrow \{\text{PLZ}\} \\
fd_4 &: \{\text{Ort, BLand}\} \rightarrow \{\text{Vorwahl, EW}\} \\
fd_5 &: \{\text{BLand}\} \rightarrow \{\text{Landesregierung}\} \\
fd_6 &: \{\text{PLZ}\} \rightarrow \{\text{Ort, BLand}\}
\end{aligned}
$$

In Schritt 2. des Synthesealgorithmus werden diese sechs FDs $fd_1, \ldots, fd_6$ jetzt sukzessive behandelt. Aus fd_1 leitet sich das Relationenschema

$$\text{Professoren} : \{[\text{PersNr, Name, Rang, Raum, Straße, Ort, BLand}]\}$$

mit den FDs fd_1 und fd_2 ab. Die funktionale Abhängigkeit fd_2 liefert keine neue Relation, da alle Attribute dieser FD schon in *Professoren* enthalten sind. Hier nehmen wir also den Schritt 4. des Synthesealgorithmus schon vorweg. Die funktionale Abhängigkeit fd_3 resultiert in der Relation

$$\text{PLZverzeichnis} : \{[\text{Ort, BLand, Straße, PLZ}]\}$$

mit den zugeordneten FDs fd_3 und fd_6. Die funktionale Abhängigkeit fd_4 liefert

$$\text{Städteverzeichnis} : \{[\text{Ort, BLand, Vorwahl, EW}]\}$$

mit nur einer zugeordneten FD, nämlich fd_4 selber. Die FD fd_5 liefert die Relation

$$\text{Regierungen} : \{[\text{BLand, Landesregierung}]\}$$

mit gerade dieser einen zugeordneten FD fd_5. Die letzte FD, nämlich fd_6, liefert nichts Neues, da alle in der FD vorkommenden Attribute schon in der Relation *PLZverzeichnis* vorkommen.

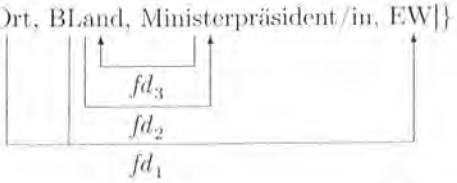
In Schritt 3. des Synthesealgorithmus wird für unser Beispiel nichts Neues erzeugt, da ein Kandidatenschlüssel – nämlich *Raum* oder *PersNr* – schon in einer der Relationen enthalten ist – nämlich in *Professoren*. Im Allgemeinen muss Schritt 3. aber beachtet werden, da sonst nicht immer die Verlustlosigkeit gesichert ist (siehe dazu auch die Übungsaufgabe 6.8)! Den Schritt 4. des Algorithmus hatten wir schon vorweg genommen, so dass hier nichts mehr zu tun bleibt.

6.9 Boyce-Codd Normalform

Die Boyce-Codd Normalform (BCNF) stellt nochmals eine Verschärfung dar. Das Ziel der BCNF besteht darin, dass Informationseinheiten (Fakten) nicht mehrmals, sondern nur genau einmal gespeichert werden. Ein Relationenschema $\mathcal{R}$ mit FDs F ist in BCNF, falls für jede funktionale Abhängigkeit $\alpha \rightarrow \beta$ mindestens eine der folgenden zwei Bedingungen gilt:

- $\beta \subseteq \alpha$, d.h. die Abhängigkeit ist trivial oder
- α ist Superschlüssel von $\mathcal{R}$.

Ein Beispiel einer 3NF-Relation, die nicht die strengeren Bedingungen der Boyce-Codd Normalform erfüllt, ist *Städte*:



Die oben eingezeichneten drei funktionalen Abhängigkeiten fd_1, fd_2 und fd_3 implizieren, dass es zwei Kandidatenschlüssel gibt:

- $\kappa_1 = \{\text{Ort}, \text{BLand}\}$
- $\kappa_2 = \{\text{Ort}, \text{Ministerpräsident/in}\}$

Hieraus folgt, dass *Städte* in dritter Normalform ist, da die rechten Seiten von fd_2 und fd_3 jeweils Primattribute (also in Kandidatenschlüsseln enthalten) sind, und die linke Seite von fd_1 ein Kandidatenschlüssel ist. Aber *Städte* ist nicht in BCNF, da die linken Seiten von fd_2 und fd_3 keine Superschlüssel sind. Die Verletzung der BCNF hat zur Folge, dass die Information, wer welches Bundesland regiert, mehrfach abgespeichert wird.

Man kann grundsätzlich jedes Relationenschema $\mathcal{R}$ mit zugeordneten FDs F so in $\mathcal{R}_1, \ldots, \mathcal{R}_n$ zerlegen, dass gilt:

- Die Zerlegung ist verlustlos und
- die $\mathcal{R}_i$ $(1 \leq i \leq n)$ sind alle in BCNF.

Leider kann man nicht immer eine BCNF-Zerlegung finden, die auch abhängigkeitsbewahrend ist. Diese Fälle sind allerdings in der Praxis selten.

Die Zerlegung eines Relationenschemas $\mathcal{R}$ in BCNF-Teilrelationen wird nach dem **Dekompositionsalgorithmus** durchgeführt, der die Menge $Z = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\}$ von Zerlegungen sukzessive generiert:

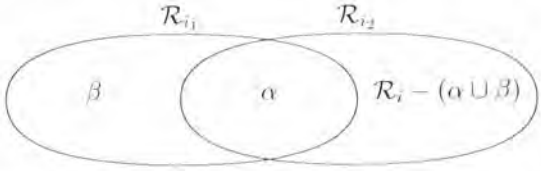
- Starte mit $Z = \{\mathcal{R}\}$
- Solange es noch ein Relationenschema $\mathcal{R}_i \in Z$ gibt, das nicht in BCNF ist, mache folgendes:
  - Finde eine für $\mathcal{R}_i$ geltende nicht-triviale FD $(\alpha \rightarrow \beta)$ mit
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \not\rightarrow \mathcal{R}_i$

    Man sollte die funktionale Abhängigkeit so wählen, dass β alle von α funktional abhängigen Attribute $B \in (\mathcal{R}_i - \alpha)$ enthält, damit der Dekompositionsalgorithmus möglichst schnell terminiert.

- Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i_1} := \alpha \cup \beta$ und $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
- Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i_1}$ und $\mathcal{R}_{i_2}$ ein, also

$$Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}$$

Sobald dieser Algorithmus beendet ist, enthält Z eine Menge von BCNF-Relationen, die eine verlustlose Zerlegung von $\mathcal{R}$ darstellen. Die nachfolgende Grafik illustriert abstrakt die Zerlegung eines Relationenschemas $\mathcal{R}_i$ in $\mathcal{R}_{i_1}$ und $\mathcal{R}_{i_2}$ entlang der FD $\alpha \rightarrow \beta$:

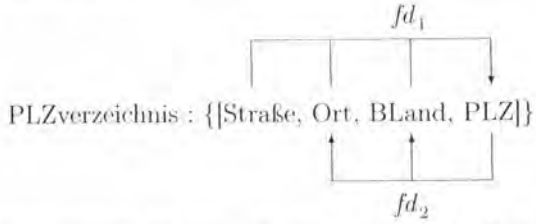


Für unser *Städte*-Beispiel ergibt sich die Zerlegung gemäß der FD $\{BLand\} \rightarrow \{Ministerpräsident/in\}$ wie folgt:

- $\underbrace{\text{Regierungen}}_{\mathcal{R}_{i_1}}$: {[BLand, Ministerpräsident/in]}
- $\underbrace{\text{Städte}'}_{\mathcal{R}_{i_2}}$: {[Ort, BLand, EW]}

Diese beiden Relationen sind jetzt in BCNF, so dass der Algorithmus terminiert. In diesem Beispiel sind auch keine Abhängigkeiten durch die Zerlegung verlorengegangen, da man fd_1 der Relation *Städte'* und fd_2 und fd_3 der Relation *Regierungen* zuordnen kann.

Ein Beispiel für eine BCNF-Zerlegung, bei der Abhängigkeiten verlorengehen, ist folgende Relation *PLZverzeichnis*:



Die funktionale Abhängigkeit fd_2 verletzt die Boyce-Codd Normalform. Die Zerlegung der Relation *PLZverzeichnis* „entlang" dieser Abhängigkeit fd_2 ergibt die folgenden 2 Relationen:

- Straßen : {[Straße, PLZ]}
- Orte : {[Ort, BLand, PLZ]}

Wie in Abschnitt 6.5.3 schon detailliert ausgeführt, geht durch diese Zerlegung die Abhängigkeit fd_1 verloren.

Das ist auf jeden Fall zu vermeiden. Deshalb gibt man sich in den Fällen, wo eine BCNF-Zerlegung zu einem Abhängigkeitsverlust führen würde, mit der weniger „scharfen" dritten Normalform zufrieden. Also bleibt uns das *PLZverzeichnis* erhalten.

6.10 Mehrwertige Abhängigkeiten

Mehrwertige Abhängigkeiten (engl. *multivalued dependencies*, abgekürzt MVD) sind eine Verallgemeinerung funktionaler Abhängigkeiten, d.h. jede FD ist auch eine MVD, aber nicht umgekehrt.

Seien α und β disjunkte Teilmengen von $\mathcal{R}$ und $\gamma = \mathcal{R} - (\alpha \cup \beta)$. Dann ist β mehrwertig abhängig von α – in Zeichen $\alpha \twoheadrightarrow \beta$ –, wenn in jeder gültigen Ausprägung von $\mathcal{R}$ gilt: Für jedes Paar von Tupeln t_1 und t_2 mit $t_1.\alpha = t_2.\alpha$ existieren zwei weitere Tupel t_3 und t_4 mit folgenden Eigenschaften:

$$\begin{array}{rcl} t_1.\alpha = t_2.\alpha = t_3.\alpha = t_4.\alpha & & \\ t_3.\beta = t_1.\beta & & \\ t_3.\gamma = t_2.\gamma & & \\ t_4.\beta = t_2.\beta & & \\ t_4.\gamma = t_1.\gamma & & \end{array}$$

Mit anderen Worten: Bei 2 Tupeln mit gleichem α-Wert kann man die β-Werte vertauschen, und die resultierenden Tupel müssen auch in der Relation sein. Aus diesem Grund nennt man mehrwertige Abhängigkeiten auch *tupel-generierende* Abhängigkeiten, da eine Relationenausprägung bei Verletzung einer MVD durch das Einfügen zusätzlicher Tupel in einen gültigen Zustand überführt werden kann. Bei funktionalen Abhängigkeiten ist dies nicht der Fall. Warum?

Grafisch kann man sich die Definition der mehrwertigen Abhängigkeit $\alpha \twoheadrightarrow \beta$ wie folgt veranschaulichen:

	R		
	$\overbrace{A_1 \ldots A_i}^{\alpha}$	$\overbrace{A_{i+1} \ldots A_j}^{\beta}$	$\overbrace{A_{j+1} \ldots A_n}^{\gamma}$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

Für den Spezialfall, dass α, β und γ jeweils nur aus einem Attribut A, B und C bestehen, kann man sich die MVD $\alpha \twoheadrightarrow \beta$ auch so vorstellen: Sortiere die Relation nach den A-Werten. Wenn $\{b_1, \ldots, b_i\}$ und $\{c_1, \ldots, c_j\}$ die B bzw. C-Werte für einen bestimmten A-Wert a sind, dann muss die Relation die folgenden $(i * j)$ dreistelligen Tupel

$$\{a\} \times \{b_1, \ldots, b_i\} \times \{c_1, \ldots, c_j\}$$

alle enthalten. Ein konkretes Beispiel möge die mehrwertigen Abhängigkeiten verdeutlichen. Betrachten wir die Relation *Fähigkeiten*, in der die Kenntnisse von natürlichen Sprachen und von Programmiersprachen der Assistenten modelliert werden:

Fähigkeiten		
PersNr	Sprache	ProgSprache
3002	griechisch	C
3002	lateinisch	Pascal
3002	griechisch	Pascal
3002	lateinisch	C
3005	deutsch	Ada

In dieser Relation gelten die MVDs $\{PersNr\} \twoheadrightarrow \{Sprache\}$ und $\{PersNr\} \twoheadrightarrow \{ProgrSprache\}$ – wie die Leser verifizieren mögen.

Offensichtlich handelt es sich hierbei um ein wenig befriedigendes Schema. Man verdeutliche sich dies an folgenden Zahlen: Für Assistenten, die fünf Programmiersprachen und vier natürliche Sprachen beherrschen, müssen jeweils 20 Tupel eingefügt werden. Diese Redundanz wird dadurch verursacht, dass zwei voneinander unabhängige Aspekte – nämlich Kenntnisse von Programmiersprachen und von natürlichen Sprachen – in derselben Relation gespeichert werden.

Glücklicherweise kann man die Relation so zerlegen, dass die Redundanz vermieden wird. Die zwei Relationen

Sprachen	
PersNr	Sprache
3002	griechisch
3002	lateinisch
3005	deutsch

ProgrSprachen	
PersNr	ProgrSprache
3002	C
3002	Pascal
3005	Ada

stellen eine verlustlose Zerlegung von *Fähigkeiten* dar, d.h.:

$$\text{Fähigkeiten} = \underbrace{\Pi_{\text{PersNr,Sprache}}(\text{Fähigkeiten})}_{\text{Sprachen}} \bowtie \underbrace{\Pi_{\text{PersNr,ProgrSprache}}(\text{Fähigkeiten})}_{\text{ProgrSprachen}}$$

Zum Glück ist es kein Zufall, dass diese Zerlegung verlustlos ist. Es gilt nämlich allgemein: Ein Relationenschema $\mathcal{R}$ mit einer Menge D von zugeordneten funktionalen und mehrwertigen Abhängigkeiten kann genau dann verlustlos in die beiden Schemata $\mathcal{R}_1$ und $\mathcal{R}_2$ zerlegt werden, wenn gilt:

- $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ und
- mindestens eine der folgenden zwei MVDs gilt:
  1. $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$ oder
  2. $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$.

In unserer Beispielzerlegung galten sogar beide MVDs. Allgemein gilt nämlich: Wenn in einem Relationenschema $\mathcal{R}$ die mehrwertige Abhängigkeit $\alpha \twoheadrightarrow \beta$ gilt, dann gilt immer auch

$$\alpha \twoheadrightarrow \gamma, \quad \text{für} \quad \gamma = \mathcal{R} - \alpha - \beta.$$

Wir wollen nun noch einen Satz von Ableitungsregeln angeben, mit denen man zu einer gegebenen Menge D von funktionalen und mehrwertigen Abhängigkeiten die Hülle D^+ bestimmen kann. Dabei seien α, β, γ und δ Teilmengen der Attribute des Relationenschemas $\mathcal{R}$. Dann gelten folgende Inferenzregeln:

- *Reflexivität*: Falls $\beta \subseteq \alpha$ erfüllt ist, dann gilt $\alpha \to \beta$.
- *Verstärkung*: Sei $\alpha \to \beta$. Dann gilt $\gamma\alpha \to \gamma\beta$.
- *Transitivität*: Sei $\alpha \to \beta$ und $\beta \to \gamma$. Dann gilt $\alpha \to \gamma$.
- *Komplement*: $\alpha \twoheadrightarrow \beta$. Dann gilt $\alpha \twoheadrightarrow \mathcal{R} - \beta - \alpha$.

- *Mehrwertige Verstärkung*: Sei $\alpha \twoheadrightarrow \beta$ und $\delta \subseteq \gamma$. Dann gilt $\gamma\alpha \twoheadrightarrow \delta\beta$.
- *Mehrwertige Transitivität*: Sei $\alpha \twoheadrightarrow \beta$ und $\beta \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \gamma - \beta$.
- *Verallgemeinerung*: Sei $\alpha \rightarrow \beta$. Dann gilt $\alpha \twoheadrightarrow \beta$.
- *Koaleszenz*: Sei $\alpha \twoheadrightarrow \beta$ und $\gamma \subseteq \beta$. Existiert ein $\delta \subseteq \mathcal{R}$, so dass $\delta \cap \beta = \emptyset$ und $\delta \rightarrow \gamma$, gilt $\alpha \rightarrow \gamma$.

In dem Buch von Maier (1983), das sich der Theorie relationaler Datenbanken widmet, kann man den Beweis finden, dass diese Regeln korrekt und vollständig sind. Die ersten drei Regeln sind gerade die Armstrong-Axiome, die benötigt werden, um die Hülle der funktionalen Abhängigkeiten, die ja in D^+ enthalten ist, zu bestimmen.

Es sind drei weitere Ableitungsregeln sinnvoll:

- *Mehrwertige Vereinigung*: sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \gamma\beta$.
- *Schnittmenge*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \beta \cap \gamma$.
- *Differenz*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \beta - \gamma$ und $\alpha \twoheadrightarrow \gamma - \beta$.

Diese drei Regeln lassen sich aus den anderen, oben angegebenen Regeln ableiten – siehe Übungsaufgabe 6.14. Sie sind also somit korrekt, aber für die Vollständigkeit nicht notwendig.

6.11 Vierte Normalform

Die vierte Normalform (4NF) ist eine Verschärfung der Boyce-Codd Normalform und somit auch der zweiten und dritten Normalform. Bei Relationen in 4NF wird die durch mehrwertige Abhängigkeiten verursachte Redundanz ausgeschlossen. Relationen in 4NF enthalten keine zwei voneinander unabhängigen mehrwertigen Fakten – wie dies in der Beispielrelation *Fähigkeiten* mit 1) *Sprache* und 2) *ProgrSprache* der Fall war.

Um die 4NF definieren zu können, müssen wir vorher noch klären, was eine *triviale* MVD ist. Eine MVD $\alpha \twoheadrightarrow \beta$ bezogen auf $\mathcal{R} \supseteq \alpha \cup \beta$ ist trivial, wenn *jede* mögliche Ausprägung R von $\mathcal{R}$ diese MVD erfüllt. Man kann zeigen – siehe Übungsaufgabe 6.11 – dass $\alpha \twoheadrightarrow \beta$ genau dann trivial ist, wenn gilt:

1. $\beta \subseteq \alpha$ oder
2. $\beta = \mathcal{R} - \alpha$.

Die Leser mögen sich erinnern, dass funktionale Abhängigkeiten nur unter der ersten Bedingung trivial sind.

Eine Relation $\mathcal{R}$ mit zugeordneter Menge D von funktionalen und mehrwertigen Abhängigkeiten ist in *vierter Normalform* 4NF, wenn für jede MVD $\alpha \twoheadrightarrow \beta \in D^+$ eine der folgenden Bedingungen gilt:

1. Die MVD ist trivial oder
2. α ist ein Superschlüssel von $\mathcal{R}$.

Es ist offensichtlich, dass eine 4NF-Relation automatisch auch die Boyce-Codd Normalform erfüllt – das folgt daraus, dass jede funktionale Abhängigkeit $\alpha \to \beta$ auch eine mehrwertige Abhängigkeit $\alpha \twoheadrightarrow \beta$ ist.

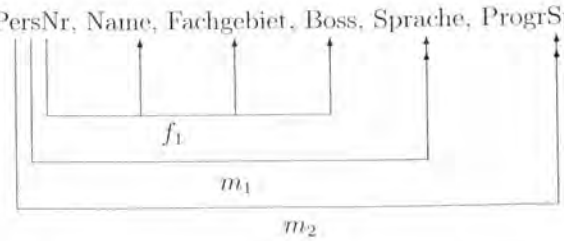
Der **Dekompositionsalgorithmus** für die Zerlegung eines gegebenen Schemas $\mathcal{R}$ mit MVDs D in eine Menge von Relationenschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$, die verlustlos bzgl. $\mathcal{R}$ und alle in 4NF sind, erfolgt analog zur BCNF-Zerlegung:

- Starte mit der Menge $Z := \{\mathcal{R}\}$,
- Solange es eine Relation $\mathcal{R}_i \in Z$ gibt, die nicht in 4NF ist, mache folgendes:
  - Finde eine für $\mathcal{R}_i$ geltende nicht-triviale MVD $\alpha \twoheadrightarrow \beta$, für die gilt
    - $\alpha \cap \beta = \emptyset$
    - $\alpha \not\to \mathcal{R}_i$
  - Zerlege $\mathcal{R}_i$ in $\mathcal{R}_{i_1} := \alpha \cup \beta$ und $\mathcal{R}_{i_2} := \mathcal{R}_i - \beta$
  - Entferne $\mathcal{R}_i$ aus Z und füge $\mathcal{R}_{i_1}$ und $\mathcal{R}_{i_2}$ ein, also

$$Z := (Z - \{\mathcal{R}_i\}) \cup \{\mathcal{R}_{i_1}\} \cup \{\mathcal{R}_{i_2}\}.$$

Sobald dieser Algorithmus terminiert, enthält Z eine Menge von 4NF-Relationenschemata, die $\mathcal{R}$ verlustfrei zerlegen.

Wir wollen die Vorgehensweise an einer Erweiterung der Beispielrelation *Assistenten'* demonstrieren:



In dieser Relation gelten die Abhängigkeiten f_1, m_1, und m_2 – die erste ist funktional, die anderen beiden sind mehrwertig.

Sicherlich ist *Assistenten'* nicht in 4NF – die Relation ist nicht einmal in 2NF (warum?). Die erste „nicht-4NF-konforme" Abhängigkeit ist f_1 – man beachte, dass f_1 auch als MVD angesehen werden kann. Deshalb wird im ersten Schritt die Zerlegung in

- Assistenten: {[PersNr, Name, Fachgebiet, Boss]}
- Fähigkeiten: {[PersNr, Sprache, ProgrSprache]}

durchgeführt. Von diesen beiden Relationen erfüllt jetzt *Assistenten* die 4NF-Eigenschaft; aber *Fähigkeiten* wegen der MVDs m_1 und m_2 nicht. Also wird *Fähigkeiten* weiter zerlegt in:

- Sprachen: {[PersNr, Sprache]}

- ProgrSprachen: {[PersNr, ProgrSprache]}

Diese beiden Relationen sind in 4NF, so dass der Algorithmus jetzt terminiert.

Also haben wir mit *Assistenten*, *Sprachen* und *ProgrSprachen* eine verlustlose Zerlegung von *Assistenten'* in drei 4NF-Relationen erzielt.

Analog zur BCNF gilt allgemein, dass immer eine verlustlose Zerlegung in 4NF-Relationen möglich ist. Aber wir können nicht immer garantieren, dass diese Zerlegung auch die in der Ursprungsrelation geltenden funktionalen Abhängigkeiten erhält. Das ist eine logische Konsequenz aus der Tatsache, dass jede 4NF-Relation auch die BCNF-Kriterien erfüllt.

6.12 Zusammenfassung

Allgemein gelten die in Abbildung 6.6 dargestellten Beziehungen zwischen den Normalformen. Die grafischen Bereichsangaben im rechten Teil des Bildes sollen verdeutlichen, bis zu welchen Normalformen die entsprechenden Zerlegungsalgorithmen Verlustlosigkeit und Abhängigkeitserhaltung garantieren:

- Die Verlustlosigkeit ist für alle Zerlegungsalgorithmen in alle Normalformen garantiert.
- Die Abhängigkeitserhaltung kann nur bei den Zerlegungen bis zur dritten Normalform garantiert werden.

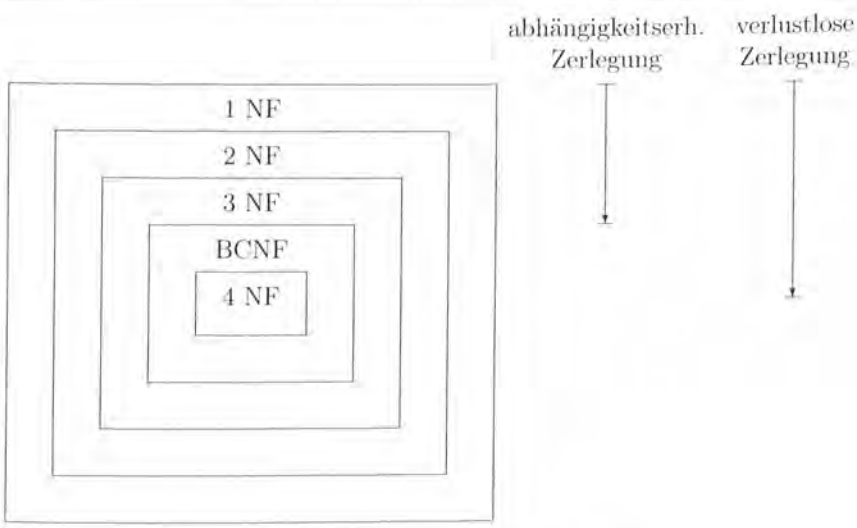


Abbildung 6.6: Beziehungen der Normalformen zueinander

Man sollte die in diesem Kapitel vorgestellte formale Entwurfstheorie aber nur als Feinabstimmung eines solide durchgeführten konzeptuellen Entwurfs ansehen. Keinesfalls sollte der konzeptuelle Entwurf mit der Begründung, das könne man später

im Zuge der Normalisierung „noch richten", nur nachlässig durchgeführt werden. Ein gewissenhafter konzeptueller Entwurf mit einer nachfolgenden systematischen Transformation in das relationale Modell resultiert in der Regel schon in „guten" Relationenschemata, die zumeist die Kriterien der hier vorgestellten Normalformen schon erfüllen.

6.13 Übungen

6.1 Beweisen Sie die Korrektheit der Armstrong-Axiome.

6.2 Zeigen Sie, dass die Armstrong-Axiome minimal sind, d.h. es lässt sich keines der drei Axiome aus den zwei anderen herleiten.

6.3 Zeigen Sie die Korrektheit der drei zusätzlich zu den Armstrong-Axiomen eingeführten Inferenzregeln (Vereinigungsregel, Dekompositionsregel und Pseudotransitivitätsregel) für funktionale Abhängigkeiten, indem Sie diese aus den in Aufgabe 6.1 als korrekt bewiesenen Armstrong-Axiomen herleiten.

6.4 Sei F eine Menge von FDs über dem Relationenschema $\mathcal{R}$. Sei G die Menge aller möglichen FDs über $\mathcal{R}$. Dann ist F^- definiert als $G - F^+$ und wird im Englischen als *exterior* von F bezeichnet. F^- enthält also die FDs, die nicht aus F ableitbar sind.

Zeigen Sie, dass es – unter der Voraussetzung, dass die Domänen der Attribute aus $\mathcal{R}$ unendlich sind (z.B. *integer*) – für jedes $\mathcal{R}$ mit zugehöriger FD-Menge F eine Relationenausprägung R gibt, in der jede FD $f \in F$ erfüllt ist aber keine FD $f' \in F^-$ erfüllt ist. Eine derart konstruierte Relation nennt man nach deren „Erfinder" Armstrong-Relation [Armstrong (1974)].

Illustrieren Sie Ihr Vorgehen an einem (hinreichend großen) Beispiel.

6.5 Zeigen Sie, dass FDs der Art

$$\alpha \rightarrow \beta$$

mit $\beta \subseteq \alpha$ trivial sind.

Zeigen Sie, dass nur FDs dieser Art trivial sind.

6.6 Ist die kanonische Überdeckung F_c einer Menge F von funktionalen Abhängigkeiten eindeutig? Begründen Sie Ihre Antwort.

6.7 Betrachten Sie ein abstraktes Relationenschema $\mathcal{R} = \{A, B, C, D, E, F\}$ mit den FDs

- $A \rightarrow BC$
- $C \rightarrow DA$
- $E \rightarrow ABC$
- $F \rightarrow CD$
- $CD \rightarrow BEF$

Bestimmen Sie hierzu die kanonische Überdeckung.

Berechnen Sie die Attributhülle von A.

Bestimmen Sie alle Kandidatenschlüssel.

6.8 Bringen Sie folgendes Relationenschema[4]

- AssisBossDiplomanden: {[PersNr, Name, Fachgebiet, BossPersNr, BossName, MatrNr, SName, Semester, SWohnOrt]}

mittels des Synthesealgorithmus in die dritte Normalform.

Gehen Sie dabei schrittweise vor, d.h.:

1. Bestimmen Sie die geltenden FDs.
2. Bestimmen Sie die Kandidatenschlüssel.
3. Bestimmen Sie die kanonische Überdeckung der FDs.
4. Wenden Sie den Synthesealgorithmus an.

Dokumentieren Sie jeden Schritt Ihres Vorgehens, so dass man die Methodik erkennen kann.

6.9 Betrachten Sie einen gerichteten Graphen $G = (V, E)$ mit Knotenmenge V und Kantenmenge E. Die Knotenmenge V sei in n Klassen $C_1, \ldots, C_n$ aufgeteilt, so dass gilt:

1. $V = C_1 \cup \cdots \cup C_n$
2. für alle $(1 \leq i \neq j \leq n)$ gilt: $(C_i \cap C_j) = \emptyset$
   D.h. die Klassen sind paarweise disjunkt.

Weiterhin seien nur Kanten der Art (v, v') mit $v \in C_i$ und $v' \in C_{i+1}$ für $(1 \leq i \leq n-1)$ erlaubt. Unter der Annahme, dass von jedem Knoten mindestens eine Kante ausgeht, und jeder Knoten von mindestens einer Kante „getroffen" wird, lässt sich der Graph G als n-stellige Relation wie folgt darstellen:

G			
C_1	C_2	$\ldots$	C_n
$\vdots$	$\vdots$	$\vdots$	$\vdots$

In dieser Relation sind also alle möglichen Pfade, die in einem Knoten $v_1 \in C_1$ anfangen und in einem Knoten $v_n \in C_n$ enden, aufgeführt.

- In welcher Normalform ist die Relation?
- Welche MVDs sind in dieser Relation gegeben?
- Überführen Sie das Schema in die vierte Normalform.

[4] MatrNr, SName, Semester, SWohnOrt sind die Daten der von den Assistenten betreuten Studenten; BossPersNr und BossName sind die Daten der Professoren, bei denen die Assistenten angestellt sind.

6.10 Eine Zerlegung eines Relationenschemas $\mathcal{R}$ in zwei Teil-Schemata $\mathcal{R}_1$ und $\mathcal{R}_2$ ist verlustlos, wenn

- $\mathcal{R}_1 \cap \mathcal{R}_2 \to \mathcal{R}_1$ oder
- $\mathcal{R}_1 \cap \mathcal{R}_2 \to \mathcal{R}_2$

gilt. Beweisen Sie dies.

6.11 Eine MVD $\alpha \twoheadrightarrow \beta$ bezogen auf $\mathcal{R} \supseteq \alpha \cup \beta$ heißt trivial, wenn *jede* mögliche Ausprägung R von $\mathcal{R}$ diese MVD erfüllt. Beweisen Sie, dass $\alpha \twoheadrightarrow \beta$ trivial ist, genau dann wenn

1. $\beta \subseteq \alpha$ oder
2. $\alpha \cup \beta = \mathcal{R}$

gilt. Beachten Sie, dass funktionale Abhängigkeiten nur unter der ersten Bedingung trivial sind.

6.12 Eine Zerlegung eines Relationenschemas $\mathcal{R}$ in zwei Teil-Schemata $\mathcal{R}_1$ und $\mathcal{R}_2$ ist genau dann verlustlos, wenn

- $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_1$ oder
- $\mathcal{R}_1 \cap \mathcal{R}_2 \twoheadrightarrow \mathcal{R}_2$

gilt. Beweisen Sie dies.

6.13 Beweisen Sie, dass die Zerlegung eines Relationenschemas $\mathcal{R}$ in n Teilschemata $\mathcal{R}_1, \ldots, \mathcal{R}_n$ verlustlos ist, wenn $\mathcal{R}$ verlustlos in die zwei Teil-Schemata $\mathcal{R}_1$ und $\mathcal{R}'_2$; $\mathcal{R}'_2$ verlustlos in die zwei Teilschemata $\mathcal{R}_2$ und $\mathcal{R}'_3$; usw. zerlegt wurde.

6.14 Zeigen Sie die Korrektheit der drei zusätzlichen Ableitungsregeln für MVDs:

- *Mehrwertige Vereinigung*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \gamma\beta$.
- *Schnittmenge*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \beta \cap \gamma$.
- *Differenz*: Sei $\alpha \twoheadrightarrow \beta$ und $\alpha \twoheadrightarrow \gamma$. Dann gilt $\alpha \twoheadrightarrow \beta - \gamma$ und $\alpha \twoheadrightarrow \gamma - \beta$.

Diese drei Regeln lassen sich aus den anderen Regeln ableiten. Sie sind also für die Vollständigkeit nicht notwendig.

6.15 Es gibt verlustlose Zerlegungen einer nicht-leeren Relation R in R_1, R_2, R_3, ohne dass überhaupt irgendwelche nicht-trivialen MVDs in der Relationenausprägung erfüllt sind.

Begründen Sie, warum dies kein Widerspruch zu dem in Übungsaufgabe 6.12 bewiesenen Satz darstellt.

Geben Sie ein Beispiel für eine derartige Relation und deren Zerlegung in drei Teilrelationen an.

6.16 Betrachten Sie folgendes Schema:

- ProfessorenAllerlei: {[PersNr, Name, Rang, Raum, VorlNr, VorlTag, Hörsaal, AssiPersNR, AssiName, DiplomandenMatrNr]}.

Dieses Schema erfüllt sicherlich nicht unsere Qualitätsanforderungen.

In welcher Normalform ist das Schema?

- Bestimmen Sie die FDs.
- Bestimmen Sie den/die Kandidatenschlüssel.
- Bestimmen Sie die MVDs.
- Bringen Sie diese Relation in die dritte Normalform.
- Erfüllt das gerade erhaltene 3NF-Schema schon die „schärfere" BCNF? Wenn nein, überführen Sie das 3NF-Schema in ein BCNF-Schema.
- Überführen Sie das ursprüngliche Schema in die 4NF.
- Bringen Sie das vorher hergeleitete BCNF-Schema in die vierte Normalform und vergleichen Sie das Ergebnis mit dem 4NF-Schema, das aus dem ursprünglichen Schema generiert wurde.

6.17 Gegeben sei das folgende Schema:

- Familie: {[Opa, Oma, Vater, Mutter, Kind]}

Hierbei sei vereinfachend vorausgesetzt, dass Personen eindeutig durch ihren Vornamen identifiziert seien. Für ein Tupel [Theo, Martha, Herbert, Maria, Else] soll gelten, dass Theo und Martha entweder die Eltern von Herbert oder von Maria sind – die Großeltern werden also immer als Paar gespeichert, ohne dass ersichtlich ist, ob es die Großeltern väterlicher- oder mütterlicherseits sind. Wir gehen weiterhin davon aus, dass zu einem Kind immer beide Elternteile und beide Großeltern-Paare (also sowohl mütterlicherseits als auch väterlicherseits) bekannt sind.

- Bestimmen Sie alle FDs und MVDs.
  Beachten Sie die Komplementregel.
- Bestimmen Sie den Kandidatenschlüssel der Relation *Familie*.
- Führen Sie für das Schema alle möglichen Zerlegungen in die vierte Normalform durch.

6.18 Die in Abschnitt 6.7 durchgeführte Zerlegung der Relation *StudentenBelegung* ist verlustlos, wie man leicht sieht. Sie ist aber auch abhängigkeitsbewahrend. Warum ist dies so, obwohl die ursprüngliche Schlüssel-FD ja keiner Relation mehr zugeordnet werden kann? Hinweis: Die Abhängigkeitsbewahrung ist über die Hülle definiert.

6.19 Der oben vorgestellte Synthesealgorithmus kann nach Thalheim (2013) unnötig viele Relationen erzeugen. Als Beispiel betrachte man das Schema $\{A, B, C\}$ mit den FDs $A \to B, B \to C, C \to A$. Der Basis-Synthesealgorithmus würde daraus drei Schemata, nämlich $\{A, B\}$, $\{B, C\}$ und $\{A, C\}$ generieren. Eine genaue Analyse würde aber ergeben, dass das Ausgangsschema auch schon normalisiert war. Warum?

Zur Behebung dieses Problems kann man im Synthesealgorithmus nicht nur FDs mit gleicher linker Seite zusammenfassen, sondern zusätzlich noch FDs in Äquivalenzklasse zusammenführen, für die dann nur ein Schema angelegt wird. Und zwar werden zwei FDs $X_1 \to Y_1$ und $X_2 \to Y_2$ in einer Äquivalenzklasse verwaltet, wenn ihre linken Seiten äquivalent sind, wenn also gilt: $X_1 \to X_2$ **und** $X_2 \to X_1$.

Zeigen Sie die Korrektheit dieses verfeinerten Synthesealgorithmus und konstruieren Sie sinnvolle, praxisrelevante Beispiele, für die eine reduzierte Anzahl Relationen durch die Verfeinerung resultiert.

6.14 Literatur

Die relationale Entwurfstheorie geht schon auf das frühe Papier von Codd (1970), dem Erfinder des relationalen Modells zurück – dort sind schon die erste, zweite und dritte Normalform eingeführt worden. Die BCNF-Normalform wurde ebenfalls von Codd (1972a) „nachgereicht".

Die vierte Normalform, basierend auf den mehrwertigen Abhängigkeiten, wurde von Fagin (1977) definiert.

Der Algorithmus zur Synthese eines Relationenschemata in dritter Normalform geht auf Biskup, Dayal und Bernstein (1979) zurück.

Sehr viel ausführlichere Abhandlungen zur relationalen Entwurfstheorie kann man in den Büchern über Datenbanktheorie von Maier (1983), Abiteboul, Hull und Vianu (1995), Kandzia und Klein (1993) finden. Das Buch von Thalheim (1991) widmet sich ganz der auf Abhängigkeiten basierenden relationalen Entwurfstheorie.

Kent (1983) behandelt die relationale Entwurfstheorie auf einer sehr anschaulichen Ebene – dieser kurze Aufsatz sei allen Lesern als Überblick empfohlen.

Die geschachtelten relationalen Datenmodelle wurden in den Achtziger Jahren entwickelt [Schek und Scholl (1986)]. Es wurden auch in Deutschland zwei renommierte Prototypen basierend auf dem NF^2-Modell entwickelt: AIM [Dadam et al. (1986)] wurde am Wissenschaftlichen Zentrum der IBM in Heidelberg und DASDBS [Schek et al. (1990)] an der Universität Darmstadt realisiert.

7. Physische Datenorganisation

Während des konzeptuellen und logischen Entwurfs untersucht man, welche Daten benötigt werden, und wie sie zusammenhängen. Die effektive Organisation der Daten und des Zugriffs auf den Hintergrundspeicher wird durch den physischen Entwurf festgelegt. Um einen auf eine Anwendung und ein Datenbanksystem sinnvoll zugeschnittenen physischen Entwurf bestimmen zu können, müssen zumindest grundlegend die Methodiken der Datenspeicherung und die Auswirkungen der verschiedenen Entwurfsstrategien auf die Leistung des Systems bekannt sein.

Zunächst werden in diesem Kapitel die Charakteristika der verschiedenen Speichermedien eines Computersystems und die Abbildung von Relationen auf diese Speichermedien betrachtet. Zur Unterstützung bestimmter Verhaltensmuster einer Anwendung werden Indexstrukturen und die sogenannte *Objektballung* eingeführt.

Maßgebliche Faktoren beim physischen Entwurf sind die Zugriffszeit, der Aufwand für die Wartung und der Platzbedarf der Daten. Zum Abschluss wird kurz darauf eingegangen, wie die wichtigen Eigenschaften des Anwendungsverhaltens erkannt und in Bezug auf diese Faktoren unterstützt werden können.

7.1 Speichermedien

Man unterscheidet meist drei Stufen von Speichermedien: Primärspeicher, Sekundärspeicher und Archivspeicher. Bei vielen Datenbanksystemen werden alle drei Speichermedien gleichzeitig eingesetzt, allerdings für unterschiedliche Zwecke.

Der Primärspeicher ist der Hauptspeicher des Rechners. Charakteristisch für den Hauptspeicher ist, dass er sehr teuer, sehr schnell und im Allgemeinen, im Vergleich zur benötigten Datenmenge, eher klein ist. Die Granularität des Hauptspeichers ist sehr fein. Es ist möglich, auf beliebige Adressen direkt zuzugreifen. Alle Operationen auf Daten müssen im Hauptspeicher durchgeführt werden, der allerdings i.A. nicht gegen Systemausfälle gesichert ist. Er übernimmt in Datenbanksystemen daher Pufferfunktionen.

Ein typischer Sekundärspeicher ist die Festplatte. Der Zugriff auf Daten im Sekundärspeicher ist gegenüber dem Primärspeicher um etwa einen Faktor 10^5 langsamer. Dafür bietet der Sekundärspeicher aber wesentlich mehr Platz, ist relativ ausfallsicher und günstiger im Preis. Auch mit einer Festplatte ist ein Direktzugriff möglich, aber mit einer gröberen Granularität. Die kleinste Einheit des Zugriffs auf eine Festplatte ist ein Block. In Datenbanksystemen wird als kleinste Einheit meistens eine Seite verwendet. Eine Seite fasst mehrere, in einer Spur liegende Blöcke zusammen. Abbildung 7.1 skizziert den typischen Aufbau einer Festplatte. In größeren Laufwerken sind üblicherweise mehrere Platten übereinander auf einer Achse montiert, wie die Seitenansicht andeutet. Die Schreib-/Leseköpfe dieser Platten bewegen sich synchron, d.h. sie stehen alle auf übereinander liegenden Spuren. Die übereinander liegenden Spuren nennt man Zylinder.

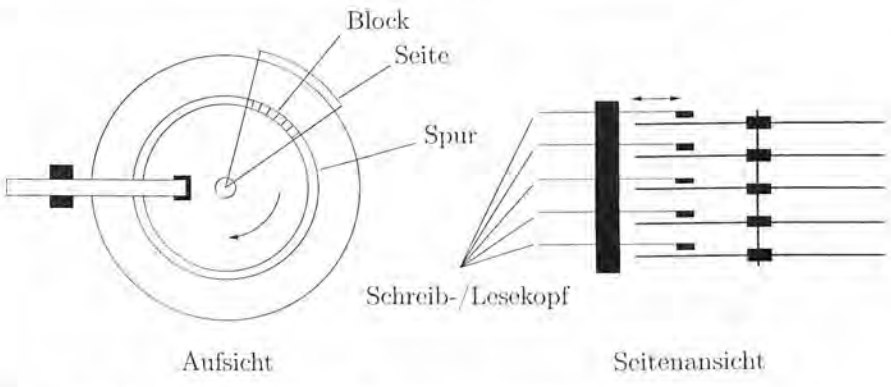


Abbildung 7.1: Schematischer Aufbau einer Festplatte

Für einen Zugriff auf einen bestimmten Block werden drei Arbeitsvorgänge benötigt. Zunächst muss der Schreib-/Lesekopf auf die entsprechende Spur plaziert werden. Die dazu benötigte Zeit wird als *Seek Time* bezeichnet. Dann wird gewartet, bis sich durch die Rotation der Platte der gesuchte Block am Kopf vorbeibewegt. Die zu erwartende Verzögerung wird *Latenzzeit* genannt. Ein Zugriff ist dementsprechend am schnellsten, wenn sich der Kopf bereits in der passenden Spur befindet, und die Blöcke innerhalb der Spur sequentiell gelesen werden. Im dritten Schritt wird der Block gelesen (*Lesezeit*). Den größten Anteil nimmt i.A. die Seek Time ein. Zusätzlich zu den rein mechanischen Arbeitsvorgängen entsteht außerdem noch ein nicht unerheblicher Programmaufwand für die Übertragung, Dekodierung und Verwaltung der von der Festplatte eingelesenen Blöcke.

Als Archivspeicher werden vielfach Magnetbänder verwendet. Heutzutage erreicht oder überschreitet die Kapazität von Festplatten häufig die der Bänder, jedoch liegt der Preis von Bandmaterial im Bereich von Cent pro Megabyte. Ein Band kann nur sequentiell gelesen und beschrieben werden, die Zugriffszeit ist daher nicht direkt vergleichbar. Im Datenbankeinsatz sind Archivspeicher gerade auch wegen ihrer Ausfallsicherheit für die Protokollierung von Operationen wichtig (siehe dazu Kapitel 10).

7.2 Speicherhierarchie

In Abbildung 7.2 ist die Speicherhierarchie veranschaulicht. Die geometrische Form des Dreiecks wurde bewusst so gewählt, um die von unten nach oben abnehmende Speicherkapazität zu dokumentieren. Während moderne Prozessoren nur ein paar Dutzend (bis Hunderte) von Registern mit einer Kapazität von 8 Byte vorweisen, ist der Speicherplatz auf Archivspeichern praktisch unbeschränkt. Die Abbildung zeigt zudem die typischen Zugriffszeiten für die einzelnen Speichermedien. Die Register können innerhalb eines Taktzyklus zugegriffen werden, so dass deren Zugriffszeit in der Größenordnung von unter 1 ns liegt. Auf der nächsten Stufe in der Speicherhierarchie befinden sich die Prozessorcaches, von denen es mehrere Level gibt: Die Kapazität des Levels L1 beträgt einige hundert KB, wohingegen im Level L2

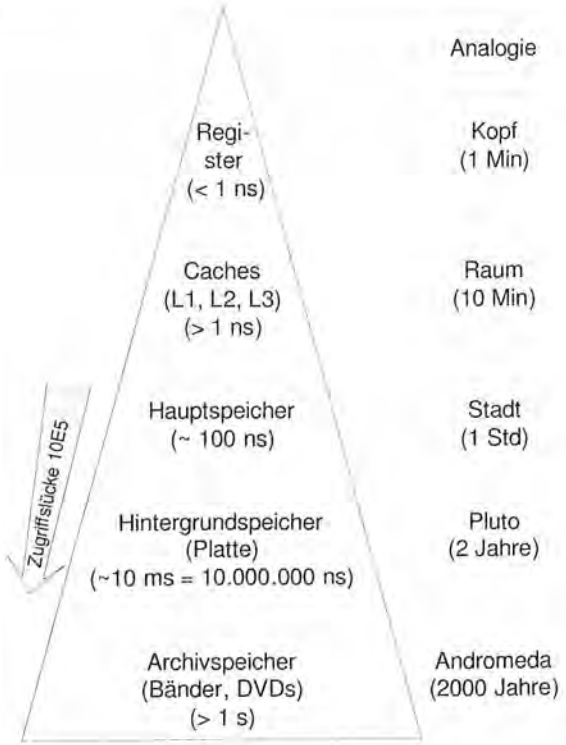


Abbildung 7.2: Visualisierung der Speicherhierarchie

schon etliche MB zur Verfügung stehen können. Der Level1- und teilweise auch schon der Level2-Cache sind in den Prozessor integriert und arbeiten mit dem vollen Prozessortakt während der Level3-Cache mit einigen hundert Megahertz getaktet ist. In neueren Datenbankarchitekturen, wie z.B. den *Column-Stores* und den Hauptspeicher-Datenbanksystemen, ist man bestrebt, die Datenverarbeitung dahingehend zu optimieren, dass die Caches besser ausgenuzt werden. Dies geschieht durch optimierte Speicherstrukturen, die darauf abzielen, die Daten physisch benachbart zu speichern, die zeitlich benachbart zugegriffen/verarbeitet werden. Dadurch wird die *Cache-Lokalität* erhöht, so dass sogenannte *Cache-Hits* häufiger und *Cache-Misses* seltener vorkommen. In einem Datenbank-Server hat man in der Regel einen Hauptspeicher mit einer Kapazität von vielen GB, um das Auslagern häufig zugegriffener Seiten auf den Hintergrundspeicher zu vermeiden. Dies ist absolut essentiell für den leistungsfähigen Betrieb eines Datenbanksystems, da die Zugriffszeit des Hauptspeichers im Bereich von einigen hundert ns liegt wohingegen die Zugriffszeit des Hintergrundseichers bis zu 10 ms betragen kann. Der relative Unterschied beläuft sich also auf einen Faktor von 10^5, den man als *Zugriffslücke* bezeichnet. Gray und Graefe (1997) haben die sogenannte 5-Minuten-Regel postuliert, wonach jede Seite, auf die im Abstand von fünf Minuten zugegriffen wird, im Hauptspeicher verbleiben sollte. Die Archivspeicher sind nochmals langsamer und größer im

Vergleich zu den Disk-basierten Hintergrundspeichern.

Von Jim Gray stammt auch die auf der rechten Seite der Abbildung gezeigte Analogie, mit der man sich die relativen Zugriffsunterschiede klar machen kann. Wenn man die Register mit den im Kopf memorisierten Daten assoziiert, dann entsprechen die Cachedaten solchen Objekten, die sich im selben Raum befinden. Die Hauptspeicher-residenten Daten befinden sich dann in derselben Stadt und sind innerhalb von einer Stunde zugreifbar. Die dramatische relative Zugriffslücke zwischen Hauptspeicher und Hintergundspeicher wird durch den Vergleich zwischen einem Datenobjekt in derselben Stadt und einem per Rakete mit einer Flugzeit von 2 Jahren vom (Zwerg-)Planeten Pluto zu holenden Datenobjekt klar gemacht. Archivierte Daten entsprechen solchen, die sich in der Andromeda-Galaxie befinden und erst nach 2000 jähriger Flugzeit verfügbar sind. Bei der Konzeption von Datenverarbeitungsalgorithmen sollte man diese Analogie beherzigen und die Algorithmen so entwickeln, dass man möglichst selten zum Pluto fliegen muss und wenn schon, dann dafür sorgt, dass die Rakete möglichst voll beladen ist (*chained I/O*) und nur für die Bearbeitung nützliche Daten enthält. Letzteres ist das Ziel der Objektballung, die in Abschnitt 7.15 behandelt wird.

7.3 Speicherarrays: RAID

Die Wartezeiten durch die mechanischen Arbeitsvorgänge in einer Festplatte sind nur schwer reduzierbar. Trotz der hohen Rotations- und Übertragungsgeschwindigkeiten moderner Laufwerke ist es nicht gelungen, die bereits erwähnte Zugriffslücke zwischen Haupt- und Hintergrundspeicher zu verkleinern – ganz im Gegenteil, die Lücke wird eher größer als kleiner.

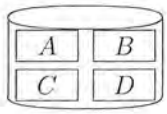
Die RAID-Technologie (*redundant array of inexpensive disks*) nutzt aus, dass man anstelle eines einzigen (entsprechend großen) Laufwerks effizienter mehrere (entsprechend kleinere und billigere) Laufwerke parallel betreiben kann. Die preiswerten Laufwerke arbeiten durch einen entsprechenden *RAID-Controller* nach außen transparent wie ein einziges logisches (virtuelles) Laufwerk mit vielen unabhängigen Schreib-/Leseköpfen.

Man unterscheidet bis zu acht *RAID-Level*: RAID 0 bis 6 und RAID 0 + 1 (oder RAID 10). Ein höherer RAID-Level bedeutet nicht unbedingt eine Leistungssteigerung, vielmehr existieren die Stufen nebeneinander und optimieren unterschiedliche Zugriffsprofile.

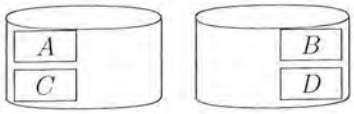
In **RAID 0** wird die Datenmenge des logischen Laufwerks durch blockweise Rotation auf die physischen Laufwerke verteilt. Existieren also beispielsweise zwei Laufwerke, erhält Laufwerk 1 die Datenblöcke $A, C, E, \ldots$ des logischen Laufwerkes, Laufwerk 2 die Blöcke $B, D, F, \ldots$ – wie in Abbildung 7.3 (b) dargestellt. Dieses Vorgehen nennt man *Striping*. Die Größe der Datenblöcke nennt man die *Stripinggranularität* und die Anzahl der Platten die *Stripingbreite*.

Wird nun eine Menge aufeinanderfolgender Blöcke vom Controller angefordert, kann er diese Anforderung auf die Laufwerke verteilen, die sie dann parallel bearbeiten können. Der Controller „sammelt" anschließend die Ergebnisse ein und fügt sie wieder zu einer logischen Einheit (z.B. $A, B, C, D, \ldots$) zusammen. Damit skaliert – für größere angefordete Datenmengen – die Bearbeitungsgeschwindigkeit nahezu

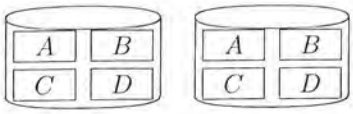
(a) virtuelle/logische Platte (hier mit vier Datenblöcken)



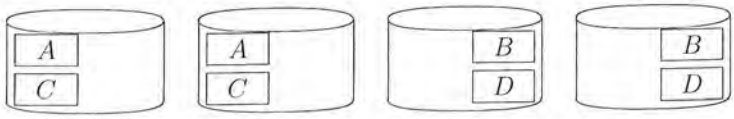
(b) RAID 0: Striping der Blöcke (hier auf nur zwei Platten)



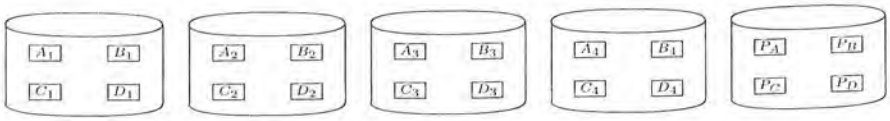
(c) RAID 1: Spiegelung (mirroring)



(d) RAID 0+1: Striping und Spiegelung



(e) RAID 3: Bit-Level-Striping + separate Parity-Platte



(f) RAID 5: Block-Level-Striping + verteilte Parity-Blöcke

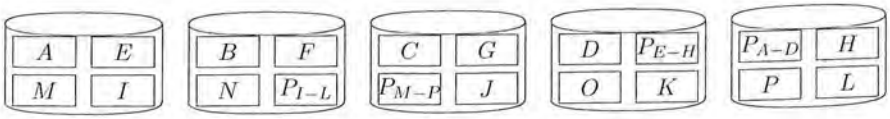


Abbildung 7.3: Illustration verschiedener RAID-Level

linear mit der Anzahl der vorhandenen Laufwerke.

Das zufällige Lesen einzelner Blöcke wird nicht so effektiv beschleunigt, da man innerhalb der einzelnen Anforderung eines Blockes natürlich keine Parallelität ausnutzen kann. Aber es tritt eine Lastbalancierung ein, wenn viele zufällig plazierte Blöcke von unterschiedlichen Prozessen parallel angefordert werden. Die Warteschlangen, die bei der Abarbeitung der Anforderungen entstehen, werden dann auf die einzelnen Festplatten verteilt und sind daher kürzer als im RAID-losen Fall.

Man beachte, dass RAID 0 bei entsprechend großer Anzahl von physischen Platten sehr fehleranfällig ist: Eine Datei wird auf alle physischen Laufwerke verteilt. Wenn auch nur eines dieser Laufwerke ausfällt, bewirkt dies wegen des Stripings den Verlust der Datei. Bei einer großen Zahl von physischen Laufwerken, sagen wir 100, beträgt bei heutiger Plattentechnologie die durchschnittliche Zeit bis zum Ausfall eines dieser Laufwerke nur etwa einen Monat (siehe Übungsaufgabe 7.1).

Während RAID 0 also auf eine möglichst große Beschleunigung von Anforderungen abzielt, berücksichtigt **RAID 1** auch die Datensicherheit. Das Prinzip ist wieder recht einfach: Jedes Laufwerk besitzt eine sogenannte Spiegelkopie (engl. *mirror*), die die gesamte Datenmenge redundant enthält – siehe Abbildung 7.3 (c). Fällt eines dieser beiden Laufwerke aus oder enthält defekte Blöcke, kann der RAID-Controller ohne Unterbrechung weiterarbeiten und das noch funktionierende Laufwerk verwenden. Leseoperationen werden auf die beiden Laufwerke verteilt, so dass jedes Laufwerk nur noch etwa die Hälfte der Leseanforderungen an die logische Platte zu bearbeiten hat. Schreiboperationen auf Blöcken müssen auf beiden Kopien durchgeführt werden, wobei auch hier der physische Schreibvorgang parallel stattfindet.

RAID 0+1 kombiniert einfach RAID 0 und RAID 1. Die Datenblöcke werden auf mehrere Laufwerke aufgeteilt und von diesen Laufwerken existieren Kopien – siehe Abbildung 7.3 (d). Es ist offensichtlich, dass man bei RAID 1 und RAID 0+1 einen doppelten Speicherplatzbedarf hat.

Ab dem RAID-Level 2 werden Paritätsinformationen verwendet, um auf eine ökonomisch günstigere Weise Datensicherheit anzubieten als RAID 1 und RAID 0+1. Dabei wird über mehrere Daten eine Art Prüfsumme – genauer, die Parität – berechnet und abgespeichert. Mit dieser Prüfsumme kann man dann feststellen, ob die Daten, die zur Berechnung verwendet wurden, noch korrekt sind und eine entsprechende Fehlerkorrektur anbringen. Wenn man mit dem Konzept der Parität nicht vertraut ist, kann man sich die Fehlerkorrektur mittels Paritätsinformation so vorstellen: Man speichert zu N Datenbereichen, die auf unterschiedlichen Platten liegen, zusätzlich deren (Prüf-)Summe auf einer anderen Platte ab. Wenn nun einer dieser N Datenbereiche (bzw. deren Platte) defekt ist, kann man den Wert dieses Datenbereichs aus der (Prüf-)Summe minus der Summe der noch intakten $N-1$ Datenbereiche wiederherstellen.

RAID 2 führt ein Striping auf Bitebene durch und verwendet zusätzliche Platten zur Speicherung von Paritätsinformationen in Form von Fehlererkennungs- und Korrekturcodes ähnlich denen von Bandlaufwerken. Es wird allerdings in der Praxis selten eingesetzt, da die Plattencontroller sowieso schon eine Fehlererkennung eingebaut haben.

RAID 3 und **RAID 4** verwenden für die Paritätsinformationen eine einzige, dedizierte Festplatte. Diese Paritätsinformation dient nur zur Fehlerkorrektur, wenn eine der Datenplatten (bzw. ein Speicherbereich darauf) defekt ist. Das Grundschema

für eine Konfiguration mit vier Datenplatten für „Stripes" und einer Paritätsplatte ist in Abbildung 7.3 (e) dargestellt.

In RAID 3 werden die Daten bit- oder byteweise auf die Datenplatten verteilt. In unserer Grafik ist dieses Verfahren für vier Datenplatten gezeigt. Dabei wird das erste Bit (oder Byte) eines Datenblocks auf die erste Platte, das zweite auf die zweite Platte, usw verteilt. Das fünfte Bit wird wiederum auf die erste Platte plaziert. Wenn wir die Bits des Datenblocks A mit $A[1], A[2], A[3], \ldots$ bezeichnen, enthält also das Stripe A_1 die Bits $A[1], A[5], A[9], A[13], \ldots$. Generell wird demnach bei vier Platten das Bit $A[i]$ auf die Platte i mod 4 plaziert. Die rechts eingezeichnete Paritätsplatte enthält die Paritätsinformation, die sich bitweise aus den zugehörigen Stripes wie folgt errechnet:

$$A[1] \oplus A[2] \oplus A[3] \oplus A[4] \ , \ A[5] \oplus A[6] \oplus A[7] \oplus A[8] \ , \ \cdots$$

wobei $\oplus$ das „exklusive oder" bezeichnet. Es wird also in einem Bit auf der Paritätsplatte abgespeichert, ob in den korrespondierenden vier Bits der vier Datenplatten eine ungerade (Parität = 1) oder eine gerade (Parität = 0) Anzahl Bits gesetzt sind. Bei N Datenplatten, die mittels einer Paritätsplatte gesichert werden, hat man demnach einen erhöhten Speicherbedarf von $1/N$ gegenüber der (unsicheren) RAID-losen Speicherung.

Bei RAID 3 muss eine Leseanforderung auf alle Datenplatten zugreifen um einen logischen Datenblock zu rekonstruieren – die Paritätsplatte wird beim Lesen nur in Fehlerfällen verwendet. Eine Schreibanforderung benötigt sowohl die Datenplatten als auch die Paritätsplatte um die Paritätsinformationen neu zu berechnen.

RAID 4 verteilt die Daten wieder blockweise auf die Platten und kann daher effizienter als RAID 3 mit kleinen Leseanforderungen umgehen. Dies geht zu Lasten von Schreibanforderungen: Hier muss sowohl der alte Inhalt des Datenblocks als auch der Paritätsblock gelesen werden. Anschließend wird der neue Inhalt des Datenblocks und die korrigierte Parität geschrieben. Insbesondere nachteilig ist, dass jede Schreiboperation auf die eine Paritätsplatte zugreifen muss.

RAID 5 arbeitet ähnlich wie RAID 4, verteilt jedoch die Paritätsinformationen auf alle Laufwerke. Das ist in Abbildung 7.3 (f) dargestellt. Bei RAID 4 konnten die Leseoperationen nicht alle Laufwerke verwenden, da ja eines für die Parität verwendet wurde, und Schreiboperationen verwendeten immer die (einzige) Paritätsplatte. Dieser Flaschenhals durch die Paritätsplatte ist durch die geänderte Verteilung effektiv beseitigt. Nach wie vor ist aber der Overhead von Schreiboperationen nicht zu vernachlässigen, da das Schreiben eines Datenblocks die Neuberechnung des zugehörigen Paritätsblocks voraussetzt. Dazu müssen die alten Zustände des Datenblocks und des Paritätsblocks gelesen werden, der neue Paritätsblock aus dem alten und dem neuen Zustand des Datenblocks und dem alten Zustand des Paritätsblocks berechnet werden (wie?) und dann die neuen Zustände des Datenblocks und des Paritätsblocks geschrieben werden.

RAID 6 ist eine Verbesserung der Fehlerkorrekturmöglichkeiten von RAID 5, auf die hier nicht näher eingangen werden soll. Wichtig ist, dass RAID 3 und 5 nur höchstens einen Fehler in den für ein Paritätsdatum verwendeten Daten korrigieren können.

Welches dieser RAID-Level für eine gegebene Anwendung zu bevorzugen ist,

hängt natürlich vom Anwendungsprofil (z.B. Anteil der Leseoperationen im Vergleich zu Schreiboperationen) und von der zu erzielenden Ausfallsicherheit ab.

Heutige kommerziell verfügbare RAID-Systeme erlauben oft eine flexible Konfiguration auf den für das jeweilige Anwendungsgebiet optimalen RAID-Level. Im Fehlerfall, wenn also eine Platte in dem Plattenarray ausfällt, können diese Systeme automatisch eine vorab installierte Ersatzplatte (ein sogenanntes *hot spare*) aktivieren und den Datenbestand der ausgefallenen Platte rekonstruieren und auf diese Ersatzplatte schreiben.

Viele Datenbanksysteme unterstützen das Striping von Datensätzen (Tupeln) auf unterschiedliche Platten auch dann, wenn keine RAID-Systeme eingesetzt werden. Bei einigen Systemen kann man die Plazierung der Datensätze nach semantischen Kriterien (also nach dem Wert bestimmter Attribute) steuern, um dadurch eine bessere Lastbalancierung der eingesetzten Platten zu erzielen.

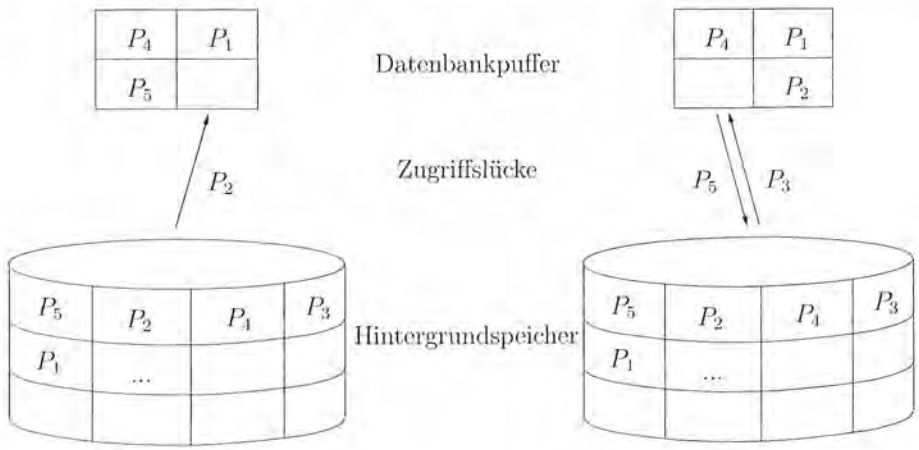
Trotz der Fehlertoleranz von RAID-Systemen, seien die Leser eindringlich davor gewarnt, die systematische Archivierung und Protokollierung von Datenbankzuständen für die Fehlerrecovery – wie sie in Kapitel 10 behandelt wird – zu vernachlässigen. Man beachte, dass die meisten RAID-Level nur den gleichzeitigen Ausfall einer einzigen Platte tolerieren. Normalerweise stehen aber alle Platten des RAID-Systems in demselben Raum, so dass sie durch äußere Einflüsse (Feuer, Wasser, etc.) gefährdet sind. Der Einsatz von RAID-Systemen kann also nur dazu dienen, die mittlere Zeitdauer bis zu einer nötigen Datenbankrecovery zu erhöhen. RAID-Systeme machen die Archivierung und Protokollierung für die Datenbankrecovery aber nicht obsolet!

7.4 Der Datenbankpuffer

Im vorigen Abschnitt wurde erwähnt, dass alle Operationen auf Daten innerhalb des Hauptspeichers durchgeführt werden müssen. Es kann also nicht direkt auf den Seiten des Hintergrundspeichers gearbeitet werden, sie werden vor der Bearbeitung in den sogenannten *Datenbankpuffer* gelesen.

Es ist sehr sinnvoll, Seiten auch länger im Hauptspeicher zu halten als nur für den Zeitraum der Operation, für die diese angefordert wurden. Meistens beobachtet man nämlich im Verhalten der Anwendungen eine *Lokalität*: Es wird mehrmals hintereinander auf ein und dieselben Daten zugegriffen. Sind die Daten dann noch im Hauptspeicher vorhanden, brauchen sie nicht ein weiteres Mal vom Hintergrundspeicher geladen zu werden. Es entsteht ein erheblicher Laufzeitgewinn, wenn man die „Zugriffslücke", also den oben erwähnten Faktor von etwa 10^5 zwischen Hauptspeicher- und Hintergrundspeicherzugriffen, in Betracht zieht.

Da aber der Hauptspeicher nicht nur wesentlich schneller, sondern auch wesentlich kleiner als der Hintergrundspeicher ist, kann eine Seite nicht ewig gepuffert bleiben. Irgendwann müssen alte Seiten aus dem Puffer entfernt werden, um Platz für neue zu machen. Üblicherweise enthält der Datenbankpuffer eine feste Anzahl von *Pufferrahmen*, also Speicherbereichen von der Größe einer Seite. Wenn diese Pufferrahmen alle gefüllt sind, wird eine Seite ersetzt und unter Umständen, falls sie modifiziert wurde, zurück auf den Hintergrundspeicher geschrieben. Die Auswahl einer zu ersetzenden Seite hängt von der *Ersetzungsstrategie* ab. Idealerweise



(a) Laden von P_2 (b) Ersetzen von P_5

Abbildung 7.4: Pufferverwaltung in DBMS

sollte eine Seite entfernt werden, die möglichst lange nicht mehr gebraucht wird. Im Rahmen dieses Buchs können wir aber nicht genauer auf Ersetzungstrategien eingehen.

Das Zusammenwirken zwischen Datenbankpuffer und Hintergrundspeicher ist in Abbildung 7.4 skizziert. Man beachte, dass es nicht immer möglich ist, „logisch benachbarte" Datenbankseiten direkt hintereinander auf die Blöcke des Hintergrundspeichers zu schreiben. Das ist durch die ungeordnete Seitennummerierung angedeutet.

In Abbildung 7.4 (a) wird gerade die Seite P_2 in den freien Rahmen im Datenbankpuffer eingelesen. Nach dem Einlesen von P_2 sind Zugriffe auf Daten, die sich in den Seiten P_1, P_2, P_4 und P_5 befinden ohne Umweg über den Hintergrundspeicher und daher sehr effizient durchführbar. Um Daten zu lesen, die sich auf Seite P_3 befinden, muss eine Seite des Puffers frei gemacht werden. Das ist in Abbildung 7.4 (b) gezeigt: Dort wurde P_5 entfernt um Platz für P_3 zu machen. Wenn die Seite P_5 im Puffer geändert worden war, muss sie auf den Hintergrundspeicher zurückgeschrieben werden – andernfalls kann man sie einfach überschreiben.

7.5 Abbildung von Relationen auf den Sekundärspeicher

Für eine geeignete Abbildung von Relationen auf den Sekundärspeicher und eine gute Unterstützung des Zugriffs muss man sich an den Merkmalen des Speichermediums orientieren.

Eine naheliegende Abbildung ist die folgende: Für jede Relation werden mehrere Seiten auf dem Hintergrundspeicher zu einer Datei zusammengefasst. Die Tupel einer Relation werden in den Seiten der Datei so gespeichert, dass sie nicht über

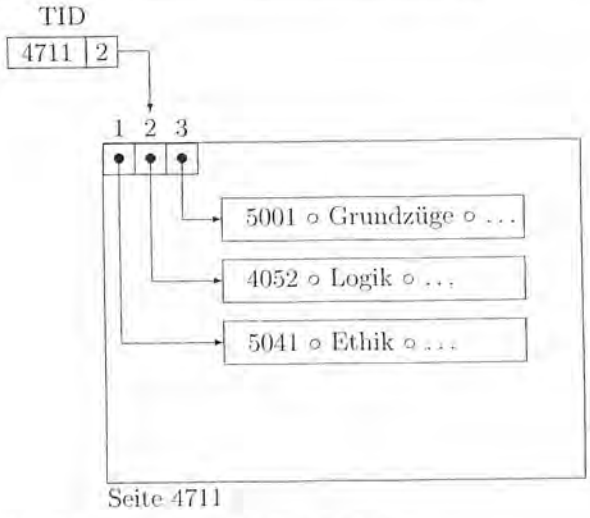


Abbildung 7.5: Speicherung von Tupeln auf Seiten

eine Seitengrenze hinausgehen.[1] Jede Seite enthält eine interne Datensatztabelle, die Verweise auf alle auf der Seite befindlichen Tupel verwaltet. In Abbildung 7.5 sind einige Tupel der Relation *Vorlesungen* in eine Seite eingetragen.

Um ein bestimmtes Tupel direkt referenzieren zu können, beispielsweise durch eine der später in diesem Kapitel vorgestellten Indexstrukturen, verwendet man einen sogenannten *Tupel-Identifikator* (TID). Ein TID besteht aus zwei Teilen: Einer Seitennummer und einer Nummer eines Eintrags in der internen Datensatztabelle, der auf das entsprechende Tupel verweist. Im Fall von Abbildung 7.5 verweist also der TID (4711, 2) auf das Tupel, das zur Vorlesung „Logik" gehört.

Diese zusätzliche Indirektion ist nützlich, wenn die Seite intern reorganisiert werden muss. Nehmen wir an, dass sich das Tupel zur Logikvorlesung vergrößert, auf der Seite aber noch genug Platz vorhanden ist. Dann kann es einfach wie in Abbildung 7.6 verschoben werden, ohne dass sich der zugehörige TID verändert. Daher bleiben auch alle Verweise auf dieses Tupel gültig.

Abbildung 7.7 demonstriert den Fall, dass sich das Tupel weiter vergrößert und nicht mehr genug Platz auf der Seite vorhanden ist. Es muss auf eine andere Seite transferiert werden. Um trotzdem die Verweise invariant zu halten, wird an der alten Position des Tupels eine Markierung hinterlassen, wo es jetzt zu finden ist. Das erfordert natürlich beim Lesen des Tupels mit TID (4711, 2) einen zusätzlichen Seitenzugriff, der vorher nicht notwendig war. Bei nochmaliger Verdrängung dieses Tupels von der Seite 4812 würde aber kein weiterer Platzhalter eingefügt, sondern die Markierung auf der Heimatseite 4711 geändert. Deshalb ist die Länge einer solchen Verweiskette auf maximal zwei beschränkt.

In unserem Beispiel enthielt eine Seite nur Tupel einer Relation. Das ist aber

[1] Außer einem Geschwindigkeitsverlust würde eine nicht an Seitengrenzen orientierte Verteilung auch Probleme bei der Adressierung, der Mehrbenutzersynchronisation und der Fehlerbehandlung hervorrufen.

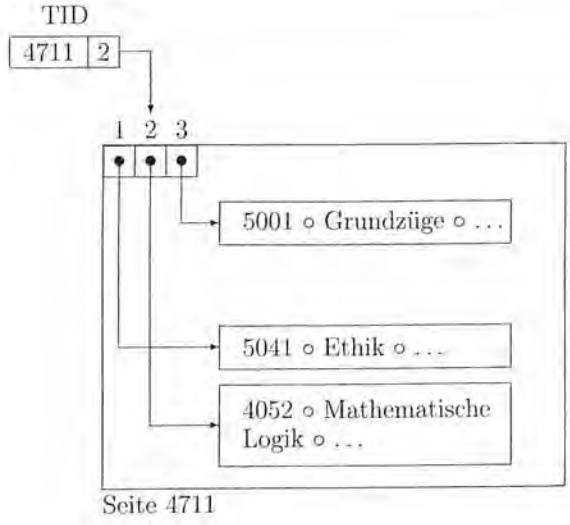


Abbildung 7.6: Verschieben von Tupeln innerhalb einer Seite

nicht zwingend notwendig. Es ist unter Umständen sehr nützlich, Tupel unterschiedlicher Relationen zusammen abzulegen, um dadurch die Lokalität einer Anwendung zu erhöhen. In Abschnitt 7.15 wird diese Variante als „verzahnte Objektballung" besprochen.

7.6 Indexstrukturen

Vielfach werden bei Anfragen auf die Datenbasis nur einige wenige Tupel einer Relation benötigt. Wenn die Datensätze allerdings ohne weitere Zusatzinformationen in den Dateien gespeichert sind, muss die ganze Datei durchsucht werden, um die ein bestimmtes Kriterium erfüllenden Tupel zu finden. Sinnvoller wäre es, die Direktzugriffsmöglichkeiten des Sekundärspeichers auszunutzen. Dazu dienen die im folgenden besprochenen Indexstrukturen, die zu einem gegebenen Suchkriterium die passenden Datensätze in einer Datei angeben.

Aber die Verbesserung des Zugriffs bekommt man nicht geschenkt: Wie alle anderen Informationen müssen auch Indices gewartet werden und benötigen einen gewissen Platz. Gegen Ende dieses Kapitels wird untersucht, wann das Anlegen eines Indexes vorteilhaft ist.

Man unterscheidet zwischen *Primär-* und *Sekundärindices*. Primärindices legen die physische Anordnung der indizierten Daten fest. Daher kann es für jede Datei nur einen Primärindex geben, aber mehrere Sekundärindices. In den meisten Fällen wird der Primärschlüssel einer Relation auch vom Primärindex indiziert.

Üblicherweise spricht man bei dem für den Index verwendeten Suchkriterium vom *Schlüssel* des Indexes. Dieser Schlüsselbegriff hat nichts mit den bisher eingeführten Schlüsseln zu tun. Es ist durchaus möglich, z.B. die Anzahl der Semester der Studenten als Suchkriterium für einen Index zu verwenden, obwohl *Semester* kein Schlüssel der Relation *Studenten* ist. Die in diesem Kapitel erwähnten Schlüssel sind

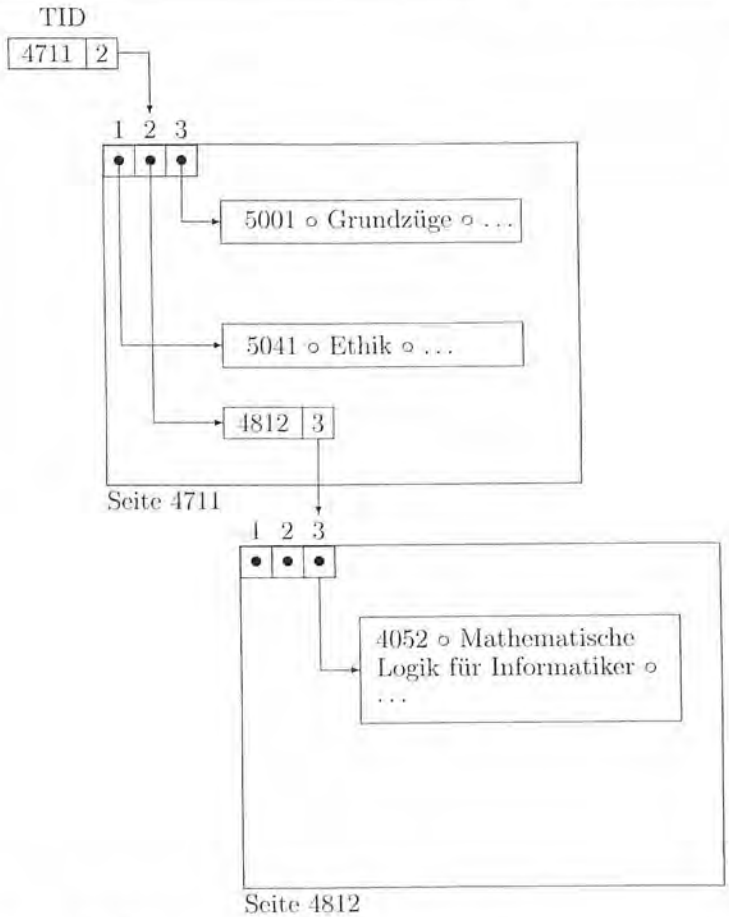


Abbildung 7.7: Verdrängung eines Tupels von einer Seite

alle als Suchschlüssel und nicht als Schlüssel von Relationen zu verstehen.

7.7 ISAM

Eine sehr einfache und unter bestimmten Voraussetzungen auch sehr effektive Indexstruktur ist die *Index-Sequential Access Method* (*ISAM*). Sie ist am ehesten mit einem Daumenindex auf dem Schnitt eines Buchs zu vergleichen, wie man ihn gelegentlich bei Wörterbüchern oder Lexika findet. Beim Nachschlagen wählt man zunächst über den Daumenindex einen Bereich aus, in dem sich das gesuchte Wort befinden müsste, falls es überhaupt vorhanden ist, und sucht es dann dort.

Abbildung 7.8 zeigt schematisch den Aufbau eines ISAM-Indexes. Sowohl der Index als auch die Datensätze D_i werden nach den Schlüsseln S_j geordnet abgespeichert. Ein Datensatz D_i besteht also aus dem Schlüssel S_i und weiteren Informationen (Attributen), die wir im folgenden aber vernachlässigen werden. Der Index

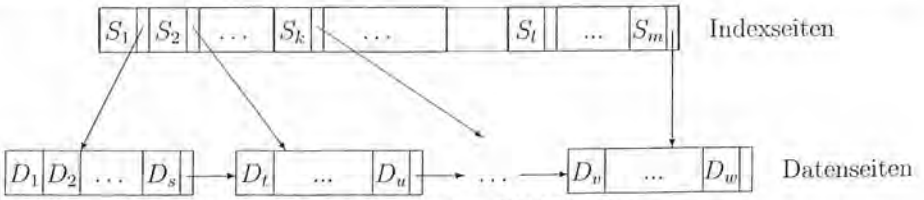


Abbildung 7.8: Schematischer Aufbau eines ISAM Indexes

befindet sich auf Seiten, die sequentiell hintereinander auf dem Sekundärspeicher abgelegt sind.

Innerhalb einer Seite des Indexes werden abwechselnd Schlüssel und Verweise abgespeichert. Ein Verweis zwischen Schlüssel S_i und S_{i+1} zeigt auf die Seite mit denjenigen Datensätzen, die einen Schlüsselwert größer als S_i und kleiner oder gleich S_{i+1} haben. Zur Vereinfachung wird angenommen, dass S_1 eine Art $-\infty$ des Wertebereiches des indizierten Schlüssels annimmt und keine Duplikate in den Suchschlüsseln vorkommen.

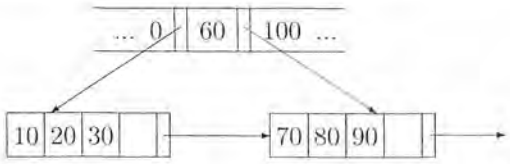
Suchen eines Schlüssels. Innerhalb des Indexes kann durch die sequentielle Anordnung der Seiten mit einer Binärsuche gearbeitet werden, um einen bestimmten Schlüsselwert oder ein Intervall zu finden. Ist die Position des Wertes gefunden, kann der zugehörige Verweis zur Datenseite verfolgt werden. Von dieser Datenseite an kann man wegen der Sortierung solange alle weiteren Datenseiten lesen, bis ein Datensatz gefunden wird, der nicht mehr das vorgegebene Suchkriterium erfüllt.

Einfügen eines Schlüssels. Die Einfachheit des Aufbaus und der Suche schlägt sich leider negativ in der Wartung nieder. Das Einfügen von Datensätzen zieht unter Umständen einen sehr hohen Aufwand nach sich, wenn die Datenseite gefüllt ist, in die der einzufügende Satz gemäß des Suchschlüssels gehört. Zuerst wird dann ein Ausgleich mit Nachbarseiten angestrebt, d.h. ein Datensatz wird in eine benachbarte Seite mit freiem Platz geschoben und der Indexeintrag korrigiert. Ist ein Ausgleich nicht möglich, muss im schlimmsten Fall eine neue Datenseite angelegt und der ganze Index von dieser Position ab nach rechts verschoben werden.

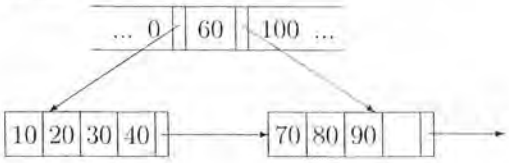
Abbildung 7.9 zeigt die drei möglichen Fälle (normales Einfügen, Ausgleich, Anlegen einer neuen Datenseite) beim Einfügen von Datensätzen an einem Beispiel.[2] Oben im Bild ist der initiale Zustand der Index- und Datenseiten angegeben.

Löschen eines Schlüssels. Schlüssel können solange aus einer Datenseite entfernt werden, bis sie leer ist. Eine leere Datenseite muss aus dem Index entfernt werden, wobei unter Umständen wieder der Index verschoben werden muss. Analog zum Einfügen kann auch zunächst ein Ausgleich mit einem Nachbarn versucht werden, wenn dieser gut gefüllt ist.

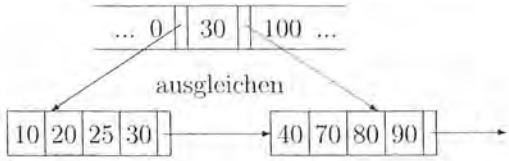
[2]Es ist unter Umständen sinnvoll, beim Anlegen einer neuen Datenseite die Daten zwischen übergelaufener und neuer Seite gleichmäßig zu verteilen, wie es später beim B-Baum besprochen wird.



a) Einfügen von 40



b) Einfügen von 25



c) Einfügen von 26

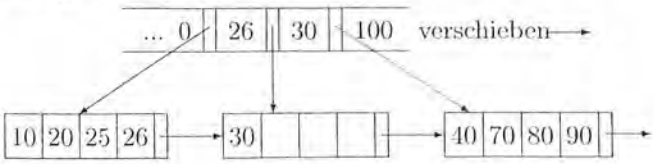


Abbildung 7.9: Einfügen in eine ISAM-Indexstruktur

Um das schlechte Verhalten der ISAM-Indexstruktur bei Update-Operationen zu verbessern, kann man eine weitere Indirektion einführen: Dann wird auch der Index, wie die Datenblöcke, als verkettete Liste verwaltet und ein Array von Zeigern auf die Index-Blöcke angelegt. Dadurch sind Verschiebungen seltener notwendig und nicht so gravierend. Mit der zweiten Indirektion bekommt die Indexstruktur Ähnlichkeit mit einem Baum. Man kann daher das ISAM-Verfahren als einen Vorgänger der im folgenden besprochenen B-Bäume ansehen.

7.8 B-Bäume

Normale Binärbäume wurden als Suchstruktur für den Hauptspeicher konzipiert. Sie eignen sich nicht als Speicherstruktur für den Hintergrundspeicher, da sie sich nicht effektiv auf Seiten abbilden lassen. Man verwendet daher für die Hintergrundspeicherung Mehrwegbäume, deren Knotengrößen auf die Seitenkapazitäten abgestimmt

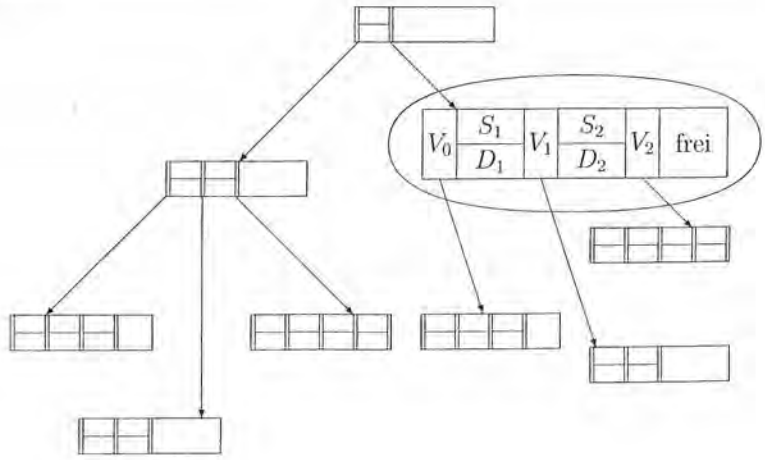


Abbildung 7.10: Aufbau eines B-Baums

werden. Ein Knoten des Baums entspricht einer Seite des Hintergrundspeichers.

B-Bäume und deren Varianten bieten sowohl für die Auslastung als auch für die Anzahl der Seitenzugriffe bei einer Suche feste Grenzen. Ein Seitenwechsel ist nur notwendig, wenn eine Kante verfolgt wird. Die maximale Anzahl der Seitenzugriffe während eines Suchvorgangs wird also durch die Höhe des Baums begrenzt. Abbildung 7.10 zeigt eine schematische Darstellung eines B-Baums. Aufgrund der Balancierung ist jeder Weg von der Wurzel zu einem Blatt im Baum gleich lang.

In dieser Darstellung nehmen wir vereinfachend an, dass eine Seite, entsprechend einem Knoten des Baums, maximal vier Einträge aufnehmen kann. In der Praxis ist das Fassungsvermögen von Seiten um Größenordnungen höher. Ein Eintrag besteht aus dem Schlüssel S_i und dem Datensatz D_i, der diesen Schlüssel enthält. Bei einem Sekundärindex werden nicht die Datensätze, sondern die TIDs der Datensätze (also Verweise) eingetragen. Zu jedem Eintrag S_i gibt es einen Verweis V_{i-1} auf Knoten, die kleinere Schlüsselwerte enthalten, und einen Verweis V_i entsprechend auf Knoten mit größeren Schlüsselwerten. Der in Abbildung 7.10 vergrößert dargestellte Knoten enthält zwei Einträge, die verbleibenden zwei Einträge sind frei.

Ein B-Baum mit Grad k ist also durch die folgenden Eigenschaften charakterisiert:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.
2. Jeder Knoten außer der Wurzel hat mindestens k und höchstens $2k$ Einträge. Die Wurzel hat zwischen einem und $2k$ Einträgen. Die Einträge werden in allen Knoten sortiert gehalten.
3. Alle Knoten mit n Einträgen, außer den Blättern, haben $n+1$ Kinder.
4. Seien $S_1, \ldots, S_n$ die Schlüssel eines Knotens mit $n+1$ Kindern. $V_0, V_1, \ldots, V_n$ seien die Verweise auf diese Kinder. Dann gilt:
   (a) V_0 weist auf den Teilbaum mit Schlüsseln kleiner als S_1.

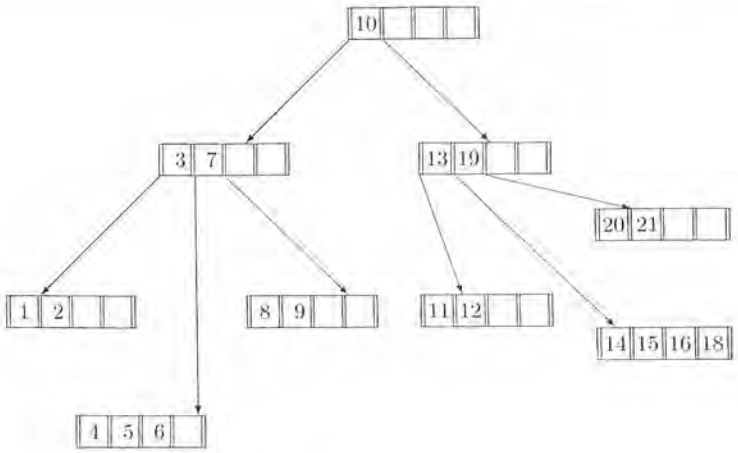


Abbildung 7.11: Ein Beispielbaum ($k = 2$)

(b) V_i ($i = 1, \ldots, n-1$) weist auf den Teilbaum, dessen Schlüssel zwischen S_i und S_{i+1} liegen.

(c) V_n weist auf den Teilbaum mit Schlüsseln größer als S_n.

(d) In den Blattknoten sind die Zeiger nicht definiert.

In der obigen Definition nehmen wir zur Vereinfachung die Eindeutigkeit des Schlüssels an (siehe dazu auch Aufgabe 7.6).

Um die geforderte Eigenschaft nach einer Mindestbelegung von k Einträgen pro Knoten einhalten zu können, müssen unter Umständen beim Löschen unterbelegte Knoten zusammengelegt werden. Ebenso muss, falls bei der maximalen Belegung von $2k$ Einträgen noch ein weiterer eingefügt werden soll, ein Knoten eventuell aufgeteilt werden. In manchen Fällen ist auch ein Ausgleich mit benachbarten Knoten möglich.

Einfügen von Schlüsseln. Das wollen wir anhand des vereinfachten Beispiels in Abbildung 7.11 demonstrieren. In dem dort abgebildeten B-Baum vom Grad 2 soll die Zahl 17 eingefügt werden - also der Datensatz mit dem Schlüssel 17, der hier allerdings nicht näher gezeigt wird. Es wird zunächst durch Absteigen im Baum die Einfügestelle gesucht, in diesem Fall zwischen der Zahl 16 und 18. In dem zugehörigen Knoten ist allerdings nicht mehr genügend Platz vorhanden; er muss aufgeteilt werden. Dazu wird der mittlere Eintrag, die Zahl 16, hochgeschoben in den Elternknoten. Die Zahlen, die vorher links und rechts von der 16 standen, bilden danach je einen separaten Knoten, wie in Abbildung 7.12 dargestellt. Diese beiden neuen Knoten erfüllen die geforderte Minimalbelegung. Unter Umständen kann sich ein Aufteilvorgang bis zur Wurzel fortsetzen. In dem Fall, dass auch die Wurzel vollständig belegt ist, muss ein neuer Wurzelknoten angelegt werden, und die ursprünglichen Einträge der Wurzel werden auf die zwei neuen Kinder aufgeteilt. Der Baum wächst so um eine Stufe in die Höhe. Abbildung 7.13 beschreibt den Einfügenvorgang noch einmal in algorithmischer Form.

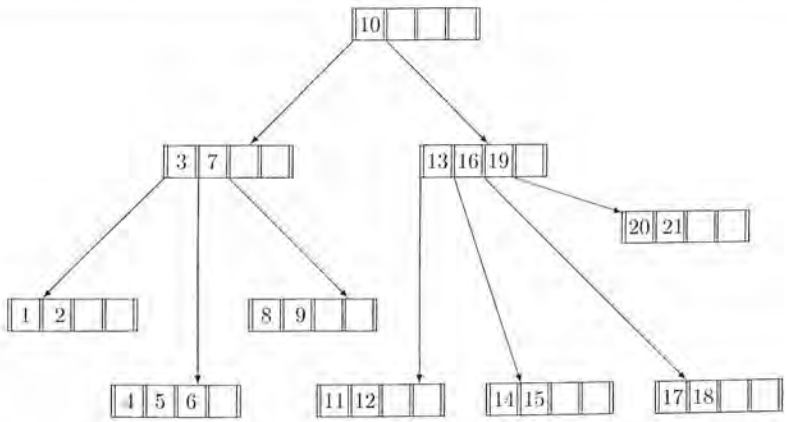


Abbildung 7.12: Einfügen einer 17

1. Führe eine Suche nach dem Schlüssel durch; diese endet (scheitert) an der Einfügestelle.
2. Füge den Schlüssel dort ein.
3. Ist der Knoten überfüllt, teile ihn:
   - Erzeuge einen neuen Knoten und belege ihn mit den Einträgen des überfüllten Knotens, deren Schlüssel größer ist als der des mittleren Eintrags.
   - Füge den mittleren Eintrag im Vaterknoten des überfüllten Knotens ein.
   - Verbinde den Verweis rechts des neuen Eintrags im Vaterknoten mit dem neuen Knoten.
4. Ist der Vaterknoten jetzt überfüllt?
   - Handelt es sich um die Wurzel, so lege eine neue Wurzel an.
   - Wiederhole Schritt 3 mit dem Vaterknoten.

Abbildung 7.13: Algorithmus zum Einfügen in einen B-Baum

Löschen eines Schlüssels. Die Vorgehensweise beim Löschen hängt davon ab, ob ein Eintrag aus einem Blattknoten oder aus einem inneren Knoten entfernt werden soll. In einem Blattknoten kann ein Eintrag einfach gelöscht werden. In einem inneren Knoten muss die Verbindung zu den Kindern des Knotens bestehen bleiben, daher wird der nächstgrößere (oder nächstkleinere) Schlüssel gesucht und an die Stelle des alten Schlüssels plaziert. In beiden Fällen kann ein Blattknoten unterbelegt zurückbleiben – im zweiten Fall ist es der ursprüngliche Aufenthaltsort des nächstgrößeren (-kleineren) Schlüssels. Damit der Baum nicht gegen die Bedingung 2 der Definition verstößt, wird der Knoten mit einem seiner Nachbarn entweder ausgeglichen oder verschmolzen. Ein Ausgleich bewirkt die gleichmäßige Verteilung der Inhalte der beiden Knoten. Ein Verschmelzen ist nur möglich, wenn beide Knoten minimal belegt sind. Dann tritt an deren Stelle ein Knoten, der zusätzlich zu deren Inhalt noch den zugehörigen Schlüssel aus dem Vaterknoten enthält. Das kann wiederum zur Unterbelegung des Vaterknotens führen und den Verschmelzungs- bzw. Ausgleichsvorgang nach oben fortsetzen.

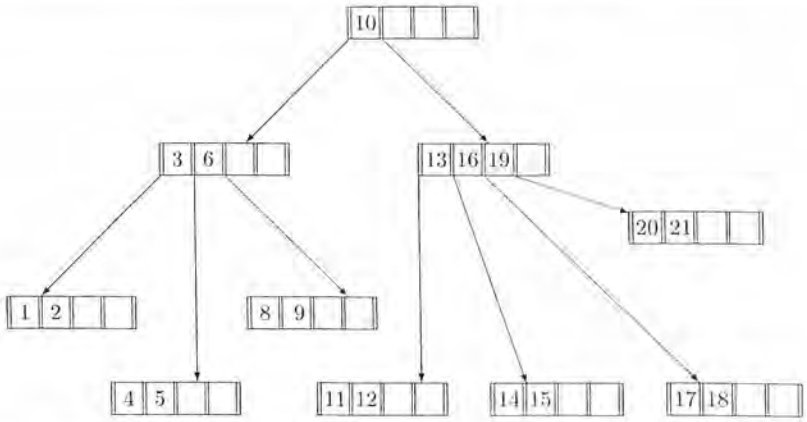


Abbildung 7.14: Löschen der 7

Abbildung 7.14 zeigt den B-Baum, nachdem die 7 gelöscht wurde. Als Ausgleich sollte die 8 an die Stelle der 7 geschoben werden, das hätte zu einer Unterbelegung geführt. Ein Ausgleich mit dem Nachbarknoten führt dazu, dass die 6 den Platz im Vaterknoten einnimmt. Ein weiterer Löschversuch in diesem Teil des Baums, z.B. der 5, zieht kompliziertere Aktionen nach sich: Jetzt ist eine Verschmelzung zweier Blattknoten notwendig. Zusätzlich ergibt sich daraus eine Unterbelegung des Vaterknotens, der durch einen Ausgleich mit der rechten Baumhälfte beseitigt werden müsste. Erfahrungen mit realen Datenbanken zeigen jedoch, dass Löschoperationen im Verhältnis zu Einfügeoperationen selten sind. Daher wird in Implementierungen von B-Bäumen häufig sogar auf Verschmelzungen ganz verzichtet – wodurch natürlich die in Bedingung 2 geforderte Minimalbelegung verletzt werden kann.

Es soll noch einmal betont werden, dass die vorgeführten Größenordnungen nicht realistisch sind. Reale B-Bäume haben Verzweigungsgrade in der Größenordnung von 100 – abhängig natürlich von der Größe der Datensätze und dem Fassungsvermögen der Seiten. Deshalb reichen z.B. etwa vier Seitenzugriffe (entsprechend der Höhe des B-Baums) um einen Datensatz unter 10^7 Einträgen zu finden.

7.9 B^+-Bäume

Dadurch, dass jeder Knoten eine Seite des Hintergrundspeichers belegt, hängt die Höhe eines B-Bauns direkt mit der Anzahl der Seitenzugriffe zum Auffinden eines Datums zusammen. Bei B-Bäumen ist also ein hoher Verzweigungsgrad wünschenswert, denn je weiter ein Baum verzweigt ist, desto flacher ist er. Der Verzweigungsgrad bei B-Bäumen hängt von der Satzgröße ab, wenn die Datensätze innerhalb der Knoten gespeichert werden. Bei B^+-Bäumen[3] wird die Höhe dadurch reduziert, dass

[3]Die Terminologie ist hier nicht ganz klar, vielfach wird auch der Name B*-Baum benutzt. Der von Knuth (1973) ursprünglich definierte B*-Baum ist eine Variante des B-Baums, bei dem eine Mindestbelegung der Knoten von 2/3 durch Umverteilungen garantiert wird. Der in diesem Abschnitt vorgestellte Baum wurde von Knuth nicht mit Namen versehen. Wir folgen einem Vorschlag

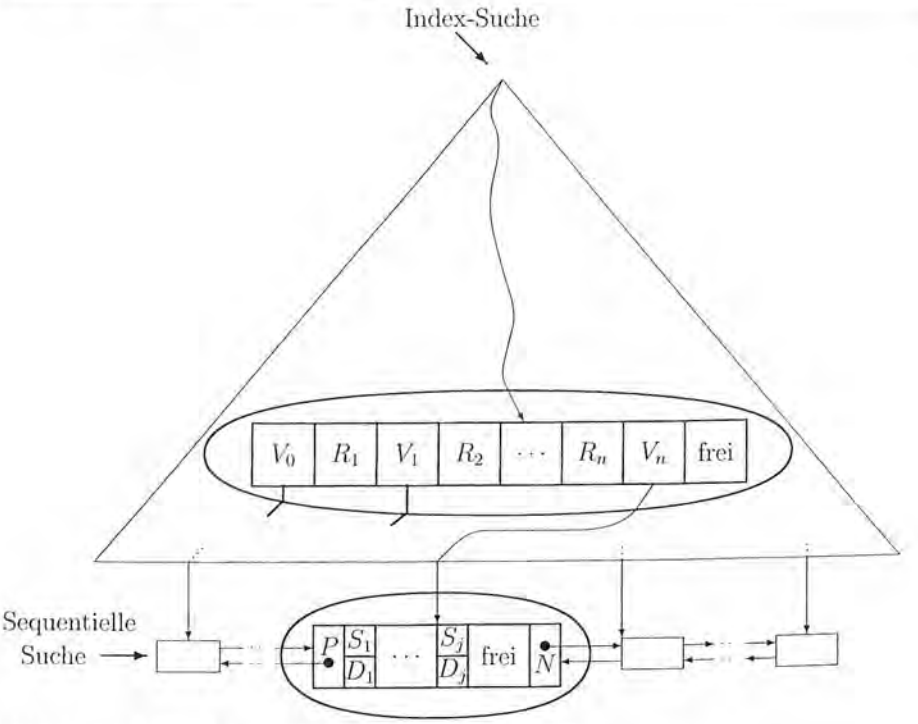


Abbildung 7.15: Schematischer Aufbau eines B⁺-Baums

Daten nur noch in den Blättern gespeichert werden. Daher spricht man auch von einem *hohlen* Baum. Die inneren Knoten enthalten lediglich Referenzschlüssel R_i als Wegweiser („road map"). Eine Suche nach einem Datensatz D_i muss deshalb immer komplett bis zu den Blättern durchgeführt werden. Der schematische Aufbau eines B⁺-Baums ist in Abbildung 7.15 dargestellt.

Um zusätzlich eine effiziente sequentielle Verarbeitung der Datensätze zu ermöglichen, sind die Blattknoten jeweils mit einem Zeiger auf den vorhergehenden (P) und nachfolgenden Blattknoten (N) in der gewünschten Suchreihenfolge verbunden.

Ein B⁺-Baum vom Typ (k, k^*) hat also folgende Eigenschaften:

1. Jeder Weg von der Wurzel zu einem Blatt hat die gleiche Länge.

2. Jeder Knoten – außer Wurzeln und Blättern – hat mindestens k und höchstens $2k$ Einträge. Blätter haben mindestens k^* und höchstens $2k^*$ Einträge. Die Wurzel hat entweder maximal $2k$ Einträge, oder sie ist ein Blatt mit maximal $2k^*$ Einträgen.

3. Jeder Knoten mit n Einträgen, außer den Blättern, hat $n + 1$ Kinder.

4. Seien $R_1, \ldots, R_n$ die Referenzschlüssel eines inneren Knotens (d.h. auch der Wurzel) mit $n + 1$ Kindern. Seien $V_0, V_1, \ldots, V_n$ die Verweise auf diese Kinder.

von Comer (1979) und nennen ihn B⁺-Baum.

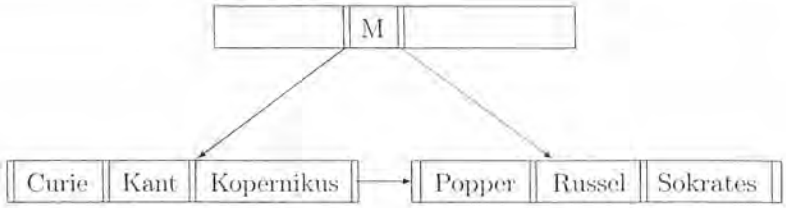


Abbildung 7.16: Schematische Darstellung zum Präfix-B$^+$-Baum

(a) V_0 verweist auf den Teilbaum mit Schlüsseln kleiner oder gleich R_1.

(b) V_i $(i = 1, \ldots, n-1)$ verweist auf den Teilbaum, dessen Schlüssel zwischen R_i und R_{i+1} liegen (einschließlich R_{i+1}).

(c) V_n verweist auf den Teilbaum mit Schlüsseln größer als R_n.

Ein zusätzlicher Vorteil des B$^+$-Baums ist die effizientere Wartung durch die Verwendung von Referenzschlüsseln. Referenzschlüssel müssen nicht unbedingt einem realen Schlüssel entsprechen. Daher brauchen Referenzschlüssel nur gelöscht zu werden, falls Blattknoten zusammengelegt werden und eventuell bei den sich daraus ergebenden weiteren Verschmelzungen. Beim Aufteilen von Blattknoten wird der mittlere Schlüssel nicht in den Vaterknoten verschoben, sondern wandert z.B. in die linke Hälfte. Im Vaterknoten wird eine Kopie (oder ein anderer Referenzschlüssel, der die Datensätze der beiden Blätter differenziert – siehe unten) eingetragen.

7.10 Präfix-B$^+$-Bäume

Eine zusätzliche Verbesserungsmöglichkeit bei B$^+$-Bäumen ist der Einsatz von Schlüsselpräfixen anstelle von kompletten Schlüsseln. Werden z.B. längere Zeichenketten als Schlüssel verwendet, wird der Verzweigungsgrad des B$^+$-Baums klein. Da B$^+$-Bäume nur Referenzschlüssel enthalten, braucht nur irgendein Schlüssel gefunden zu werden, der die Teilbäume zur Linken und zur Rechten trennt. Die Situation wird schematisch in Abbildung 7.16 verdeutlicht.

Normalerweise wäre „Kopernikus" der eingetragene Referenzschlüssel, anhand dessen die Verzweigungsentscheidung getroffen wird. Platzsparender ist es jedoch, irgendeinen anderen kürzestmöglichen Schlüssel R einzutragen, der die Eigenschaft

$$\text{Kopernikus} \leq R < \text{Popper}$$

erfüllt, z.B. ein „M". Bei dicht beieinanderliegenden Schlüsseln kann das Verfahren jedoch versagen, z.B. wenn ein R gesucht wird mit

$$\text{Systemprogramm} \leq R < \text{Systemprogrammierer}$$

7.11 Hintergrundspeicher-Struktur der B-Bäume

Wir werden hier vereinfachend die Struktur der B-Bäume diskutieren. Der Entwurf für die heterogene Knotenstruktur der B$^+$-Bäume sei den Lesern als Übung über-

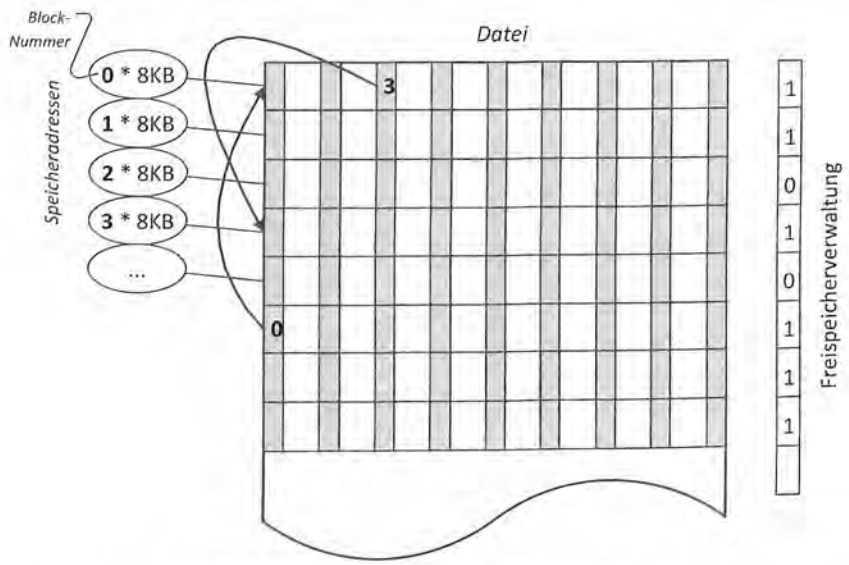


Abbildung 7.17: Hintergrundspeicher-Struktur eines B-Baums

lassen. Die B-Bäume sind als Indexstruktur für den Hintergrundspeicher konzipiert. Deshalb entsprechen die Knoten eines B-Baums den Seiten, die zwischen dem Hintergrundspeicher und dem im Hauptspeicher angesiedelten Systempuffer hin- und her-transferiert werden. Eine typische Größe dieser Seiten liegt im Bereich von 8 KB (KiloByte). Der Verzweigungsgrad (in der vorhergehenden Diskussion k bzw. $2k$ genannt) errechnet sich dann aus der Größe der Zeiger auf die nachfolgenden Knoten (typischerweise 4 Byte) sowie der Größe der indexierten Suchschlüssel und der Größe der Zeiger auf die Tupel (TIDs), die mindestens 8 Byte groß sind. Wir überlassen es den Lesern, den Verzweigungsgrad für übliche Suchschlüssel (int, long, char(20), etc.) zu berechnen. Die Speicherstruktur dieser Knoten/Seiten innerhalb einer Datei ist in Abbildung 7.17 gezeigt.

Die Datei wird also in Blöcke/Seiten der jeweiligen Größe 8 KB aufgeteilt. Diese Blöcke entsprechen den Knoten im B-Baum. Die grauen Felder stellen Verweise/-Zeiger auf Kind-Knoten dar, wohingegen die weißen Felder den Suchschlüssel sowie den TID auf den Datensatz mit diesem Suchschlüssel enthalten. Da Knoten/Seiten des B-Baums auch „leer laufen" können ist eine Freispeicherverwaltung in der Form eines Bitvektors nötig. Ein „0"-Eintrag kennzeichnet einen freien Block, der beim nächsten Überlauf eines Knotens herangezogen werden kann. Die Verweise auf Kindknoten werden also als Blocknummern abgelegt, in unserer Abbildung sind nur die beiden Verweise auf die Blöcke 3 und 0 gezeigt. Deren Anfangsposition in der Datei berechnet sich demnach als Vielfache der Knoten/Block-Größe – wie in den Kommentarblasen an der linken Seite angedeutet. Zusätzlich zu den gezeigten Strukturen muss auch noch die Speicherposition des Wurzelknotens bekannt sein, damit man den Baum überhaupt navigieren kann. Dieser befindet sich nämlich nicht an der Position 0, da ja der B-Baum beim Höhenwachstum immer wieder eine neue Wurzel bekommt, deren Position man konsistent und dauerhaft auf dem Hintergrundspei-

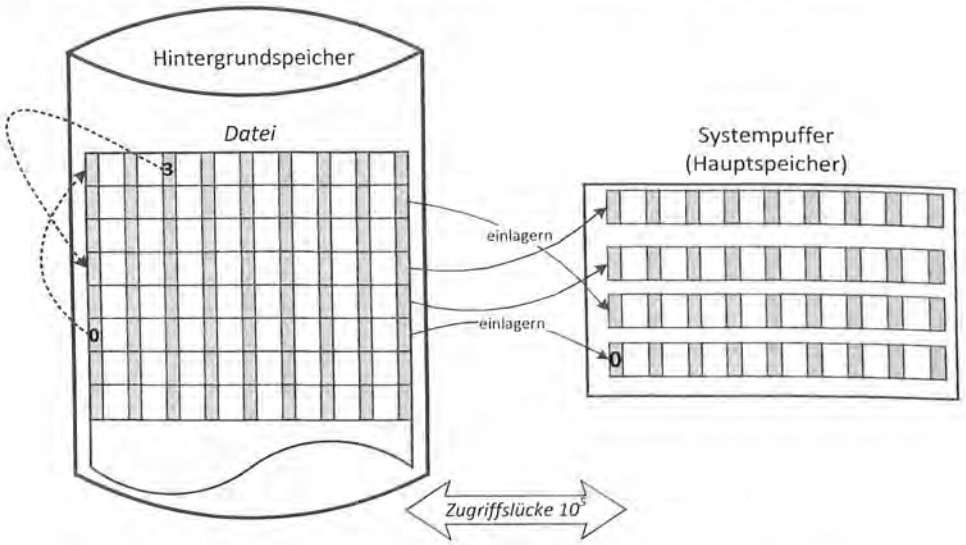


Abbildung 7.18: Zusammenspiel des Hintergrundspeichers und des Systempuffers bei B-Bäumen

cher fortschreiben muss.

In Abbildung 7.18 ist das Zusammenspiel zwischen Hintergrundspeicher und Systempuffer illustriert. Ein (mehr oder weniger) großer Anteil der Knoten des Indexes befindet sich im Systempuffer. Als Daumenregel sollte gelten, dass fast alle inneren Knoten des B-Baums im Systempuffer Platz finden und Zugriffe auf den Hintergrundspeicher allenfalls für die Einlagerung von Blattknoten zu zahlen sind. Nur so kann das Datenbanksystem leistungsfähig arbeiten, da der Zugriff auf einen Knoten des Hintergrundspeichers einen Faktor 10^5 mal länger dauert als der Zugriff auf einen im Systempuffer angesiedelten Knoten.

Wenn man also von einem Knoten zu einem anderen Knoten traversiert, wird zunächst in einer entsprechenden *Seitentabelle* ermittelt, ob sich dieser Knoten (also die Seite) im Systempuffer befindet und, wenn ja, an welcher Position. Falls der Knoten nicht im Systempuffer vorhanden ist, muss er eingelagert werden, wozu erst noch ein anderer Knoten verdrängt werden muss. Der Zugriff auf einen noch nicht im Systempuffer befindlichen Knoten verursacht einen sogenannten *Seitenfehler*, der wegen der Latenz des Zugriffs auf den Hintergrundspeicher mit erheblichen Verzögerungen verbunden ist. Deshalb ist es notwendig, hinreichend viel Platz für den Systempuffer zur Verfügung zu stellen; also die Datenbankserver mit entsprechend hohen Hauptspeicherkapazitäten zu konfigurieren.

7.12 Hashing

Das ultimative Ziel aller Bemühungen um ein gutes physisches Design ist es, wirklich nur diejenigen Seiten vom Hintergrundspeicher zu lesen, die absolut benötigt werden.

0	
1	(27550, 'Schopenhauer', 6)
2	(24002, 'Xenokrates', 18)
	(25403, 'Jonas', 12)

Abbildung 7.19: Eine aus drei Seiten bestehende Hash-Tabelle

Hash-Verfahren ermöglichen es, ein bestimmtes Datum im Durchschnitt mit einem bis zwei Seitenzugriffen zu finden. Bäume benötigen Seitenzugriffe in der Ordnung von $\log_k(n)$, wobei k der durchschnittliche Verzweigungsgrad und n die Anzahl der eingetragenen Datensätze ist.[4]

Beim Hashing wird mit Hilfe einer sogenannten *Hashfunktion* der Schlüssel auf einen Behälter (engl. *bucket*) abgebildet, der das dem Schlüssel zugehörige Datum aufnehmen soll. Im Allgemeinen ist nicht für den gesamten Wertebereich des Schlüssels Platz im Speicher vorhanden. Es kann daher vorkommen, dass mehrere Datensätze an die gleiche Stelle gespeichert werden sollen. In diesem Fall wird entweder eine hier nicht weiter ausgeführte Kollisionsbehandlung eingeschaltet oder das sogenannte *offene Hashing* verwendet, das weiter unten erläutert wird.

Formaler ausgedrückt ist also eine Hashfunktion (oder auch *Schlüsseltransformation*) h eine Abbildung

$$h : S \to B,$$

wobei S eine beliebig große Schlüsselmenge und B eine Nummerierung der n Behälter, also ein Intervall $[0..n)$ ist. Normalerweise ist die Anzahl der möglichen Elemente in der Schlüsselmenge sehr viel größer als die Anzahl der zur Verfügung stehenden Behälter ($|S| \gg |B|$), daher kann h i.A. nicht injektiv sein. Es sollte aber die Elemente von S gleichmäßig auf B verteilen, da eine Kollisionsbehandlung bzw. der Überlauf eines Behälters zusätzlichen Aufwand verursacht. Gilt für zwei Schlüssel S_1 und S_2, dass $h(S_1) = h(S_2)$ ist, nennt man S_1 und S_2 *synonym*.

Nehmen wir an, die Daten der Studenten werden häufig anhand ihrer Matrikelnummer gesucht. Deshalb sollen sie in eine Hash-Tabelle eingetragen werden, für die drei Seiten reserviert sind, die jeweils zwei Einträge aufnehmen können. Häufig wird als Hashfunktion der Schlüsselwert modulo der Tabellengröße verwendet. Für einen aus drei Seiten bestehenden Speicherbereich könnte also die folgende Hashfunktion verwendet werden:

$$h(x) = x \bmod 3$$

Dieses *Divisionsrestverfahren* ist die gebräuchlichste Art einer Hashfunktion. Es hat sich gezeigt, dass man am günstigsten eine Primzahl für die Berechnung des Divisionsrestes wählt, um eine gute Streuung zu gewährleisten.

Abbildung 7.19 zeigt die Hash-Tabelle nachdem Xenokrates ($h(24002) = 2$), Jonas ($h(25403) = 2$) und Schopenhauer ($h(27550) = 1$) eingetragen wurden. Die durchgezogenen Linien deuten Seitengrenzen an.

[4] Diese Zahlen werden meist durch Pufferungseffekte relativiert.

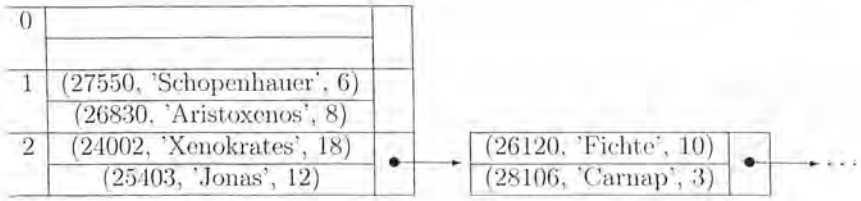


Abbildung 7.20: Kollisionsbehandlung durch Überlaufbehälter

Versucht man, in diese Tabelle noch Fichte ($h(26120) = 2$) einzutragen, tritt ein Überlauf auf, da Seite 2 bereits durch Xenokrates und Jonas belegt ist. Beim offenen Hashing wird nun in der Seite ein Zeiger auf einen weiteren Behälter gespeichert. Dieser weitere Behälter ist ein Überlaufbereich fester Größe (in unserem Fall eine Seite), der die zusätzlichen Kandidaten für den zugehörigen Speicherplatz enthält. Ein Überlaufbehälter kann wiederum überlaufen und einen Verweis auf weitere Behälter enthalten (siehe Abbildung 7.20). Man sieht schon, dass unsere Hashfunktion $h(x)$ nicht gut gewählt wurde. Es gibt zu viele Matrikelnummern, die auf den Platz 2 abgebildet werden.

Das gerade vorgestellte Hash-Verfahren ist für eine reale Datenbasis zu statisch. Da eine einmal angelegte Hash-Tabelle nicht effizient vergrößert werden kann, gibt es nur zwei wenig wünschenswerte Alternativen, wenn viele Einfügeoperationen erwartet werden: Entweder es wird von vornherein sehr viel Platz für die Tabelle reserviert, so dass der Platz zunächst verschwendet ist, oder es entstehen im Laufe der Zeit immer längere Überlaufketten. Diese Überlaufketten können nur durch Änderung der Hashfunktion und aufwendige Reorganisation der Tabelle beseitigt werden.

7.13 Erweiterbares Hashing

Eine Verbesserung bietet das *erweiterbare Hashing*. Dazu wird die Hashfunktion h so modifiziert, dass sie nicht mehr unbedingt auf einen Index eines tatsächlich vorhandenen Behälters abbildet, sondern auf einen wesentlich größeren Bereich. Das Ergebnis einer Berechnung von $h(x)$ wird binär dargestellt und nur ein Präfix dieser binären Darstellung berücksichtigt, der dann auf den tatsächlich verwendeten Behälter verweist.

Abbildung 7.21 zeigt eine schematische Darstellung des erweiterbaren Hashings. Die binäre Darstellung des Ergebnisses der Hashfunktion wird in zwei Teile aufgeteilt: $h(x) = dp$. d gibt die Position des Behälters im *Verzeichnis* an. Das Verzeichnis (engl. *directory*) fasst in der gezeigten Konstellation 2^2 Einträge, also werden zwei Bits für d gebraucht. Die Größe von d wird die *globale Tiefe* t genannt. p ist der zur Zeit nicht benutzte Teil des Schlüssels.

Man kann sich das Verzeichnis des erweiterbaren Hashings konzeptuell auch als binären Entscheidungsbaum „entlang" des Hashcodes vorstellen. Dies ist auf der linken Seite in Abbildung 7.22 gezeigt. Man traversiert für einen gegebenen Hashcode $h(x)$ Bit-für-Bit so lange nach unten, bis man den Zeiger zum Bucket findet (ge-

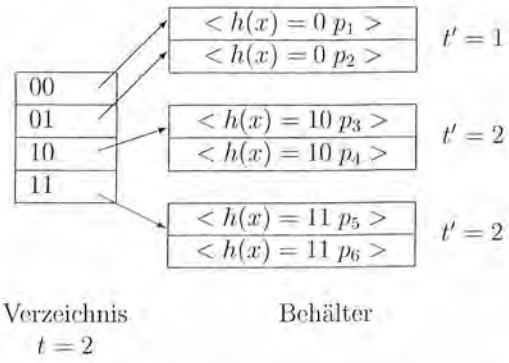


Abbildung 7.21: Schematische Darstellung des erweiterbaren Hashings

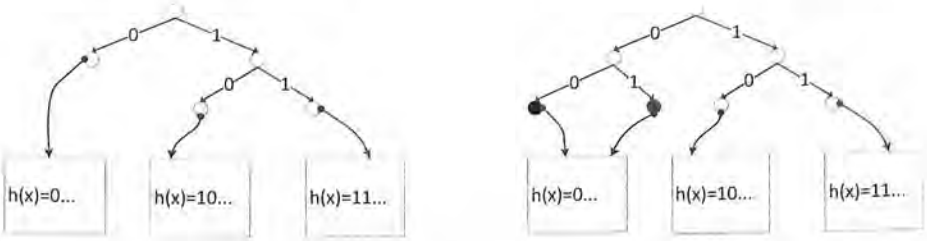


Abbildung 7.22: Das Verzeichnis als Entscheidungsbaum (links) sowie als vollständiger Entscheidungsbaum mit „Dummy Knoten“ (rechts)

nauer gesagt, die Seitennummer der Bucket-Seite). In dieser Bucket-Seite ist der Datensatz, falls er überhaupt existiert, abgelegt. Beim Überlauf des Buckets muss man (mindestens) eine weitere Entscheidungsstufe einbauen, wie es für die Präfixe 10 und 11 im Vergleich zum Präfix 0 in dem Entscheidungsbaum gezeigt ist.

Auf der rechten Seite der Abbildung ist gezeigt, wie man durch das Einfügen von „Dummy-Knoten“ den unbalancierten Entscheidungsbaum in einen vollständigen, balancierten Entscheidungsbaum umformen kann. Diese Umformung bildet die Basis für die Hintergrundspeicher-optimierte Array-Darstellung des Entscheidungsbaums, die in Abbildung 7.23 gezeigt ist. Der Entscheidungsbaum eignet sich selbst wegen seiner großen Höhe nicht als Verzeichnisstruktur, die man auf dem Hintergrundspeicher effizient verwalten könnte. Das für den Hintergrundspeicher optimierte Verzeichnis-Array erhält man, indem man jeden Pfad in dem balancierten Entscheidungsbaum als Index in ein Zeiger-Array auffasst. Man beginnt also beim „linkesten“ Pfad 000 und hört beim „rechtesten“ Pfad 111 mit der Konvertierung auf. In der Algorithmik des erweiterbaren Hashings spielt der Entscheidungsbaum jetzt auch keine Rolle mehr – er dient de facto nur als Gedankenmodell.

Wenn auf der rechten Seite, also im schlimmstmöglichen Fall (warum?), das Bucket überläuft, muss man den Entscheidungsbaum und damit das Verzeichnis um eine Stufe erweitern. Dies ist in Abbildung 7.24 illustriert. Die globale Tiefe

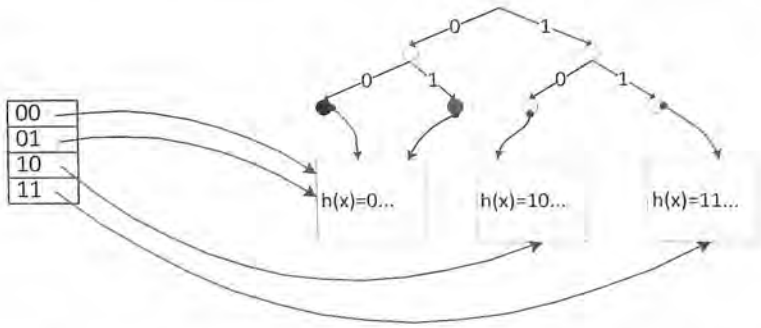


Abbildung 7.23: Der Entscheidungsbaum wird in ein Verzeichnis-Array transformiert, indem jeder Pfad materialisiert wird.

des Verzeichnisses (also des gedanklichen Entscheidungsbaums) beträgt jetzt 3. Die *lokale Tiefe* t' eines Behälters gibt an, wieviele Bits des Schlüssels für diesen Behälter tatsächlich verwendet werden. Eine Verdoppelung des Verzeichnisses erfolgt also, wenn nach einer Aufteilung eines Behälters die lokale Tiefe größer als die globale Tiefe ist. Jetzt haben die beiden rechten Buckets die lokale Tiefe $t' = 3$, die mit der globalen Tiefe $t = 3$ übereinstimmt. Das Bucket ganz links hat die lokale Tiefe $t' = 1$ und das zweitlinke Bucket die lokale Tiefe $t' = 2$.

Müsste ein neuer Datensatz in einen bereits vollen Behälter eingetragen werden, würde er aufgeteilt werden. Die Aufteilung erfolgt anhand eines weiteren Bits des bisher unbenutzten Teils p. Ist die globale Tiefe nicht ausreichend, um den Verweis auf den neuen Behälter eintragen zu können, muss das Verzeichnis verdoppelt werden. Es wäre – insbesondere bei Anwendung der Hashfunktion auf einen Nichtschlüssel – denkbar, dass mehr Datensätze auf denselben (vollständigen) Hashwert abgebildet werden, als in einem Behälter Platz haben. In diesem Fall muss man das erweiterbare Hashing mit einer Überlauftechnik wie in Abbildung 7.20 kombinieren.

Abbildung 7.25 zeigt einen Hash-Index auf dem Attribut *PersNr* der Relation *Professoren*, in dem schon Sokrates, Russel und Kopernikus eingetragen sind. Als Hashfunktion wird die umgedrehte binäre Darstellung der Personalnummer verwendet. In realistischen Anwendungen sollte jedoch zur besseren Streuung noch z.B. das Divisionsrestverfahren vorgeschaltet werden. Zur Orientierung ist oberhalb des Indexes eine Tabelle mit den Hashwerten der Personalnummern angegeben. Nun soll Descartes eingefügt werden.

Descartes hat die Personalnummer 2129 und fällt in den bereits von Sokrates und Kopernikus vollständig belegten Behälter. Die globale Tiefe stimmt mit der lokalen Tiefe dieses Behälters überein, also muss das Verzeichnis verdoppelt werden (Abbildung 7.26). Durch die Vergrößerung des maßgebenden Teils des Hash-Wertes kann Descartes jetzt eingeordnet werden.

Ist durch die Hinzunahme eines neuen Bits zum relevanten Teil des Hash-Wertes immer noch keine Aufteilung des angestrebten Behälters möglich, muss das Verzeichnis nochmals verdoppelt werden.

Werden Daten gelöscht, ist es unter Umständen möglich, Behälter wieder zu

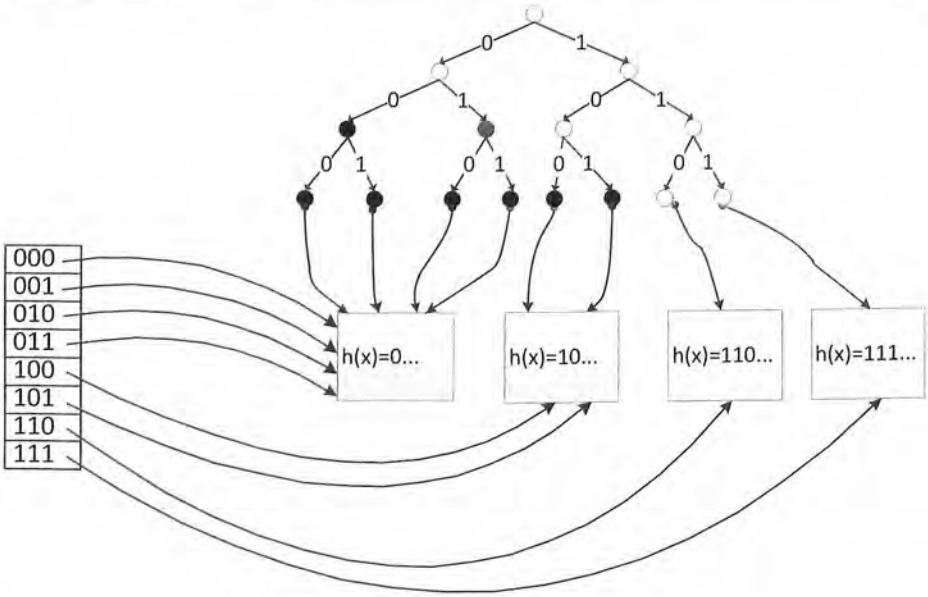


Abbildung 7.24: Ein Entscheidungsbaum der Höhe 3 führt zu einem Verzeichnis mit $2^3 = 8$ Einträgen

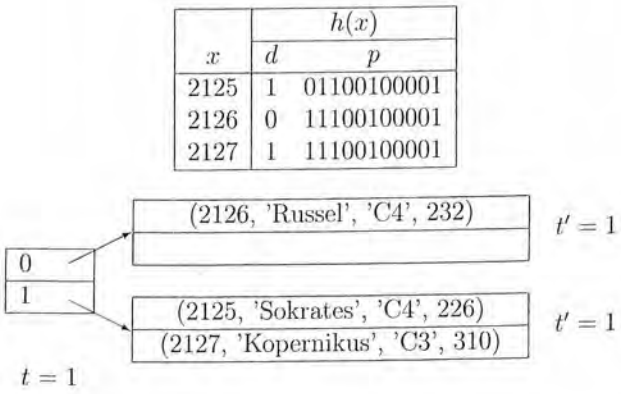


Abbildung 7.25: Ein Hash-Index

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

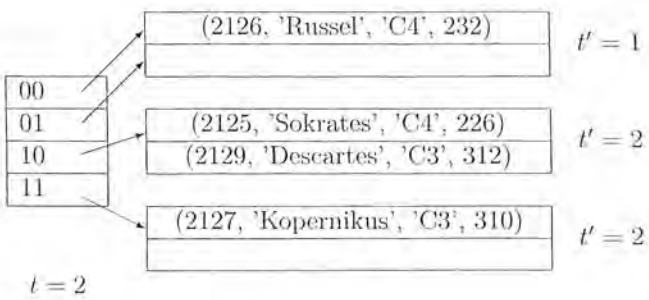


Abbildung 7.26: Einfügen von (2129, Descartes, C3, 312)

verschmelzen oder gar das Verzeichnis zu halbieren. Eine Verschmelzung ist immer dann möglich, wenn sich der Inhalt zweier benachbarter Behälter zusammen in einem unterbringen lässt. „Benachbart" sind Behälter, wenn sie die gleiche lokale Tiefe haben und der Wert der ersten $t' - 1$ Bits des Hash-Wertes (von links) übereinstimmt. In Abbildung 7.26 sind die unteren beiden Behälter benachbart. Sie haben beide die lokale Tiefe $t' = 2$, der d-Anteil des Hash-Wertes ist binär 10 und 11. Hätten sie insgesamt zwei Einträge, könnten sie wieder zusammengelegt werden. Ihre lokale Tiefe würde sich um eins erniedrigen. Der obere Behälter hat keinen Nachbarn.

Das Verzeichnis kann immer dann halbiert werden, wenn alle lokalen Tiefen echt kleiner sind als die globale Tiefe t. Durch das Halbieren erniedrigt sich die globale Tiefe um eins.

7.14 Mehrdimensionale Indexstrukturen

In vielen Anfragen hat man Selektionsprädikate, die sich auf mehrere Attribute einer Relation beziehen. Als Beispiel betrachte man die Anfrage nach den gut verdienenden jungen Angestellten, also z.B. solchen, deren *Alter* zwischen 22 und 25 ist und deren *Gehalt* zwischen 80K und 120K – also gerade diplomierte Datenbankexperten. Hätte man B^+-Bäume für beide Attribute *Alter* und *Gehalt*, so gäbe es mehrere mögliche Auswertungsstrategien. Eine mögliche Strategie besteht darin, die sich gemäß Alter qualifizierenden Tupel (genauer deren TIDs) zu suchen, danach die TIDs der sich nach Gehalt qualifizierenden TIDs zu ermitteln, dann den Durchschnitt dieser beiden Mengen berechnen und schließlich die entsprechenden Daten gemäß den TIDs „von der Platte zu holen".

Die mehrdimensionalen Indexstrukturen haben das Ziel, solche Operationen wesentlich effizienter zu gestalten, indem bei der Indexerstellung gleich mehrere Dimen-

sionen (Attribute) berücksichtigt werden. Wir wollen hier nur den *R*-Baum, sozusagen den „Urvater" der baumstrukturierten mehrdimensionalen Indices, vorstellen. In Abbildung 7.27 sind drei Phasen in der Entstehungsgeschichte eines *R*-Baums gezeigt. Schauen wir uns die erste Phase an, die oben links in der Abbildung dargestellt ist. Man unterscheidet innere Knoten (eckig dargestellt) und abgerundet dargestellte Blattknoten, die die eigentlichen Datensätze bzw. die Verweise (TIDs) darauf enthalten. Ein Eintrag im inneren Knoten besteht aus zwei Teilen: einer *n*-dimensionalen *Region* – salopp Box genannt – und einem Verweis auf einen Nachfolger (innerer Knoten oder Blatt). Die *n*-dimensionale Box ist die minimale Box, die alle Boxen oder Datenpunkte des Nachfolgerknotens begrenzt. Nachfolgend wollen wir uns auf zwei Dimensionen beschränken, sollten aber „im Hinterkopf" behalten, dass der *R*-Baum auch auf mehr Dimensionen anwendbar ist.

Der zweidimensionale Datenraum unseres Beispiels ist rechts in Abbildung 7.27 gezeigt – dieser dient natürlich nur der Intuition; gespeichert ist nur der linke Teil der Abbildung. Wir haben anfangs vier Datensätze, entsprechend vier Punkten im Datenraum. Die begrenzende zweidimensionale Box hat also die Ausdehnung [18,60] bezüglich der *Alter*-Dimension und [60,120] bezüglich der Dimension *Gehalt*.

Beim Einfügen des Datensatzes *Speedy* gibt es einen Überlauf des Blatts, da wir eine Kapazität von vier annehmen. Es muss also ein Ausgleich durchgeführt werden. Nun gibt es natürlich viele unterschiedliche Möglichkeiten, die fünf Datenelemente auf zwei Knoten aufzuteilen – es gibt ja jetzt keine totale Ordnung wie beim B^+-Baum, wo die Aufteilung deterministisch „in der Mitte" vollzogen wurde. Beim *R*-Baum gilt allgemein, dass man bei der Aufteilung so vorgehen soll, dass die resultierenden Boxen klein sind und sich sehr wenig (wenn überhaupt) überlappen. Wir können aber nicht hoffen, immer die optimale Aufteilung zu finden, da man dazu alle Möglichkeiten der Aufteilung durchprobieren müsste. Dies ist bei realistischen Knotenkapazitäten von, sagen wir, 100 nicht mehr möglich. Deshalb muss man entsprechende Heuristiken anwenden. In Abbildung 7.28 sind zwei mögliche Partitionierungen der fünf Datenelemente gezeigt. Die rechte Aufteilung ist deutlich schlechter, da sie viel größere Boxen als Ergebnis hat. Intuitiv kann man sich das so vorstellen, dass größere umrandende Boxen die Präzision des Indexes degradieren und Anfragen (siehe unten) deshalb mehr Aufwand verursachen würden.

Die bessere der beiden Aufteilungen ist in der Mitte der Abbildung 7.27 angewendet worden: *Speedy* und *Bond* teilen sich einen Knoten/eine Box und *Duck*, *Mini*, und *Mickey* kommen in die andere Box. Die Wurzel des *R*-Baums hat jetzt zwei Einträge, gemäß den beiden Boxen im rechts dargestellten Datenraum.

Beim Einfügen eines neuen Datensatzes geht man wie folgt vor. Man traversiert rekursiv von der Wurzel beginnend aus nach unten. Es kann an jedem inneren Knoten mehrere Möglichkeiten geben: (1) der Datensatz „fällt" in eine Box, (2) er fällt in mehrere Boxen (weil sich Boxen überlappen können) oder (3) er fällt in keine Box. Im ersten Fall nehmen wir den zugehörigen Weg der Box nach unten. Im zweiten Fall kann man irgendeine Box auswählen und weiter nach unten traversieren. Im dritten Fall wählt man die Box, die am wenigsten vergrößert werden muss, um den Datensatz aufzunehmen. Wenn wir beispielsweise den neuen Datensatz *Bert* mit *Alter=45* und *Gehalt=55K* einfügen wollen (siehe Abbildung 7.27 Mitte rechts), so sollte er sicherlich in die linke Box eingefügt werden, da diese viel weniger vergrößert werden muss als die rechte. Unten in der Abbildung 7.27 ist der Zustand des *R*-Baums

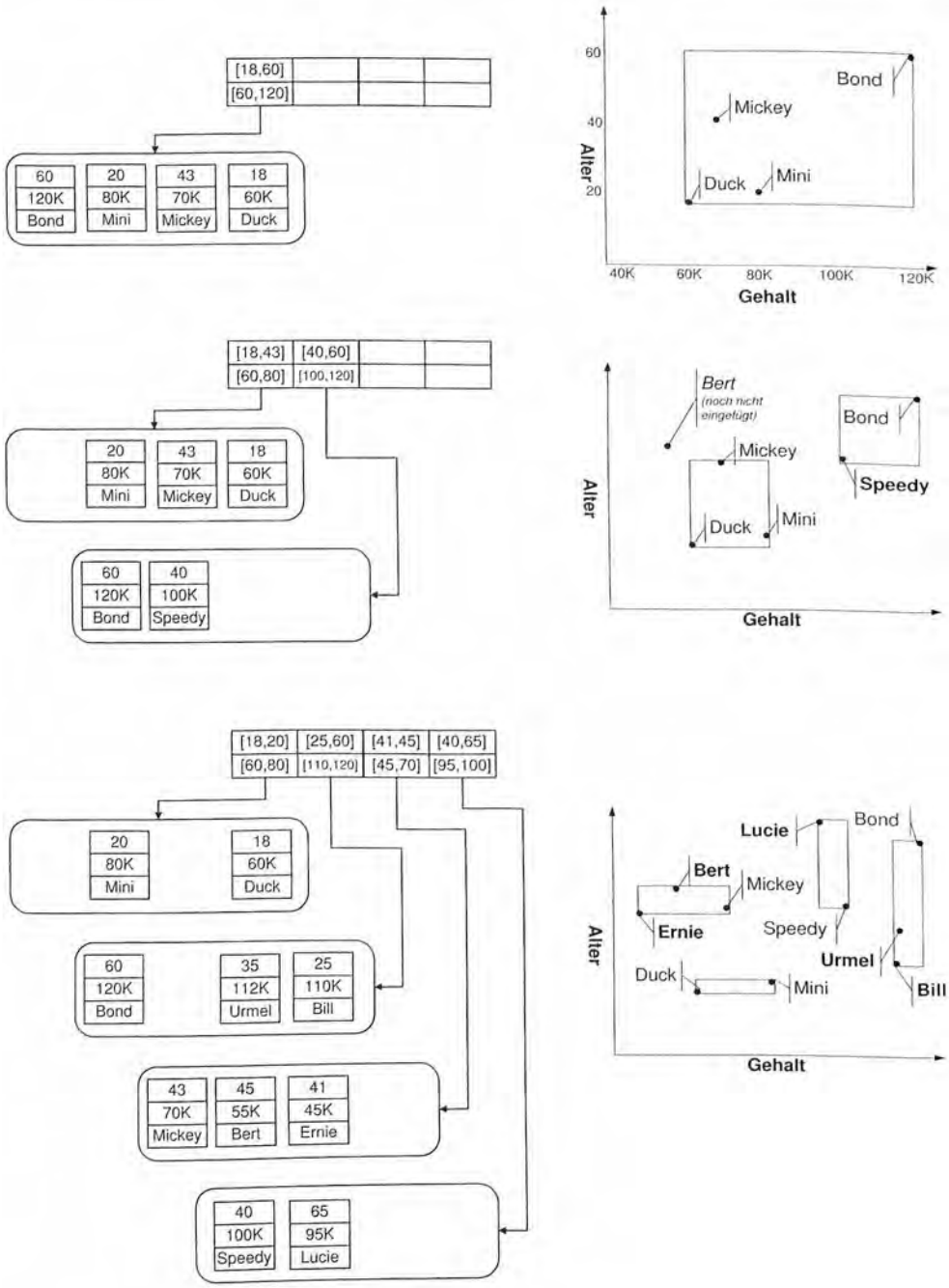


Abbildung 7.27: Ein R-Baum mit Entstehungsgeschichte

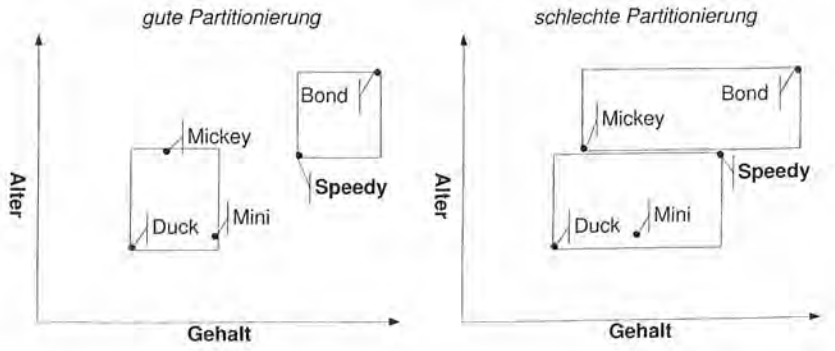


Abbildung 7.28: Gute versus schlechte Partitionierung der Datenelemente

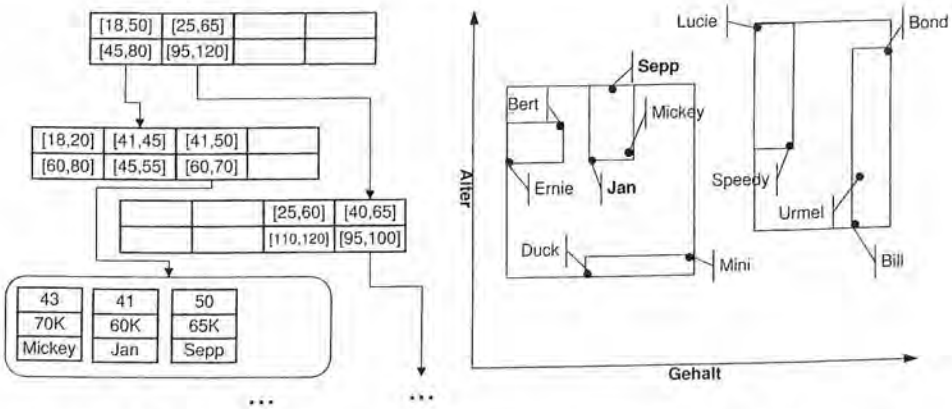


Abbildung 7.29: Die nächste Phase des R-Baums: „Nach oben" gewachsen

nach Einfügen der fett-gedruckten Datensätze gezeigt. Zu diesem Zeitpunkt ist die Wurzel voll belegt – auch bei inneren Knoten haben wir eine Kapazität von vier angenommen. Wenn wir jetzt noch *Jan* und *Sepp* (in die Box von *Ernie, Bert* und *Mickey*) einfügen, führt das zu einem Überlauf. Das Ergebnis ist in Abbildung 7.29 gezeigt: Der Baum ist um eine Stufe „nach oben" gewachsen.

In Abbildung 7.30 ist die Anfragebearbeitung im R-Baum gezeigt – allerdings ist dort ein „schlimmer" Fall gezeigt, der in der Praxis (hoffentlich) selten vorkommt. Bei einer Bereichsanfrage hat man ein Anfragefenster gegeben, das selbst eine Box repräsentiert. In unserem Fall ist es die Box mit den Intervallen [47,67] für die *Alter*-Dimension und [55K,105K] für das *Gehalt*. Wir starten an der Wurzel und gehen **jeden** Weg nach unten, dessen Box das Anfragefenster überlappt. Wir haben also jetzt die schöne Eigenschaft des B^+-Baums aufgeben müssen, wonach man immer nur einen Weg nach unten gehen muss. Die Wege sind in der Abbildung links markiert – genau wie das Anfragefenster mit gestrichelten, fetten Linien. Unten angekommen, muss man dann noch die Datensätze durchsuchen, um zu überprüfen, ob sie auch tatsächlich im Anfragefenster liegen. In unserem Fall sind es gerade mal *Sepp*

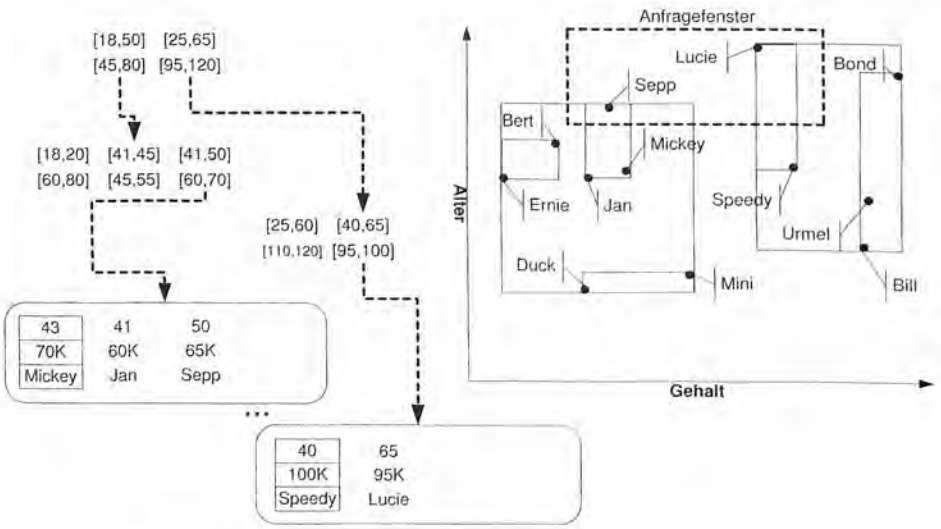


Abbildung 7.30: Anfrageauswertung im R-Baum

und *Lucie* – die anderen Datensätze liegen zwar in Boxen, die das Anfragefenster überlappen, aber selber liegen sie nicht im Anfragefenster.

Auch Punktanfragen lassen sich natürlich mit einem R-Baum auswerten – sie sind sogar einfacher auszuführen. Als Beispiel wollen wir die Person(en) ermitteln, die 65 Jahre alt sind und 100K verdienen. Das triviale Anfragefenster (der eine Datenpunkt) überlappt nur die rechte Box der Wurzel, also nehmen wir diesen Weg. Von da aus geht es weiter auf dem rechten Weg zum Blattknoten, der *Speedy* und *Lucie* enthält. Erst dort stellt man fest, dass es kein qualifizierendes Datenelement gibt. Hätte man eine 65-jährige Person mit einem Gehalt von 90K gesucht, hätte man die Suche schon an der Wurzel als erfolglos beenden können. Im Allgemeinen muss man aber selbst bei Punktanfragen mehrere Wege im R-Baum traversieren, da sich die Boxen der inneren Knoten – anders als in unserem „geschönten" Beispiel überlappen können.

7.15 Ballung logisch verwandter Datensätze

Ein weiteres wichtiges Mittel zur Zugriffsbeschleunigung stellt die sogenannte *Ballung* (engl. *Clustering*) dar. Mit der Ballung *logisch verwandter* Datensätze wird dafür gesorgt, dass Daten, die häufig zusammen benötigt werden, dicht beieinander auf dem Hintergrundspeicher liegen – idealerweise auf einer Seite.

Abbildung 7.31 skizziert den Hintergrundspeicher und den als Puffer verwendeten Hauptspeicher bei der Bearbeitung einer Anfrage der Form

select $*$
from R
where $A = x$;

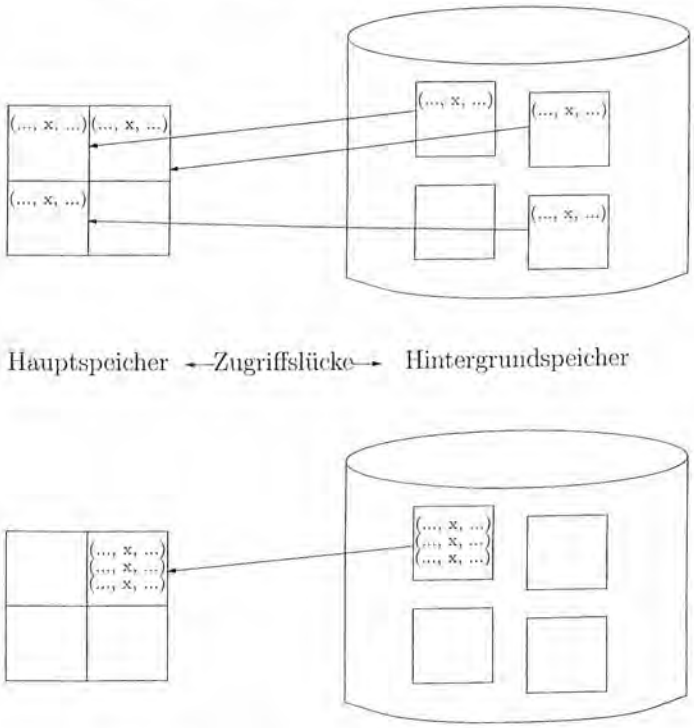


Abbildung 7.31: Lesen ungeballter und geballter Tupel

Im oberen Fall sind die drei Tupel, die den Wert x im Attribut A enthalten, auf drei unterschiedliche Seiten des Hintergrundspeichers verteilt. Es sind also – unter der Annahme, dass die Tupel mit Hilfe eines Indexes direkt gefunden werden können – drei Seitenzugriffe für den Transfer der Tupel in den Hauptspeicher notwendig. Zusätzlich werden für das Laden der drei Seiten auch drei Pufferrahmen im Hauptspeicher verschwendet. Im unteren Fall sind die Tupel geballt auf einer Seite abgespeichert, und es muss nur ein Seitenzugriff zum Laden aller benötigten Tupel investiert werden. Es ist offensichtlich, dass auf diese Weise der Aufwand erheblich reduziert werden kann.

Auf Kosten einer zusätzlichen Indirektion lassen sich Indexstrukturen mit einer Ballung verträglich machen. Es wurde bereits in Abschnitt 7.8 erwähnt, dass bei Sekundärindices nur Verweise auf Datensätze eingetragen werden. Diese Situation ist noch einmal in Abbildung 7.32 für den B^+-Baum verdeutlicht. Dort sind beim Primärindex die Datensätze direkt in den Blättern eingetragen. Während der Primärindex die Ballung festlegt, also die Anordnung der Datensätze auf den Seiten, benötigt man im Sekundärindex eine zusätzliche Indirektion, um zu den Datensätzen zu gelangen. Es lassen sich bei einem Sekundärindex also zumeist keine Ballungseffekte in den Datenseiten ausnutzen, wohl aber in den Knoten des Indexes.

Anschaulich kann man sich den Primärindex wie den Daumenindex in einem Lexikon vorstellen, wohingegen der Sekundärindex das Analogon zu einem Sachindex

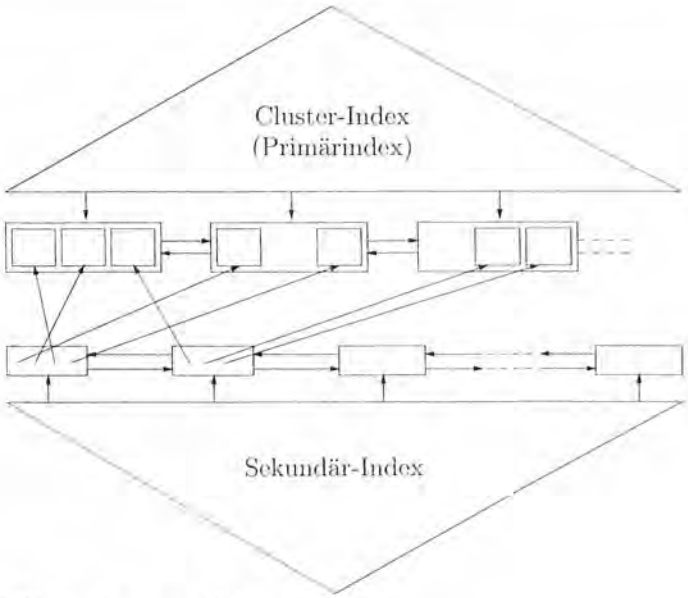


Abbildung 7.32: Indices und Ballung

in einem Lehrbuch (wie diesem) darstellt. Zu einem Schlüssel des Sekundärindexes kann es also mehrere Verweise geben, die u.U. auf unterschiedliche Seiten verweisen.

Bei Verwendung des TID-Konzepts – siehe Abschnitt 7.5 – ergibt sich aber ein Problem: Bei der Aufspaltung eines Blattknotens werden etwa die Hälfte der Datensätze in einen neuen Knoten verschoben. Nach dem TID-Konzept müsste für all diese Datensätze ein „Forward" im alten Knoten angelegt werden. Dies ist in der Regel nicht tolerierbar, da sich der Zugriff auf diese verschobenen Tupel aufgrund der Indirektion erheblich aufwendiger gestaltet. Es sind mehrere Lösungen praktikabel: Man könnte auch in den geballten Primärindices nur TIDs anstatt der Datensätze ablegen, wobei dann aber benachbarte TIDs (meist) auf dieselbe Datenseite verweisen. Wird ein neues Tupel eingefügt, wird zuerst der Index „befragt", auf welche Datenseite die benachbarten TIDs verweisen, um das neue Tupel, wenn möglich, dort zu plazieren. Eine andere Möglichkeit besteht darin, den geballten Primärindex erst anzulegen, wenn schon (fast) alle Datensätze vorhanden sind. Dazu müssen die Tupel aber neu in die Datenbank geladen werden. Die TIDs werden erst nach Aufbau des Primärindex-Baums vergeben. Erst danach werden die Sekundärindices mit den TID-Verweisen aufgebaut. Die (wenigen) in Zukunft neu hinzukommenden Tupel haben dann hoffentlich Platz in den Blattseiten, ohne zu Knotenspaltungen zu führen. Wenn doch zu viele Spaltungen vorkommen, muss man periodisch die Datenbasis neu strukturieren, indem die Datensätze neu plaziert werden und die Indices neu angelegt werden. Viele DBMS-Produkte haben spezielle „Utilities" für die effiziente Durchführung dieser Reorganisation, so dass das Datenbanksystem nur kurzzeitig außer Betrieb genommen werden muss.

Eine weitergehende Möglichkeit des Clusterings besteht in der „Materialisierung" von Beziehungen. Es ist beispielsweise denkbar, dass Vorlesungen häufig mit den

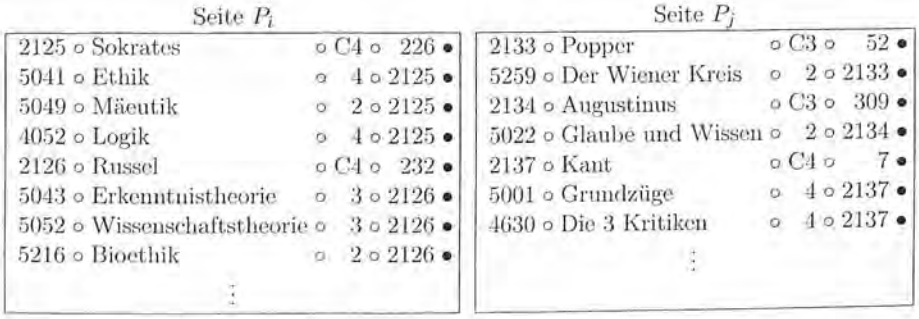


Abbildung 7.33: Verzahnte Objektballung

zugehörigen Referenten benötigt werden. Das kann unterstützt werden, indem die Referenten quasi verzahnt mit ihren Vorlesungen abgespeichert werden, wie es Abbildung 7.33 verdeutlicht. Auf diese Weise können beispielsweise die Daten von Russel mit all seinen Vorlesungen in einem Seitenzugriff gelesen werden.

7.16 Unterstützung eines Anwendungsverhaltens

Sowohl B$^+$-Bäume als auch Hashing sind Standardtechniken, die heute von den meisten DBMS angeboten werden. Während B$^+$-Bäume dort angewendet werden, wo ein gutes Verhalten „in allen Lebenslagen" gefragt ist, können Hash-Indices einen Geschwindigkeitsvorteil in bestimmten Anwendungsgebieten – genauer, bei sogenannten Punktanfragen – bringen.

Nehmen wir an, es würde ein ganz bestimmtes Tupel einer Tabelle gesucht, z.B. eine Personalnummer in der Tabelle *Professoren*. Eine entsprechende *Exact Match Query* hätte die Form

select Name
from Professoren
where PersNr = 2136;

Wurde ein B$^+$-Baum auf das Attribut *PersNr* angelegt, kann das gewünschte Tupel durch das Absteigen im Baum von der Wurzel aus gefunden werden. Bei einem Hash-Index muss zuerst das Verzeichnis gelesen werden, dann das passende Bucket. Damit ist der Hash-Index i. a. mit einem geringeren Aufwand verbunden.[5]

[5]Nehmen wir an, es wären eine Million Tupel in der Relation R gespeichert. Bei einer 3/4-Auslastung der Knoten, einer Seitengröße p von 1024 Bytes und einer Größe von vier Bytes sowohl für einen Verweis v als auch für einen Referenzschlüssel (r) kann man die Höhe des Baums mit $\lceil \log_{0.75\cdot(1+(p-v)/(v+r))}(1000000) \rceil = 4$ abschätzen. Dabei ist $\lfloor 1 + (p-v)/(v+r) \rfloor$ der maximale Verzweigungsgrad eines Knotens. Zusätzlich wird noch ein Seitenzugriff für den Blattknoten benötigt. Im Vergleich dazu braucht ein Hash-Index nur zwei Seitenzugriffe, wenn die richtige Stelle im Verzeichnis direkt gefunden werden kann. Diese Abschätzung berücksichtigt allerdings nicht, dass DBMS einen Puffer verwenden: Bei häufigem Zugriff auf den B-Baum bleiben im Allgemeinen die Wurzel und Teile des ersten Levels im Hauptspeicher gepuffert.

Wird eine andere Anfrage-Form gewählt, schneiden B^+-Bäume deutlich besser ab. Eine sogenannte *Range Query* testet, ob ein Attribut innerhalb eines bestimmten Bereichs liegt; nehmen wir für unser Beispiel an, dass die Relation *Professoren* ein Attribut *Gehalt* besitzt.

select Name
from Professoren
where Gehalt >= 90 000 **and** Gehalt <= 100 000;

Wenn es einen B^+-Baum-Index auf *Gehalt* gibt, sind seine Blätter nach dem Attribut *Gehalt* sortiert. Der Baum wird auf der Suche nach dem ersten passenden Wert (hier 90 000) abgestiegen, und von dort aus kann eine sequentielle Abarbeitung erfolgen, bis ein Tupel mit einem Attributwert größer als 100 000 gefunden wird.

Hashfunktionen hingegen können meistens nicht die Ordnung der Tupel erhalten, ohne dass darunter die gleichförmige Ausnutzung der Buckets leidet. So müssen die passenden Tupel für jeden Wert aus dem Bereich 90 000 bis 100 000 einzeln gesucht werden.

Gerade bei Bereichsanfragen ist aber auch Vorsicht bei der Verwendung nichtgeballter Sekundärindices geboten: Der Zugriff auf die Datensätze erzeugt *random I/O* – d.h. die Datensätze liegen verstreut auf der Platte – wodurch sehr hohe Plattenzugriffszeiten entstehen können. Hier ist das sequentielle Durchsuchen aller Datensätze oft effizienter, da dabei weniger *seek time* anfällt (Stichwort: *chained I/O*). Es ist Aufgabe des Anfrageoptimierers, anhand der Selektivitätsabschätzung eines derartigen Prädikats den optimalen Auswertungsplan (also Indexnutzung oder sequentielle Suche) zu generieren – siehe dazu Kapitel 8.

Generell ist ein wichtiger Faktor für den Einsatz einer Indexstruktur das Verhältnis von Leseoperationen zu Änderungsoperationen. Während durch einen Index Leseoperationen beschleunigt werden, erfordern Änderungsoperationen oftmals mehr Zugriffe auf den Hintergrundspeicher. Ein gutes Beispiel dafür ist die ISAM-Indexstruktur, bei der das Verhältnis besonders unterschiedlich ist. Daher werden ISAM-Indices überwiegend für statische Daten eingesetzt.

7.17 Physische Datenorganisation in SQL

Gerade im Bereich der physischen Datenorganisation unterscheiden sich die derzeit erhältlichen Datenbanksysteme sehr stark. Selbst in SQL-92 wurden keinerlei Maßnahmen getroffen, um zumindest einige Konzepte des physischen Entwurfs zu vereinheitlichen, wie zum Beispiel das Anlegen oder Entfernen eines Indexes auf einem Attribut. Es hat sich aber die folgende Syntax eingebürgert (am Beispiel eines Indexes auf dem Attribut *Semester* der Relation *Studenten*):

create index SemesterInd **on** Studenten(Semester);

Über den Namen *SemesterInd* könnte der Index auch wieder gelöscht werden:

drop index SemesterInd;

7.18 Übungen

7.1 Nehmen wir an, dass heutige Plattenspeicher im Durchschnitt 100.000 Stunden fehlerfrei arbeiten bis ein Fehler auftritt (MTBF: mean time before failure). Berechnen Sie die MTBF für ein RAID 0 Platten-Array bestehend aus 100 solcher Platten. Beachten Sie, dass bei RAID 0 der Defekt einer Platte immer auch zu einem Datenverlust führt, so dass die MTBF mit der mittleren Dauer bis zum Datenverlust (mean time before data loss (MTDL)) übereinstimmt. Wie sieht das bei anderen RAID-Levels aus? Berechnen Sie die MTDL-Zeit für ein RAID 3- oder RAID 5-System bestehend aus 9 Platten (einschließlich der Parity-Platte). Wir nehmen an, dass die Reparatur (bzw. der Ersatz) einer defekten Platte 24 Stunden dauert.

7.2 Skizzieren Sie einen Algorithmus zum Einfügen eines Datensatzes in eine Datei mit ISAM-Index. Dabei soll so weit wie möglich auf das Verschieben von Seiten verzichtet werden.

7.3 Fügen Sie in einen anfänglich leeren B-Baum mit $k = 2$ die Zahlen eins bis zwanzig in aufsteigender Reihenfolge ein. Was fällt Ihnen dabei auf?

7.4 Beschreiben Sie das Löschen in einem B-Baum in algorithmischer Form, ähnlich der Beschreibung des Einfügevorgangs in Abbildung 7.13.

7.5 Modifizieren Sie den Einfüge- und Löschalgorithmus für den B-Baum so, dass eine Minimalbelegung von 2/3 des Platzes in den Knoten garantiert werden kann. Hinweis: Betrachten Sie beim Löschen den linken und rechten Nachbarn des Knotens, in dem gelöscht wird. Beim Aufsplitten werden zwei Knoten gleichzeitig betrachtet.

7.6 [Helman (1994)] Der vorgestellte B-Baum geht von der Duplikatfreiheit der Schlüssel aus. Eine einfache Erweiterung wäre es, bei Duplikaten anstelle des TID's einen Verweis auf einen externen „Mini-Index" zu hinterlassen. Denken Sie sich sinnvolle Datenstrukturen und Algorithmen dafür aus.

7.7 Geben Sie Algorithmen für das Einfügen und Löschen von Schlüsseln in B^+-Bäumen an.

7.8 Geben Sie für den B- und den B^+-Baum je eine Formel an, mit der man die obere und untere Schranke für die Höhe des Baums bei gegebenem k, k^* und n (der Anzahl der eingetragenen TIDs) bestimmen kann.

7.9 Bestimmen Sie k und k^* für einen B^+-Baum bei gegebener Seitengröße p und Schlüsselgröße s. Verweise innerhalb des Baums (V_i, P, N) haben die Größe v, die TIDs die Größe d. Berechnen Sie k und k^* für den Fall $p = 4096$, $s = 4$, $v = 6$ und $d = 8$.

7.10 Beim Hashing wird der Modulofunktion häufig eine *Faltung* vorgeschaltet. Das kann beispielsweise für Zahlen die Quersumme sein und für Zeichenketten die Summe der Buchstabenwerte. Fügen Sie die Studenten aus Abbildung 3.8 in eine Hashtabelle der Größe vier mit Überlaufbuckets ein und schalten Sie

bei der Berechnung der Hashwerte zusätzlich eine Quersummenfunktion vor. Werden die Studenten jetzt gleichmäßiger verteilt?

7.11 Gegeben sei eine erweiterbare Hashtabelle mit globaler Tiefe t. Wie viele Verweise zeigen vom Verzeichnis auf einen Behälter mit lokaler Tiefe t'?

7.12 Was wäre in dem Beispiel zum erweiterbaren Hashing passiert, wenn Kopernikus die Personalnummer 2121 gehabt hätte?

7.13 Warum wurde die binäre Darstellung *rückwärts* verwendet?

7.14 Zum intuitiven Verständnis könnten Sie das Adressverzeichnis einer erweiterbaren Hashtabelle mittels eines binären Trie veranschaulichen.

7.15 Geben Sie eine algorithmische Beschreibung für die Operationen suchen, einfügen und löschen in einer erweiterbaren Hashtabelle an.

7.16 Erfinden Sie ein Verfahren um mit dem Mechanismus des erweiterbaren Hashings auch direkt den Datensatz innerhalb eines Buckets zu finden. Hinweis: Eine weitere Möglichkeit der Kollisionsbehandlung beim Hashing ist beispielsweise, einfach den nächsten freien Platz zu benutzen.

7.17 Entwickeln Sie eine Heuristik zur Partitionierung eines Blattknotens im R-Baum. Welche Komplexität hätte ein optimales Verfahren, das die Boxengröße minimiert?

7.18 Fügen Sie eine dritte Dimension, sagen wir *Geschlecht*, mit hinzu und bauen Sie den Beispiel-R-Baum aus Abschnitt 7.14 neu auf. Illustrieren Sie die einzelnen Phasen im Aufbau des R-Baums.

7.19 Helmer, Neumann und Moerkotte (2003) haben ein adaptives Verfahren für das erweiterbare Hashing entwickelt, mit dem Schieflagen in der Verteilung der Daten ohne wiederholte Directory-Verdoppelung ausgeglichen werden. Bei dem Standard-Verfahren muss das gesamte Directory verdoppelt werden – auch wenn nur punktuell Überläufe stattfinden. In dem neuen Verfahren wird im Falle einer eklatanten Schieflage (skew) nur für den Überlaufbereich eine weitere (also partiell verdoppelte) Hashtabelle allokiert. Konzipieren Sie dieses Verfahren in Pseudo-Code und visualisieren Sie an Hand von Beispielen dieses adaptive Verfahren.

7.20 Sie wollen eine Telefonauskunft realisieren, die sowohl die Vorwärtssuche (finde zu einem gegebenen Namen die Telefonnummer) als auch die Rückwärtssuche (finde zu einer gegebenen Telefonnummer den Namen) effizient unterstützt. Wir gehen davon aus, dass es in Ihrer Telefonauskunft, die alle Telefonnummern der Welt abdecken soll, 10 Milliarden Einträge gibt. Für die Vorwärtssuche entscheiden Sie sich für einen B-Baum als Indexstruktur, damit man auch die Bereichssuche unterstützen kann. Beispielsweise könnte man dann den Bereich von „Maier“ bis „Meyer“ bei unbekannter Schreibweise suchen. Welche Höhe hat der resultierende B-Baum und wieviel Speichervolumen nimmt allein dieser B-Baum in Anspruch. Wir gehen, wie in Abschnitt 7.11 von einer Knotengröße von 8KB aus.

Für die Rückwärtssuche ist die Bereichssuche irrelevant, da man nur Punktanfragen unterstützen muss. Deshalb entscheiden wir uns für das erweiterbare Hashing als Indexstruktur. Wie groß wird das Verzeichnis, wenn die Bucketgröße bei 8KB liegt. Kann man diese Telefonauskunft mit 10 Mrd. Einträgen auf heute marktgängigen Handys vorinstallieren, so dass man sowohl die Vorwärtssuche als auch die Rückwärtssuche lokal ausführen kann? Der Vorteil wäre, dass man zu jedem eingehenden Anruf automatisch den Namen eingeblendet bekäme und nicht nur die Telefonnummer.

7.19 Literatur

Detaillierte Informationen zur RAID-Technologie findet man in dem Übersichtspapier von Chen et al. (1994), den Erfindern dieser Techniken. Weikum und Zabback (1993a) befassen sich mit dem Problem der Datenallokation in Platten-Arrays, um möglichst gleichmäßige Auslastungen – und damit eine hohe Parallelität der Plattenzugriffe – zu erzielen. In einem Folgeartikel behandeln Weikum und Zabback (1993b) die Fehlertoleranz und gehen auch auf die Ausfallwahrscheinlichkeiten ein. Aktuelle Produktinformationen zu RAID-Systemen findet man auf den Web-Seiten der Hardwareanbieter, wie z.B. Sun Microsystems (1997). Berchtold et al. (1997) behandeln das Striping von Daten zum Zweck der optimierten parallelen Auswertung von Ähnlichkeits-Suchanfragen in Multimedia-Datenbanken. Scheuermann, Weikum und Zabback (1998) behandeln die Abbildung von Daten auf Platten zur Lastbalancierung. Seeger (1996) behandelt die Optimierung von Plattenzugriffen auf Seitenmengen.

Ein Buch, das sich speziell mit Fragen der physischen Datenorganisation auseinandersetzt, wurde von Shasha und Bonnet(2002)) geschrieben. Dort werden Daumenregeln für den Einsatz der verschiedenen Techniken vorgestellt und beispielhaft in Szenarien eingesetzt. Automatisiertes physisches Design wird von Rozen und Shasha (1991) besprochen. Weikum et al. (1994) untersuchen automatisches Tuning im Rahmen von Sperren und Pufferstrategien. Scholl und Schek (1992) beschreiben das COCOON Projekt, in dem die Optimierung der physischen Datenorganisation ein zentrales Anliegen ist.

Die möglichen Strategien zur Pufferverwaltung wurden sehr systematisch von Effelsberg und Härder (1984) untersucht. Küspert, Dadam und Günauer (1987) haben eine Pufferverwaltung für das am IBM Wissenschaftlichen Zentrum Heidelberg entwickelte Datenbanksystem AIM entwickelt. O'Neil, O'Neil und Weikum (1993) entwarfen das für Datenbankpuffer besonders sinnvolle Seitenersetzungsverfahren LRU/k, das darauf beruht, die Seiten auf der Basis ihrer letzten k Referenzen zu ersetzen. Johnson und Shasha (1994) schlugen eine effiziente Realisierung für eine Approximation des LRU/2-Verfahrens vor.

Zum Gebiet Indexstrukturen gibt es sehr vielfältige Literatur. B-Bäume wurden Anfang der siebziger Jahre von Bayer und McCreight (1972) vorgestellt. Im dritten Band seines „The Art of Computer Programming" stellt Knuth (1973) einige Varianten von B-Bäumen vor. Dort werden auch verschiedene Hashfunktionen untersucht. Comer (1979) beschreibt den „allgegenwärtigen" B-Baum in einem Übersichtsartikel der Computing Surveys. Trotz des „hohen Alters" des B-Baums werden immer noch

neue Optimierungen bei seiner Realisierung gefunden, beispielsweise das Verteilen des Baums auf mehrere Festplatten [Seeger und Larson (1991)]. Bercken, Seeger und Widmayer (1997) und Bercken und Seeger (2001) behandeln das Bulk-Loading von Indexstrukturen. Demgegenüber haben Gärtner et al. (2001) eine Technik zur optimierten Löschung von Indexeinträgen entwickelt. Aktuelle deutschsprachige Werke über Datenstrukturen wurden z.B. von Güting und Dieker (2003) und Ottmann und Widmayer (2002) verfasst. Ein deutschsprachiges Buch über Implementierungstechniken für Datenbanksysteme ist das Datenbankhandbuch, das von Lockemann und Schmidt (1987) herausgegeben wurde. Aktueller ist das Buch von Härder und Rahm (2001) über Datenbank-Implementierungstechniken. Knuth (1973) diskutiert ausführlich Hashverfahren für den Hauptspeicher. Dynamische Hashverfahren sind noch nicht so lange verbreitet. Das hier beschriebene erweiterbare Hashing wurde von Fagin et al. (1979) vorgestellt. Larson (1988) gibt einen Überblick über zwei dynamische Hashverfahren, die kein Verzeichnis verwalten müssen. Eine neuere Variante stellt Ahn (1993) vor. Neubert, Görlitz und Benn (2001) haben die inhaltsbasierte Indexierung untersucht, bei der man ähnliche Objekte „clustert".

Besonders in geografischen Informationssystemen ist es notwendig, mehrdimensionale Daten zu indizieren. Günther und Schek (1991) gaben einen Sammelband über fortgeschrittene Datenstrukturen zur Realisierung sogenannter „Spatial Databases" heraus. Populäre mehrdimensionale Indexstrukturen sind das Grid-File von Nievergelt, Hinterberger und Sevcik (1984), der K-D-B-Baum von Robinson (1981), der von uns behandelte R-Baum von Guttman (1984), der LSD'-Baum von Henrich, Six und Widmayer (1989) und, neueren Datums, der R*-Baum von Beckmann et al. (1990), der eine deutliche Verbesserung des R-Baums darstellt, und schließlich der Buddy-Baum, beschrieben von Seeger und Kriegel (1990). Hinrichs (1985) hat Implementierungstechniken für das Grid-File konzipiert. Ein umfassender Survey stammt von Gaede und Günther (1998). Ramsak et al. (2000) beschreiben den UB-Baum, den sogenannten Universal B-Tree. Dabei handelt es sich um eine mehrdimensionale Indexstruktur auf der Basis „normaler" B^+-Bäume. Durch eine sogenannte *space filling curve* werden mehrdimensionale Datensätze auf eine Dimension „projeziert". Markl, Zirkel und Bayer (1999) erläutern einen Algorithmus für das Sortieren mit dieser mehrdimensionalen Indexstruktur.

Kailing et al. (2006) erweitern metrische Indexstrukturen für die Auswertung von Bereichsanfragen. Der TS-Baum von Assent et al. (2008) erlaubt die Indexierung von Zeitreihen. Augsten, Böhlen und Gamper (2006) haben einen Index für hierarchische Daten entwickelt.

Für die erweiterte Funktionalität, die die in Kapitel 13 und 14 besprochenen objektorientierten und objekt-relationalen Datenbanken bieten, sind maßgeschneiderte Indexstrukturen konzipiert worden. Stichworte hier sind Multiset-Indices [Kilger und Moerkotte (1994)], Pfad-Indices [Kemper und Moerkotte (1992)] und Funktionenmaterialisierung [Kemper, Kilger und Moerkotte (1994)]. Eine Übersicht geben Kemper und Moerkotte (1995) und Bertino (1993). Um die Ortsunabhängigkeit von Objekten zu gewähren, gibt es in objektorientierten Datenbanken ein dem TID ähnliches Konzept [Eickler, Gerlhof und Kossmann (1995)]. Pufferungsstrategien, die flexibel sowohl Seiten als auch Objekte verwalten können, werden von Kemper und Kossmann (1994) beschrieben. Gerlhof et al. (1993) untersuchen die Effizienz von Ballungsverfahren in Objektbanken.

8. Anfragebearbeitung

Wie wir bereits in Kapitel 4 festgestellt hatten, werden Anfragen im Allgemeinen deklarativ auf dem logischen Schema formuliert. Dies unterstützt die Idee der Datenunabhängigkeit: Die Anfrage des Benutzers ist nicht vom physischen Schema der Datenbasis – also der Speicherungsstruktur – abhängig. Bei der Anfragebearbeitung muss nun die Grenze von der logischen zur physischen Ebene überschritten und eine geeignete Implementierung der Anfrage gefunden werden. Der Weg dahin ist in Abbildung 8.1 skizziert.

Zunächst wird die Anfrage syntaktisch und semantisch analysiert und in einen äquivalenten Ausdruck der relationalen Algebra umgewandelt. Bei diesem Schritt werden auch die vorkommenden Sichten durch ihre definierende Anfrage ersetzt.

Mit der relationalen Algebra als Eingabe wird die *Anfrageoptimierung* gestartet. Die Anfrageoptimierung sucht zu einem gegebenen algebraischen Ausdruck eine effiziente Implementierung, einen sogenannten *Auswertungsplan* (engl. query evaluation plan, QEP). Dieser Auswertungsplan kann dann entweder kompiliert oder bei interaktiven Anfragen direkt interpretativ gestartet werden.

Auch ein Algorithmus zur Implementierung eines Operators kann wieder als Operator einer *physischen Algebra* angesehen werden. Genau wie relationale Operatoren „verbraucht" eine Implementierung ein oder mehrere Eingabequellen, um eine oder mehrere Ausgaben zu erzeugen.

Man spricht von einem „Anfrageoptimierer", da es zu einer gegebenen deklarativen Anfrage eine unter Umständen große Menge möglicher Auswertungsstrategien gibt, die sich in ihrer Ausführungsdauer stark unterscheiden. Leider ist es nicht auf effiziente Weise möglich, die schnellste dieser Alternativen zu finden. Man ist auf eine Art „try and error"-Verfahren angewiesen, das mehr oder weniger gezielt Alternativen erzeugt und deren Ausführungsdauer (oder *Kosten*) mit Hilfe eines *Kostenmodells* abschätzt. Das Kostenmodell arbeitet auf der Grundlage von Schemainformationen, dem Wissen über den Aufwand der eingesetzten Algorithmen und Statistiken über Relationen, Indexstrukturen und der Verteilung der Attributwerte.

Die Alternativen entstehen auf zwei verschiedene Weisen, im Folgenden *logische* und *physische Optimierung* genannt. Zum einen besteht die Möglichkeit, auf relationenalgebraische Ausdrücke Äquivalenzumformungen anzuwenden; z.B. können die Argumente der Joinoperation aufgrund ihrer Kommutativität vertauscht werden. Zum anderen gibt es für einen Operator der logischen Algebra oft mehrere unterschiedliche Implementierungen, d.h. Übersetzungsmöglichkeiten in die physische Algebra. In beiden Fällen werden *Heuristiken* zur Steuerung der Alternativengenerierung eingesetzt. Heuristiken repräsentieren Erfahrungswerte über die sinnvolle Anwendung bestimmter Umformungsregeln.

Dieses Kapitel ist zweigeteilt: Zunächst werden die Eigenschaften der relationalen Algebra vorgestellt und die Anwendung von Heuristiken demonstriert. Im zweiten Teil werden Implementierungstechniken und Kostenmaße vorgestellt.

Abbildung 8.1: Ablauf der Anfragebearbeitung

8.1 Logische Optimierung

Ausgangspunkt der Optimierung ist eine sogenannte algebraische Normalform, die schon in Kapitel 4 eingeführt wurde. Diese Normalform ist noch einmal zur Wiederholung in Abbildung 8.2 dargestellt: Eine SQL-Anfrage der allgemeinen Form **select** ... **from** ... **where** ... wird in einen algebraischen Ausdruck mit Kreuzprodukten der Basisrelationen, gefolgt von einer Selektion und einer Projektion umgewandelt.

In diesem Kapitel wird oft auch die anschauliche Baumdarstellung algebraischer Ausdrücke verwendet, um deren Manipulation besser zu verdeutlichen. Die Blätter des Baums werden dabei von Basisrelationen gebildet, die inneren Knoten von Operatoren der relationalen Algebra. Auf diese Weise wird der „Fluss" der Daten verdeutlicht.

Ein einfaches Beispiel soll den Sinn der algebraischen Optimierung veranschaulichen. In SQL bestimmt man die von Popper gehaltenen Vorlesungen mit

select Titel
from Professoren, Vorlesungen
where Name = 'Popper' **and** PersNr = gelesenVon;

Laut Kapitel 4 lässt sich diese Anfrage in den folgenden algebraischen Ausdruck übersetzen:

$$\Pi_{\text{Titel}}(\sigma_{\text{Name='Popper'} \wedge \text{PersNr=gelesenVon}}(\text{Professoren} \times \text{Vorlesungen}))$$

Überlegen wir uns, welche Schritte zur Berechnung des Ausdrucks notwendig sind. Das Kreuzprodukt verknüpft alle Professoren und Vorlesungen, insgesamt ergeben sich bei sieben Professoren und zehn Vorlesungen also $7 \cdot 10 = 70$ Tupel. Aus diesen

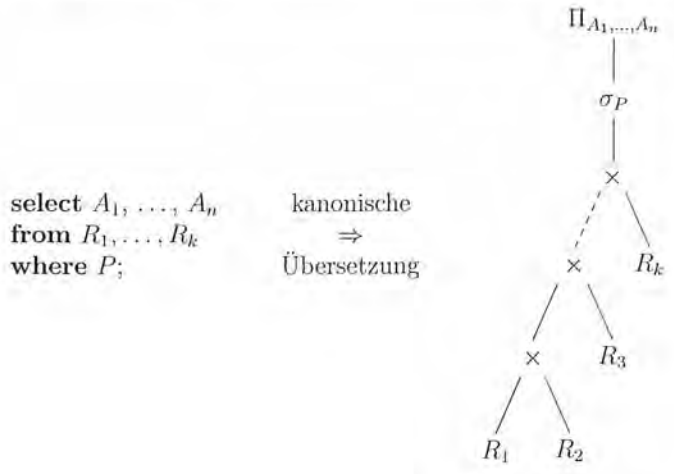


Abbildung 8.2: Kanonische Übersetzung einer SQL-Anfrage

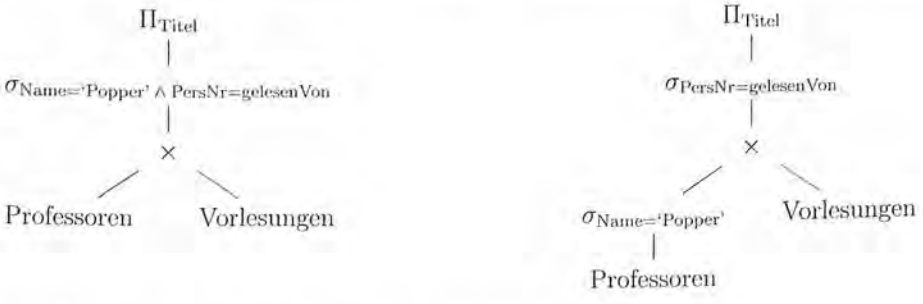


Abbildung 8.3: Baumdarstellung der algebraischen Ausdrücke

70 Tupeln werden diejenigen ausgewählt, die die Selektionsbedingung erfüllen, in diesem Fall ist es nur eines.

Offensichtlich wurde hier viel zu viel Arbeit investiert. Eine einfache Verbesserung wäre es, zuerst den „richtigen" Professor zu finden und dann das Kreuzprodukt zu bilden, also

$$\Pi_{\text{Titel}}(\sigma_{\text{PersNr=gelesenVon}}(\sigma_{\text{Name='Popper'}}(\text{Professoren}) \times \text{Vorlesungen}))$$

Auf diese Weise werden zuerst die sieben Professoren durchsucht. Anschließend wird der verbleibende Professor mit den zehn Vorlesungen verknüpft. Das Ergebnis kann also in $7 + 10 = 17$ Schritten bestimmt werden. Damit haben wir bereits die erste wichtige Heuristik der Anfrageoptimierung kennengelernt: das Aufbrechen von Selektionen und deren Verschieben in den Ausdruck hinein.

Dieser Auswertungsplan ließe sich dann noch durch die Zusammenfassung der Selektion $\sigma_{\text{PersNr=gelesenVon}}$ und des Kreuzprodukts $\times$ zu einem Join $\bowtie_{\text{PersNr=gelesenVon}}$ weiter optimieren. Abbildung 8.3 zeigt die obigen Ausdrücke in der Baumdarstellung.

8.1.1 Äquivalenzen in der relationalen Algebra

Vor einer systematischeren Untersuchung der Transformation von relationalen Ausdrücken sollten erst einmal die möglichen Regeln vorgestellt werden. Seien R, R_1, R_2,... Relationen (seien es Basis- oder abgeleitete Relationen, d.h. Zwischenergebnisse), p, q, p_1, p_2,... Bedingungen, l_1, l_2,... Attributmengen und *attr* die Abbildung von Bedingungen auf die Menge der in ihnen enthaltenen Attribute (z.B. $attr(\text{Name} = \text{'Popper'}) = \{\text{Name}\}$). Nach wie vor bezeichnen wir mit $\mathcal{R}$ das Schema (also die Menge der Attribute) und mit R die aktuelle Ausprägung einer Relation. Dann gilt:

1. Join, Vereinigung, Schnitt und Kreuzprodukt sind kommutativ, also:

$$\begin{aligned} R_1 \bowtie R_2 &= R_2 \bowtie R_1 \\ R_1 \cup R_2 &= R_2 \cup R_1 \\ R_1 \cap R_2 &= R_2 \cap R_1 \\ R_1 \times R_2 &= R_2 \times R_1 \end{aligned}$$

2. Selektionen sind untereinander vertauschbar:

$$\sigma_p(\sigma_q(R)) = \sigma_q(\sigma_p(R))$$

3. Join, Vereinigung, Schnitt und Kreuzprodukt sind assoziativ, also:

$$\begin{aligned} R_1 \bowtie (R_2 \bowtie R_3) &= (R_1 \bowtie R_2) \bowtie R_3 \\ R_1 \cup (R_2 \cup R_3) &= (R_1 \cup R_2) \cup R_3 \\ R_1 \cap (R_2 \cap R_3) &= (R_1 \cap R_2) \cap R_3 \\ R_1 \times (R_2 \times R_3) &= (R_1 \times R_2) \times R_3 \end{aligned}$$

4. Konjunktionen in einer Selektionsbedingung können in mehrere Selektionen aufgebrochen, bzw. nacheinander ausgeführte Selektionen können durch Konjunktionen zusammengefügt werden.

$$\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$$

5. Geschachtelte Projektionen können eliminiert werden.

$$\Pi_{l_1}(\Pi_{l_2}(\ldots(\Pi_{l_n}(R))\ldots)) = \Pi_{l_1}(R)$$

   Damit eine solche Schachtelung überhaupt sinnvoll ist, muss gelten:

$$l_1 \subseteq l_2 \subseteq \ldots \subseteq l_n \subseteq \mathcal{R} = \mathbf{sch}(R)$$

6. Eine Selektion kann an einer Projektion „vorbeigeschoben" werden, falls die Projektion keine Attribute aus der Selektionsbedingung entfernt. Es gilt also:

$$\Pi_l(\sigma_p(R)) = \sigma_p(\Pi_l(R)), \text{ falls } attr(p) \subseteq l$$

7. Eine Selektion kann an einer Joinoperation (oder einem Kreuzprodukt) vorbeigeschoben werden, falls sie nur Attribute *eines* der beiden Join-Argumente verwendet. Enthält die Bedingung p beispielsweise nur Attribute aus $\mathcal{R}_1$, dann gilt:

$$\begin{aligned}\sigma_p(R_1 \bowtie R_2) &= \sigma_p(R_1) \bowtie R_2\\ \sigma_p(R_1 \times R_2) &= \sigma_p(R_1) \times R_2\end{aligned}$$

8. Auf ähnliche Weise können auch Projektionen verschoben werden. Hier muss allerdings beachtet werden, dass die Joinattribute bis zum Join erhalten bleiben.

$$\begin{aligned}&\Pi_l(R_1 \bowtie_p R_2) = \Pi_l(\Pi_{l_1}(R_1) \bowtie_p \Pi_{l_2}(R_2)) \text{ mit}\\ &\quad l_1 = \{A | A \in \mathcal{R}_1 \cap l\} \cup \{A | A \in \mathcal{R}_1 \cap attr(p)\} \text{ und}\\ &\quad l_2 = \{A | A \in \mathcal{R}_2 \cap l\} \cup \{A | A \in \mathcal{R}_2 \cap attr(p)\}\end{aligned}$$

9. Selektionen können mit Mengenoperationen wie Vereinigung, Schnitt und Differenz vertauscht werden, also:

$$\begin{aligned}\sigma_p(R \cup S) &= \sigma_p(R) \cup \sigma_p(S)\\ \sigma_p(R \cap S) &= \sigma_p(R) \cap \sigma_p(S)\\ \sigma_p(R - S) &= \sigma_p(R) - \sigma_p(S)\end{aligned}$$

10. Der Projektionsoperator kann mit der Vereinigung vertauscht werden. Sei $\mathbf{sch}(R_1) = \mathbf{sch}(R_2)$, dann gilt

$$\Pi_l(R_1 \cup R_2) = \Pi_l(R_1) \cup \Pi_l(R_2)$$

Eine Vertauschung der Projektion mit Durchschnitt und Differenz ist allerdings nicht zulässig (siehe Aufgabe 8.1).

11. Eine Selektion und ein Kreuzprodukt können zu einem Join zusammengefasst werden, wenn die Selektionsbedingung eine Joinbedingung ist, sie also Attribute einer Argumentrelation mit Attributen der anderen vergleicht. Für Equijoins gilt beispielsweise

$$\sigma_{R_1.A_1=R_2.A_2}(R_1 \times R_2) = R_1 \bowtie_{R_1.A_1=R_2.A_2} R_2$$

12. Auch an Bedingungen können Veränderungen vorgenommen werden. Beispielsweise kann eine Disjunktion mit Hilfe von DeMorgans Gesetz in eine Konjunktion umgewandelt werden, um vielleicht später die Anwendung von Regel 4 zu ermöglichen:

$$\begin{aligned}\neg(p_1 \vee p_2) &= \neg p_1 \wedge \neg p_2\\ \neg(p_1 \wedge p_2) &= \neg p_1 \vee \neg p_2\end{aligned}$$

Weiterhin ist diese Regel anwendbar um Negationen „von außen nach innen" zu schieben.

```
select distinct s.Semester
from Studenten s, hören h,
     Vorlesungen v, Professoren p
where p.Name = 'Sokrates' and
      v.gelesenVon = p.PersNr and
      v.VorlNr = h.VorlNr and
      h.MatrNr = s.MatrNr;
```

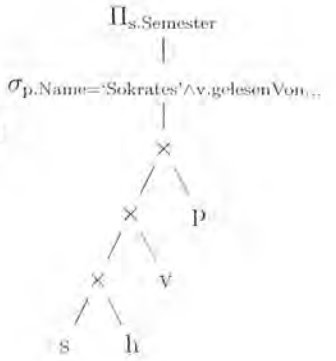


Abbildung 8.4: Die Ausgangsanfrage und ihre kanonische Übersetzung

8.1.2 Anwendung der Transformationsregeln

Anhand eines komplexeren Beispiels können wir nun eine typische Vorgehensweise bei der Anfrageoptimierung vorstellen. Die Grundidee besteht darin, die Regeln so anzuwenden, dass die Ausgaben der einzelnen Operatoren möglichst klein bleiben. Das ist umso wichtiger, wenn die Ausgaben aufgrund Hauptspeichermangels temporär auf dem Hintergrundspeicher abgelegt werden müssen.

Die zu optimierende Anfrage lautet: In welchen Semestern sind die Studenten, die Vorlesungen von Sokrates hören? Sie ist in Abbildung 8.4 sowohl in SQL als auch in der Baumdarstellung der kanonischen Übersetzung in die relationale Algebra abgebildet. In der Baumdarstellung wurden aus Platzgründen die abgekürzten Namen der Relationen verwendet.

Als erstes werden die Konjunktionsglieder der Selektion „aufgebrochen" (Regel 4). Es entstehen vier Selektionen, die einzeln innerhalb des Ausdrucks verschoben werden können (Regeln 2, 6, 7 und 9). Es ist sinnvoll, eine Selektion so früh wie möglich einzusetzen und bereits einen großen Anteil später nicht mehr benötigter Tupel auszusortieren. In Abbildung 8.5 wird die Auswahl des „richtigen" Professoren-Tupels – also das mit dem Namen „Sokrates" – direkt getroffen, so dass am Kreuzprodukt nur noch ein Professor und nicht wie vorher alle Professoren teilnehmen. Ferner wurden die Vergleiche von *MatrNr* und *VorlNr* unmittelbar über der Stelle plaziert, wo beide im Vergleich benötigten Attribute das erste Mal gleichzeitig auftreten.

Wie bereits in Kapitel 3 beschrieben, sind Joinoperationen Kreuzprodukten vorzuziehen, da Kreuzprodukte ein Zwischenergebnis stark aufblähen würden. Es ist daher sinnvoll, wann immer möglich, Kreuzprodukte in Joins umzuwandeln (Regel 11). In unserem Fall können alle Kreuzprodukte durch Joinoperationen ersetzt werden, wie Abbildung 8.6 zeigt.

Nun bestimmen wir die Reihenfolge der Joinoperationen. Mit Hilfe der Kommutativität und der Assoziativregel (Regel 3) kann die Joinreihenfolge verändert werden. Dieser Schritt allein ist schon ein komplexes Thema, und es gibt keine effi-

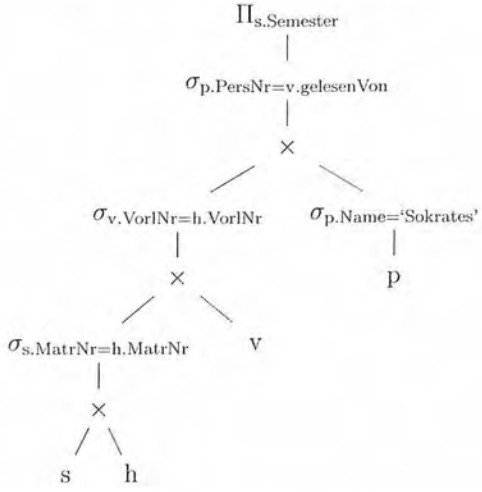


Abbildung 8.5: Aufbrechen und Verschieben von Selektionen

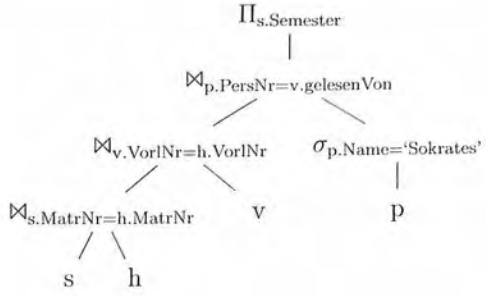


Abbildung 8.6: Zusammenfassen von Selektionen und Kreuzprodukten

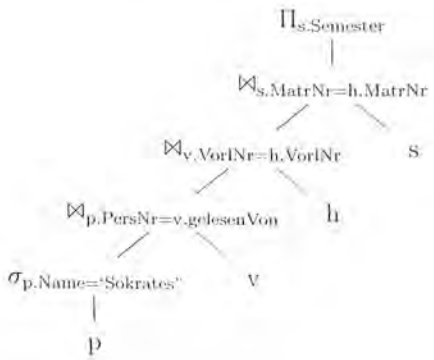


Abbildung 8.7: Bestimmung der Joinreihenfolge

ziente Methode, in jedem Fall eine Reihenfolge zu finden, die eine minimale Größe der Zwischenergebnisse garantiert. In unserem überschaubaren Beispiel kann man sich aber verdeutlichen, dass es nur einen Professor mit Namen Sokrates gibt, der nur drei Vorlesungen hält. Daher sollte der Join von *Professoren* mit *Vorlesungen* als erstes vorgenommen werden. Dies ist in Abbildung 8.7 dargestellt.

Eine mögliche Heuristik zur Bestimmung der vollständigen Joinreihenfolge könnte so verlaufen: Mit Hilfe der sogenannten *Selektivität*, mit der man die Kardinalität des Joins relativ zum Kreuzprodukt abschätzt und die noch in Abschnitt 8.3.1 vorgestellt wird, ist es möglich, die Ergebnisgröße eines Joins zu bestimmen. Man verbindet zuerst die beiden Relationen, die nach dieser Abschätzung das kleinste Zwischenergebnis liefern. Im Beispiel würden die Joins *Vorlesungen* $\bowtie$ *hören* und *hören* $\bowtie$ *Studenten* jeweils 13 Tupel liefern. Sokrates aber hält, wie oben bereits erwähnt, nur drei Vorlesungen. Also wird Sokrates zuerst mit seinen Vorlesungen verbunden. Im zweiten Schritt wählt man die Relation aus den verbleibenden, die das kleinste Zwischenergebnis beim Join mit der aus dem ersten Schritt hervorgegangenen Relation erzeugt. Das wäre im Beispiel die Relation *hören*, da eine Verbindung von *Professoren* $\bowtie$ *Vorlesungen* mit *Studenten* zu einem Kreuzprodukt entarten würde. Im dritten und letzten Schritt bleibt nur die Relation *Studenten* übrig, also wird sie mit dem Ergebnis des zweiten Schrittes verbunden.

Zum Vergleich wollen wir die Größe der Zwischenergebnisse der alten und neuen Version anhand der Beispielausprägung (aus Abbildung 3.8, Seite 86) bestimmen.

In der neuen Version enthält das Zwischenergebnis nach dem Verbund von *Professoren* und *Vorlesungen*, wie gesagt, drei Tupel (die Vorlesungen Ethik, Mäeutik und Logik). Zu diesen drei Vorlesungen existieren vier Einträge in der Relation *hören*. Wird dieses Ergebnis wiederum mit *Studenten* verbunden, ergeben sich keine neuen Tupel, die vorhandenen werden lediglich um die Informationen aus *Studenten* „angereichert". Die Summe der Zwischenergebnisse der Joins ist hier also $3+4+4 = 11$.

Die alte Version aus Abbildung 8.6 verbindet zuerst die Relationen *Studenten* und *hören*. Im Ergebnis befinden sich für unsere Beispielausprägung 13 Tupel. Bei einem Join mit *Vorlesungen* ergeben sich keine weiteren Tupel. Durch den letzten

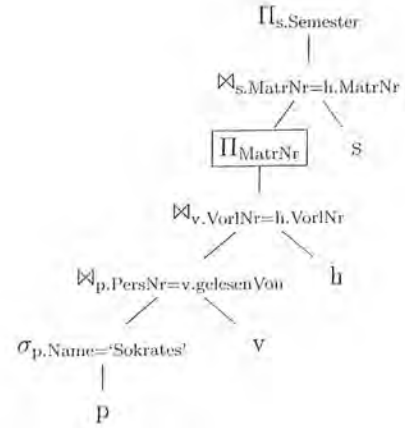


Abbildung 8.8: Einfügen und Verschieben von Projektionen

Join werden nur die Hörer von Sokrates weitergereicht, das Ergebnis enthält vier Tupel. Insgesamt ergibt sich eine Größe von $13 + 13 + 4 = 30$ Tupel in den Zwischenergebnissen.

Eine letzte Maßnahme, die aber mit Vorsicht anzuwenden ist, besteht im Verschieben bzw. Einfügen von Projektionen (Regeln 5, 6, 8 und 10). Die dadurch entstehende Reduzierung der Zwischenergebnisse hat zwei Ursachen: Einerseits können durch die Projektion Duplikate entstehen, die eliminiert werden können. Dieser Effekt tritt natürlich nicht auf, wenn in der Projektion noch ein Schlüssel enthalten ist. Andererseits wird die Größe der einzelnen Tupel reduziert, so dass bei einer Zwischenspeicherung des Ergebnisses auf dem Hintergrundspeicher weniger Seiten benötigt werden. Dieser geringere Platzbedarf äußert sich dann später in kleinerem Zugriffsaufwand bei der weiteren Bearbeitung, ein wichtiger Faktor bei sehr großen Attributen.

In unserem Beispiel kann durch Einfügen einer Projektion auf die Matrikelnummer – wie in Abbildung 8.8 gezeigt – ein Tupel eliminiert und viele für die weitere Bearbeitung unnötige Attribute entfernt werden. An dieser Stelle enthält das Zwischenergebnis alle Attribute aus *Professoren*, *Vorlesungen* und *hören*, aber lediglich das Attribut *MatrNr* wird gebraucht. Eine solche Maßnahme ist aber aufgrund der Duplikateliminierung, wie später noch besprochen wird, mit einigen Kosten verbunden. An dieser Stelle würde sie sich wahrscheinlich nicht lohnen. Folgendes soll als Entscheidungshilfe dienen: Ist der Wertebereich der zu projizierenden Attribute klein im Vergleich zu der Anzahl der Tupel (wie beispielsweise das Attribut *Semester* bei *Studenten*) oder können sehr große Attribute (z.B. Photos der Studenten in Form von Bitmaps) entfernt werden, lohnt sich eine Projektion.

Fassen wir nochmal die verwendeten Techniken zusammen und formulieren diese als Optimierungsheuristik, die auf der kanonischen Normalform aufsetzt:

1. Aufbrechen von Selektionen,
2. Verschieben der Selektionen soweit wie möglich nach unten im Operatorbaum (engl. pushing selections),
3. Zusammenfassen von Selektionen und Kreuzprodukten zu Joins,
4. Bestimmung der Reihenfolge der Joins in der Form, dass möglichst kleine Zwischenergebnisse entstehen,
5. unter Umständen Einfügen von Projektionen,
6. Verschieben der Projektionen soweit wie möglich nach unten im Operatorbaum.

8.1.3 Optimierung durch Entschachtelung von Unteranfragen

Wir haben uns in Kapitel 4.9 schon mit geschachtelten Unteranfragen auseinandergesetzt. Dort wurde auch diskutiert, wie man zwecks Optimierung diese Anfragen entschachteln kann – allerdings von Hand. Wir wollen jetzt zeigen, dass man das in modernen DBMS auch voll-automatisch durchführen kann.

Die folgende Anfrage ermittelt für alle Studenten ihre besten Prüfungen:

Q1:

```
select s.Name, p.VorlNr
from Studenten s, prüfen p
where s.MatrNr = p.MatrNr and p.Note = (
      select min(p2.Note)
      from prüfen p2
      where s.MatrNr=p2.MatrNr)
```

Konzeptuell wird für jedes *Studenten/prüfen*-Paar (s, p) in der Unteranfrage ermittelt, ob diese Prüfung p die beste Note hat, die diese/r Student/in jemals erzielt hat.

Hinsichtlich der Performanz ist diese korrelierte Unteranfrage natürlich nicht gut, da die Unteranfrage in der Form einer geschachtelten Schleife (nested loop) jeweils neu ausgewertet wird. Wir sprechen in diesem Zusammenhang auch von einem *abhängigen Join (dependent join)*, für den wir jetzt extra einen relationalen Algebra-Operator $\ltimes$ einführen werden.

Zunächst wollen wir aber eine entschachtelte SQL-Formulierung für diese Anfrage zeigen, die dem automatisch zu generierenden logischen Algebraausdruck entspricht:

Q1':

```
select s.Name, p.VorlNr
from Studenten s, prüfen p,
        (select p2.MatrNr as ID, min(p2.Note) as beste
         from prüfen p2
         group by p2.MatrNr) m
where s.MatrNr=p.MatrNr and m.ID=s.MatrNr and p.Note=m.beste
```

Hier ist die Unteranfrage nicht mehr abhängig von der äußeren Tupelvariablen s und die Anfrage kann deshalb mittels „normaler" Joinoperatoren ausgewertet werden. Für die Leistungsfähigkeit eines SQL-Datenbanksystems ist die vollautomatische Entschachtelung solcher Anfragen elementar. Wir werden hier zeigen, dass man dies durch Äquivalenzregeln der Relationenalgebra im Zuge der logischen Optimierung durchführen kann.

Anfragen mit korrelierten Unteranfragen werden zunächst in Algebraausdrücke mit den nachfolgend definierten abhängigen (dependent) Joins übersetzt, so dass die Unteranfrage für jedes Tupel der äußeren Anfrage evaluiert wird:

$$T_1 \bowtie_p T_2 \quad := \quad \{t_1 \circ t_2 | t_1 \in T_1 \wedge t_2 \in T_2(t_1) \wedge p(t_1 \circ t_2)\}.$$

Hierbei wird die rechte Seite für jedes Tupel der linken Seite separat evaluiert. Wir notieren die von einem Ausdruck T generierten Attribute als $\mathcal{A}(T)$, und die freien Variablen eines Ausdrucks T als $\mathcal{F}(T)$. Für die Auswertung des abhängigen Joins muss dann gelten: $\mathcal{F}(T_2) \subseteq \mathcal{A}(T_1)$. Also müssen alle Attribute, die von T_2 benötigt werden, von T_1 generiert werden.

Man kann die Definition des abhängigen Joins analog auch auf die anderen Joinoperatoren ausdehnen, also für Semi-Joins ($\ltimes$), Anti-Semijoins ($\triangleright$), und äußere Joins ($\bowtie$, $\bowtie$) werden die abhängigen Join-Varianten ($\ltimes$, $\triangleright$, $\bowtie$, ...) entsprechend so definiert, dass der rechte Ausdruck für jedes vom linken Ausdruck generierte Tupel separat und abhängig ausgewertet wird.

Neben den Joins verwenden wir hier auch noch den schon bekannten Gruppierungsoperator:

$$\Gamma_{A;a:f}(e) \quad := \quad \{x \circ (a : f(y)) | x \in \Pi_A(e) \wedge y = \{z | z \in e \wedge \forall a \in A : x.a = z.a\}\}$$

Er gruppiert die Eingabe e gemäß des Attrbuts (bzw. der Attributmenge) A und evaluiert eine oder mehrere Aggregatfunktionen, hier f und weist das Ergebnis dem neuen Attribut a zu.

Weiterhin verwenden wir noch die *map*-Funktion, die den Tupeln der Eingabe e ein neues berechnetes Attribut a hinzufügt:

$$\chi_{a:f}(e) \quad := \quad \{x \circ (a : f(x)) | x \in e\}.$$

Im Folgenden müssen manchmal Mengen von Attributen auf Gleichheit getestet werden. Dazu verwenden wir folgende Kurznotation:

$$t_1 =_A t_2 \quad := \quad \forall_{a \in A} : t_1.a = t_2.a.$$

Wie bereits beschrieben, wird eine korrelierte Unteranfrage initial in einen abhängigen Join übersetzt:

$$T_1 \bowtie_p T_2.$$

Dieser abhängige Join verursacht aufgrund der „Nested Loops"-Semantik quadratische Laufzeit. Deshalb sollte er im Zuge der logischen Optimierung möglichst

vollständig durch „normale" Join-Operatoren ersetzt werden. Dazu dienen die beiden nachfolgend beschriebenen Techniken. Zunächst wird die einfache Entschachtelung versucht, die dann Erfolg hat, wenn die korrelierte Unteranfrage nur aus syntaktischen Gründen formuliert wurde. In „schwierigeren" Fällen benötigt man aber die allgemeine Entschachtelungstechnik.

Einfache Entschachtelung

Eine einfache korrelierte Unteranfrage, die nur der syntaktischen Vereinfachung halber entstand, ist im folgenden Beispiel gezeigt:

Q2:
```
select s.*
from Studenten s
where exists (select * from prüfen p
                       where s.MatrNr = p.MatrNr)
```

Es werden hier die aktiven Studenten ermittelt, also diejenigen, die schon mindestens eine Prüfung absolviert haben.

Die Anfrage wird initial in folgenden Algebrausdruck übersetzt:

$$(Studenten\ s) \ltimes (\sigma_{s.MatrNr=p.MatrNr}(prüfen\ p))$$

Es ist einfach zu erkennen, dass man diesen Ausdruck äquivalenzerhaltend in nachfolgenden Anfrageplan mit einem regulären Semi-Join übersetzen kann:

$$(Studenten\ s) \ltimes_{s.MatrNr=p.MatrNr} (prüfen\ p)$$

Bei der einfachen Entschachtelung versucht man, alle abhängigen Prädikate so weit wie möglich nach oben zu transferieren. Dabei werden sie möglichst an Joins, Selektionen, Gruppierungen, etc. vorbei hochtransferiert, bis alle benötigten Attribute aus der Eingabe verfügbar sind. Wenn dies erreicht werden konnte, kann der abhängige Join in einen regulären Join umgewandelt werden – wie an unserem einfachen Beispiel demonstriert. Nach dieser Join-Umwandlung kann man dann wieder einige Prädikate nach unten transferieren, um durch frühzeitige Filterung eine bessere Selektivität zu erzielen.

Allgemeine Entschachtelung

Die gerade beschriebene Technik der Prädikat-Vertauschung führt bei vielen korrelierten Unteranfragen schon zum Erfolg, weshalb sie immer als erstes versucht werden sollte. Bei Erfolglosigkeit benötigt man aber ein allgemeineres Vorgehen, das die abhängigen Joins transformiert. Das wird an unserer Anfrage $Q1$ demonstriert, deren initiale Übersetzung in die Algebra in Abbildung 8.9 gezeigt ist. Zum Zweck der Optimierung wird der initiale abhängige Join gemäß folgender Äquivalenzregel aufgespalten:

$$T_1 \ltimes_p T_2 \equiv T_1 \ltimes_{p \wedge T_1 = \mathcal{A}(D)} D(D \ltimes T_2)$$

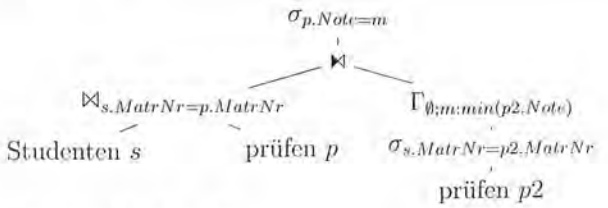


Abbildung 8.9: Original-Anfrage Q1

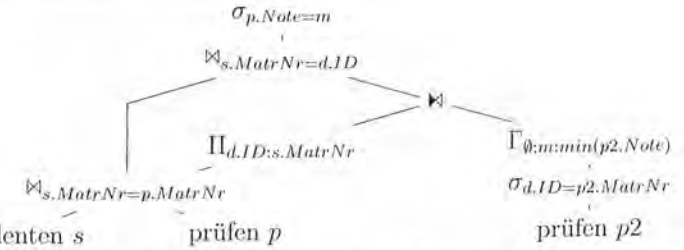


Abbildung 8.10: Seitwärts wird die duplikatfreie Domäne an den abhängigen Join transferiert.

wobei $D := \Pi_{\mathcal{F}(T_2)\cap\mathcal{A}(T_1)}(T_1)$.

Die Anwendung dieser Transformation für unsere Beispielanfrage ist in Abbildung 8.10 gezeigt. Auf den ersten Blick erscheint es wenig sinnvoll, einen abhängigen Join durch zwei Joins (einen abhängigen und einen regulären Join) zu ersetzen. Man beachte aber, dass der abhängige Join jetzt nicht mehr für Duplikate des linken Arguments ausgewertet werden muss. Dazu muss man natürlich die duplikateliminierende Projektion, und nicht die SQL-typische duplikaterhaltende Projektion realisieren. Es wird also die Domäne D aller möglichen Variablenbindungen berechnet und „seitwärts" an den abhängigen Join übermittelt, so dass jetzt für jede einzigartige Kombination der abhängige Join nur noch einmal ausgewertet wird. Bei Vorliegen vieler Duplikate resultiert das in einer großen Einsparung.

In unserem Beispiel werden jetzt die besten Noten nur noch einmal für jeden Studenten berechnet – anstatt für jedes Studenten/prüfen-Paar.

Noch wichtiger ist, dass der abhängige Join jetzt nur auf einer echten (duplikatfreien) Menge D ausgewertet wird, was weitere Transformationen des abhängigen Joins ermöglicht. In den folgenden Transformationsregeln setzen wir demnach immer voraus, dass die linke Seite D duplikatfrei ist.

Letztendlich wollen wir den abhängigen Join soweit nach unten drücken bis die rechte Seite gar nicht mehr abhängig von der linken Seite ist. Dann kann man den abhängigen Join nämlich wie folgt gänzlich entfernen:

$$D \bowtie\!\!\!\!\!\!\cdot\; T \equiv D \bowtie T \quad \text{falls} \quad \mathcal{F}(T) \cap \mathcal{A}(D) = \emptyset.$$

Jetzt muss man zwar immer noch einen Join berechnen, aber keinen abhängigen Join. Oft kann man sogar diesen Join noch entfernen, wie wir an unserem Beispiel noch sehen werden.

Nachdem wir den Startpunkt und das Ziel der Optimierung skizziert haben, wollen wir jetzt einige Transformationsregeln angeben, die den Weg zum Ziel schrittweise ermöglichen.

Abhängige Joins können leicht an Selektionen vorbei transferiert werden:

$$D \rtimes \sigma_p(T_2) \equiv \sigma_p(D \rtimes T_2).$$

Diese Transformation erscheint kontraproduktiv, da man normalerweise Selektionen nach unten drücken will. Dies kann man nach der Entschachtelungsoptimierung auch wieder machen. Wir transferieren aber die abhängigen Joins erst mal soweit nach unten bis man sie vollständig eliminieren oder durch reguläre Joins ersetzen kann. Dazu dienen die nachfolgenden Regeln für die Vertauschung von abhängigen mit regulären Joins:

$$D \rtimes (T_1 \bowtie_p T_2) \equiv \begin{cases} (D \rtimes T_1) \bowtie_p T_2 & : \mathcal{F}(T_2) \cap \mathcal{A}(D) = \emptyset \\ T_1 \bowtie_p (D \rtimes T_2) & : \mathcal{F}(T_1) \cap \mathcal{A}(D) = \emptyset \\ (D \rtimes T_1) \bowtie_{p \wedge \text{natural join } D} (D \rtimes T_2) & : \text{andernfalls.} \end{cases}$$

Wenn die vom abhängigen Join produzierten Werte nur auf einer Seite benötigt werden, wird er dorthin bewegt; anderenfalls muss man ihn duplizieren. Oftmals kann man die Ausdrücke unterhalb des Joins noch weiter optimieren – wie wir auch an unserem Beispiel noch sehen werden. Man beachte dass die Replikation des abhängigen Joins auf beiden Seiten noch verlangt, dass wir die Werte gemäß gleicher D-Attributwerte wieder zusammenführen, was mit dem „*natural join D*" notiert wurde.

Wenn man einen abhängigen Join an einer Gruppierung vorbeidrücken will, muss man darauf achten, dass die Gruppierung alle Attribute die der abhängige Join produziert, beibehält, wie in nachfolgender Äquivalenzregel:

$$D \rtimes (\Gamma_{A;a:f}(T)) \equiv \Gamma_{A \cup \mathcal{A}(D);a:f}(D \rtimes T)$$

Wiederum basiert die Korrektheit der Regel darauf, dass D eine duplikatfreie Menge ist.

Projektionen kann man ähnlich zur Gruppierung behandeln:

$$D \rtimes (\Pi_A(T)) \equiv \Pi_{A \cup \mathcal{A}(D)}(D \rtimes T)$$

Die noch fehlenden Mengenoperatoren werden mittels der nachfolgenden Äquivalenzen behandelt:

$$\begin{aligned} D \rtimes (T_1 \cup T_2) &\equiv (D \rtimes T_1) \cup (D \rtimes T_2) \\ D \rtimes (T_1 \cap T_2) &\equiv (D \rtimes T_1) \cap (D \rtimes T_2) \\ D \rtimes (T_1 \setminus T_2) &\equiv (D \rtimes T_1) \setminus (D \rtimes T_2) \end{aligned}$$

Wir zeigen in den Abbildungen 8.11, 8.12 und 8.13 die weiteren Schritte zur Optimierung der Beispielanfrage, die letztendlich zu der vollständigen Eliminierung des abhängigen Joins führen. Die Leser mögen sich vergewissern, dass der optimierte Algebraplan der hand-optimierten Anfrage Q1' entspricht.

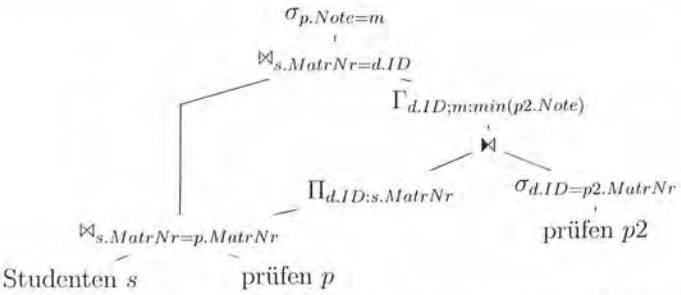


Abbildung 8.11: Vertauschen von Gruppierung/Aggregation mit dem abhängigen Join

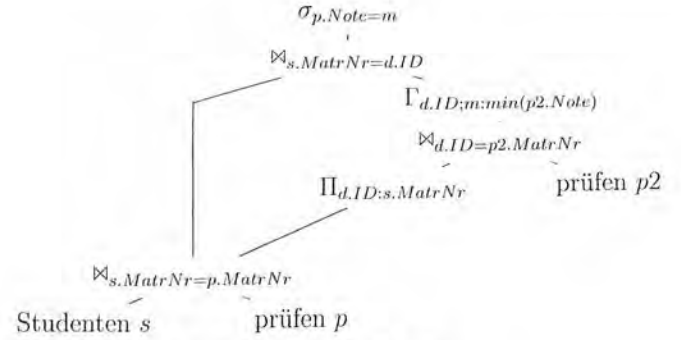


Abbildung 8.12: Elimination des abhängigen Joins

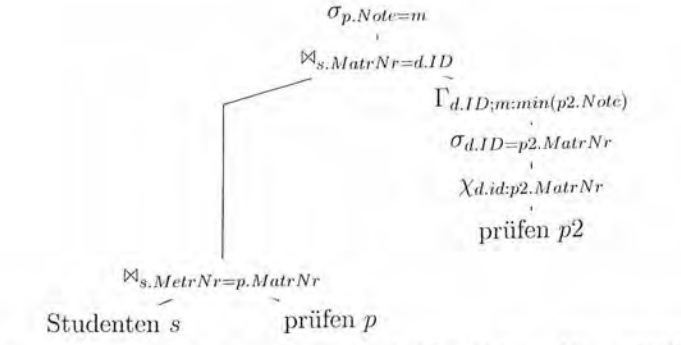


Abbildung 8.13: Optional: Entkopplung der beiden Seiten. Die „seitwärts" transferierte Domäne $d.ID$ ist eine Teilmenge der in $p2$ enthaltenen *MatrNr*-Werte und kann deshalb eliminiert werden. Der Join wird dadurch in eine (eigentlich unnötige) Selektion transformiert.

8.2 Physische Optimierung

Man unterscheidet die logischen Algebraoperatoren von den physischen Algebraoperatoren, die die Realisierung der logischen Operatoren darstellen. Es kann für einen logischen Operator durchaus mehrere physische Operatoren geben.

Bisher haben wir uns nur auf der logischen Ebene bewegt. In diesem Abschnitt wird der physische Aufbau der Datenbank, also z.B. Indices oder Sortierung von Relationen, bei der Auswahl von Implementierungen für die logischen Operatoren ausgenutzt.

Eine elegante Lösung um Auswertungspläne baukastenartig zusammenzusetzen, stellen die sogenannten *Iteratoren* dar. Ein Iterator ist ein abstrakter Datentyp, der Operationen wie **open**, **next**, **close**, **cost** und **size** als Schnittstelle zur Verfügung stellt.

Die Operation **open** ist eine Art Konstruktor, der die Eingaben öffnet und eventuell eine Initialisierung vornimmt. Die Schnittstellenoperation **next** liefert das nächste Tupel des Ergebnisses der Berechnung. Die Operation **close** schließt die Eingaben und gibt möglicherweise noch belegte Ressourcen frei. Diese drei Funktionen sind vergleichbar mit denen eines Cursors in Embedded SQL (vergleiche Kapitel 4).

Die beiden Operationen **cost** und **size** geben Informationen über die geschätzten Kosten für die Berechnung und die Größe des Ergebnisses an. Halbwegs realistische Kostenmodelle sind leider sehr umfangreich und kompliziert. Wir werden daher auf eine vollständige Diskussion verzichten und in Abschnitt 8.3 nur einige Varianten angeben.

Genau wie die relationalen Operatoren einer Anfrage baumartig dargestellt werden können, ist dies auch mit Iteratoren möglich. Mehrere Iteratoren werden so zu einem Auswertungsplan kombiniert. Schematisch kann man sich das Zusammensetzen von Iteratoren wie in Abbildung 8.14 vorstellen.

Das Anwendungsprogramm (oder die Benutzerschnittstelle bei einer interaktiven Anfrage) öffnet den Wurzeliterator und fordert mit Hilfe des **next**-Befehls solange Ergebnisse an, bis keine mehr geliefert werden können. Der Wurzeliterator benötigt für die Berechnung der Ergebnistupel die Ausgaben der mit ihm verbundenen Tochteriteratoren. Daher ruft er bei den Tochteriteratoren wieder entsprechend **open**, **next** und **close** auf. Der Prozess setzt sich so bis zu den Blättern fort, an denen die Basisrelationen der Datenbank stehen.

Zusätzlich zu einer eleganteren Architektur bietet das Iteratorkonzept den Vorteil, dass man nicht notwendigerweise Zwischenergebnisse speichern muss. Nehmen wir beispielsweise an, dass eine Anfrage ausschließlich Selektionen und Projektionen enthält. Ständen nur Prozeduren zur Verfügung, die jeweils einen algebraischen Operator komplett berechnen, müsste im Allgemeinen für jedes Teilergebnis eine Zwischenspeicherung stattfinden. Bei der schrittweisen Realisierung werden Ergebnisse Stück für Stück durchgereicht. Man spricht hier auch von *Pipelining*. Ein Nachteil des Iteratorkonzepts ist, dass die Realisierung der unterschiedlichen Iteratoren komplizierter ist als eine entsprechende Realisierung durch Prozeduren.

Für die Diskussion der Funktionsweise der einzelnen Iteratortypen wollen wir sie in fünf Gruppen unterteilen:[1]

[1] Die Umbenennung dient in der relationalen Algebra der eindeutigen Identifizierung von At-

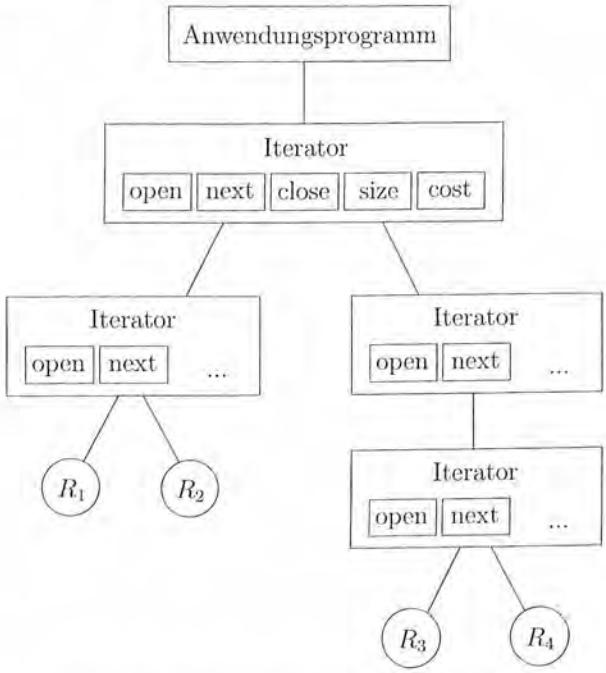


Abbildung 8.14: Schematische Darstellung eines Auswertungsplanes

1. Selektion
2. Binäre Zuordnung (Matching)
3. Gruppierung und Duplikateliminierung
4. Projektion und Vereinigung
5. Zwischenspeicherung

Im Allgemeinen gibt es drei prinzipielle Methoden, die Operatoren der ersten drei Gruppen zu implementieren. Zunächst wäre da der „Brute Force"-Ansatz, bei dem einfach sequentiell alle Möglichkeiten durchgetestet werden. Der zweite Weg besteht in der Ausnutzung der Reihenfolge bzw. Sortierung der Tupel. Als dritte Möglichkeit können Indexstrukturen ausgenutzt werden, um direkt auf bestimmte Tupel zuzugreifen.

8.2.1 Implementierung der Selektion

Abbildung 8.15 zeigt eine Implementierung der Selektion in der „Brute Force"-Variante und über den Zugriff auf eine Indexstruktur (sei es ein B-Baum oder eine

tributen. In der physischen Algebra spielt sie keine Rolle, da solche Schema-Informationen im Allgemeinen über interne und somit eindeutige Bezeichner gehandhabt werden.

a) **iterator** Select_p

open
- Öffne Eingabe

next
- Hole solange nächstes Tupel, bis eines die Bedingung p erfüllt, ansonsten ist man fertig
- Gib dieses Tupel zurück

close
- Schließe Eingabe

b) **iterator** IndexSelect_p

open
- Schlage im Index die erste Stelle nach, an der ein Tupel die Bedingung erfüllt

next
- Gib nächstes Tupel zurück, falls es die Bedingung p noch erfüllt

close
- Schließe Eingabe

Abbildung 8.15: Zwei Implementierungen der Selektion: a) ohne und b) mit Indexunterstützung

Hashtabelle). Bei jedem Aufruf von **next** wird ein die Bedingung erfüllendes Tupel zurückgeliefert, bis die Eingabequellen erschöpft sind. In der Index-Variante wird beim Öffnen des Iterators zusätzlich schon das erste passende Tupel nachgeschlagen. Bei einem B^+-Baum beispielsweise geschieht das durch Absteigen innerhalb des Baums bis zu den Blättern. Die Blätter können dann bei jedem **next**-Aufruf sequentiell durchsucht werden, bis die Bedingung nicht mehr zutrifft.

8.2.2 Implementierung von binären Zuordnungsoperatoren

Join, Mengendifferenz und Mengendurchschnitt lassen sich auf sehr ähnliche Weise implementieren. Daher fasst man sie unter der Bezeichnung *binäre Zuordnungsoperatoren* zusammen. Beim Join werden Attribute zweier Tupel verglichen, bei Differenz und Schnitt komplette Tupel. In diesem Abschnitt werden nur Implementierungen von Equijoins vorgestellt (siehe dazu auch Übungsaufgabe 8.6).

Ein einfacher Join-Algorithmus

Es liegt am nächsten, zwei ineinander geschachtelte Schleifen (engl. nested loops) zu verwenden. Dabei wird jedes Tupel der einen Menge mit jedem der anderen verglichen. In vereinfachter Form als normale Prozedur formuliert sähe der Join $R \bowtie_{R.A=S.B} S$ so aus:

for each $r \in R$
  for each $s \in S$
    if $r.A = s.B$ **then**
      $res := res \cup (r \times s)$

iterator NestedLoop$_p$

open

- Öffne die linke Eingabe

next

- Rechte Eingabe geschlossen?
  - Öffne sie
- Fordere rechts solange Tupel an, bis Bedingung p erfüllt ist
- Sollte zwischendurch rechte Eingabe erschöpft sein
  - Schließe rechte Eingabe
  - Fordere nächstes Tupel der linken Eingabe an
  - Starte **next** neu
- Gib den Verbund von aktuellem linken und aktuellem rechten Tupel zurück

close

- Schließe beide Eingabequellen

Abbildung 8.16: Nested-Loop Iterator

Hierbei wird das Ergebnis *res* sukzessive mit (r,s)-Kombinationen gefüllt, bei denen die Joinbedingung erfüllt ist.

Die Iteratorformulierung in Abbildung 8.16 ist schon etwas komplizierter, da bei jedem Aufruf der Funktion **next** ja nur ein Tupel weitergereicht wird. Dieser Pseudocode ist stark vereinfacht; reale Implementierungen berücksichtigen auch Fragen der Verteilung der Tupel auf Hintergrundspeicherseiten und der Systempufferverwaltung.

Ein verfeinerter Join-Algorithmus

Die Tupel einer Relation sind auf Seiten abgespeichert und müssen dementsprechend für eine Bearbeitung seitenweise vom Hintergrundspeicher in den Hauptspeicher geladen werden.

Stehen im Hauptspeicher m Seiten für die Berechnung des Joins zur Verfügung, reserviert der verfeinerte Join-Algorithmus k Seiten für die innere Schleife und $m-k$ für die äußere. Die äußere Relation, nennen wir sie R, wird in Portionen zu $m-k$ Seiten eingelesen. Für jede dieser Portionen muss die komplette innere Relation S in Portionen zu k Seiten eingelesen werden. Alle Tupel der Relation R, die sich auf den $m-k$ Seiten befinden, werden mit allen Tupeln aus S in den k Seiten verglichen.

Man kann pro Durchlauf das Einlesen einer Portion von k Seiten sparen, wenn die innere Relation im Zick-Zack-Verfahren durchlaufen wird, also abwechselnd vorwärts und rückwärts. Das ist in Abbildung 8.17 skizziert. Der optimale Fall tritt bei $k = 1$ und Verwendung der kleineren Relation als äußeres Argument – hier also R – ein, wie wir in Abschnitt 8.3.3 sehen werden.

Im Allgemeinen ist eine Nested-Loop Auswertung mit ihrem quadratischen Aufwand zu teuer. Sie hat jedoch den Vorteil, dass sie sehr einfach ist und ohne wesentliche Modifikation auch andere Joinformen (Theta-Joins und Semi-Joins) berechnen kann.

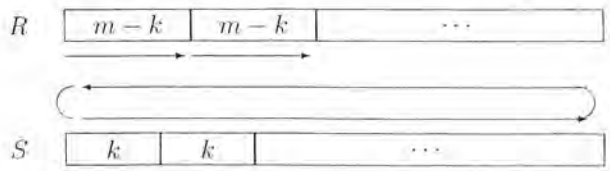


Abbildung 8.17: Schematische Darstellung des seitenorientierten Nested-Loop Joins

R	
	A
...	0
...	7
...	7
...	8
...	8
...	10
⋮	⋮

z_r ← → z_s

S	
B	
5	...
6	...
7	...
8	...
8	...
11	...
⋮	⋮

Abbildung 8.18: Beispiel einer Merge-Join Ausführung

Ausnutzung der Sortierung

Falls eine Sortierung beider Eingaben nach den zu verbindenden Attributen vorliegt, kann eine wesentlich effizientere Methode gewählt werden: der sogenannte *Merge-Join*. Dabei werden beide Relationen parallel von oben nach unten abgearbeitet. An jeder Position innerhalb der Relationen ist bekannt, dass kein Tupel mit einem kleineren Wert im Joinattribut mehr folgt. Wenn also ein potentieller (Equi-) Joinpartner des gerade aktuellen Tupels schon größer ist, braucht das aktuelle Tupel nicht mehr betrachtet zu werden.

In Abbildung 8.18 soll $R \bowtie_{R.A=S.B} S$ berechnet werden. Beim Öffnen der Eingaben (hier R und S) wird je ein Zeiger auf das erste Tupel von R und S positioniert – hier z_r und z_s genannt. Wir beginnen mit dem kleinsten Attributwert in der Eingabe, hier der 0. Die andere Eingabe besitzt den Wert 5 im Joinattribut. Wir wissen daher aufgrund der Sortierung, dass kein Joinpartner für das Tupel mit der 0 existiert und bewegen z_r vorwärts auf die 7. Nun ist die 5 der kleinste Wert in der Eingabe und wir bewegen z_s vorwärts. Nach zwei Schritten erreicht z_s die 7, und ein Joinpartner ist gefunden. Der Join wird durchgeführt und in R nach weiteren potentiellen Joinpartnern gesucht. Es existiert noch ein weiteres Tupel mit Attributwert 7, daher kann noch ein zweiter Join durchgeführt werden. Dieser Prozess kann jetzt fortgesetzt werden, bis beide Tabellen durchlaufen sind.

Eines muss allerdings noch beachtet werden: Sobald beim Durchlauf ein erster Joinpartner gefunden wird, muss er markiert werden. Existieren nämlich auf beiden Seiten mehrere Tupel mit gleichem Attributwert, muss nach einem Durchlauf auf einer Seite der Zeiger wieder auf die Markierung zurückgesetzt werden. Dies ist bei der 8 der Fall, bei der vier Ergebnistupel erzeugt werden müssen. Der Leser möge

iterator MergeJoin$_p$

open
- Öffne beide Eingaben
- Setze *akt* auf linke Eingabe
- Markiere rechte Eingabe

next
- Solange Bedingung p nicht erfüllt
  - Setze *akt* auf Eingabe mit dem kleinsten anliegenden Wert im Joinattribut
  - Rufe **next** auf *akt* auf
  - Markiere andere Eingabe
- Gib Verbund der aktuellen Tupel der linken und rechten Eingabe zurück
- Bewege andere Eingabe vor
- Ist Bedingung nicht mehr erfüllt oder andere Eingabe erschöpft?
  - Rufe **next** auf *akt* auf
  - Wert des Joinattributs in *akt* verändert?
    - Nein, dann setze andere Eingabe auf Markierung zurück
    - Ansonsten markiere andere Eingabe

close
- Schließe beide Eingabequellen

Abbildung 8.19: Merge-Join Iterator

das Beispiel mit Hilfe der Iterator-Darstellung in Abbildung 8.19 nachvollziehen.

Im Durchschnitt kann man bei diesem Algorithmus mit linearem Aufwand rechnen – falls die Sortierung gegeben ist. Im schlechtesten Fall kann er natürlich auch quadratisch werden, wenn der Join zu einem Kreuzprodukt entartet. Dies wäre bei der Situation $\Pi_A(R) = \{c\} = \Pi_B(S)$ gegeben, wenn also sowohl im Attribut A von R als auch im Attribut B von S nur gleiche Werte, nämlich c, vorkommen.

Bei nicht vorhandener Sortierung muss diese natürlich vorher durchgeführt werden, um den Merge-Join anwenden zu können. Man bezeichnet diese Variante oft als *Sort/Merge-Join*.

Ausnutzung von Indexstrukturen

Ein weiteres Verfahren besteht in der Ausnutzung eines Indexes auf einem der Joinattribute. Das ist in Abbildung 8.20 demonstriert. Auf das Attribut B der Relation S ist ein Index angelegt. Daher braucht man für jedes Tupel aus R nur die passenden Tupel aus B im Index nachzuschlagen. Die Iterator-Darstellung ist in Abbildung 8.21 angegeben. Auch hier muss berücksichtigt werden, dass unter Umständen zu einem Attributwert mehrere Tupel im Index eingetragen sind.

Für den B$^+$-Baum wäre das Nachschlagen des Joinattributwerts im Index also gleichbedeutend mit dem Absteigen des Baums zu der Stelle in den Blättern, an der der Wert das erste Mal vorkommt. Weitere Tupel mit dem Joinattributwert findet man im B$^+$-Baum, indem man den Blattknoten „nach rechts“ durchsucht und gegebenenfalls die Verkettungen zu anderen Blattknoten verfolgt.

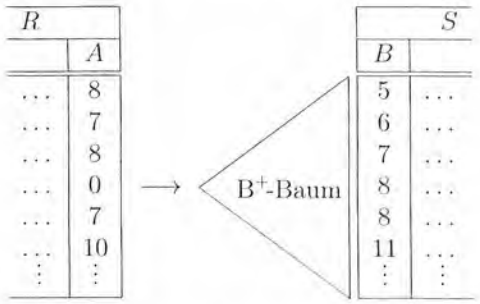


Abbildung 8.20: Schematische Darstellung eines Index-Joins

iterator IndexJoin_p

open

- Sei Index auf Joinattribut der rechten Eingabe vorhanden
- Öffne die linke Eingabe
- Hole erstes Tupel aus linker Eingabe
- Schlage Joinattributwert im Index nach

next

- Bilde Join, falls Index ein (weiteres) Tupel zu diesem Attributwert liefert
- Ansonsten bewege linke Eingabe vor und schlage Joinattributwert im Index nach

close

- Schließe die Eingabe

Abbildung 8.21: Index-Join Iterator

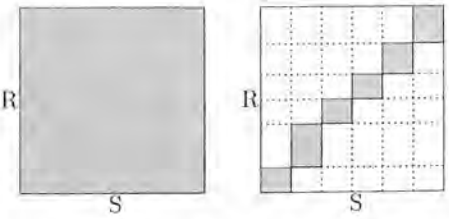


Abbildung 8.22: Effekt der Partitionierung (angelehnt an Mishra und Eich (1992))

Hash-Joins

Ein einfacher Index-Join, wie er oben vorgestellt ist, hat verschiedene Nachteile:

- Manchmal sind die Eingaben eines Joins Zwischenergebnisse anderer Berechnungen, für die keine Indexstrukturen existieren.
- Das Anlegen von temporären Hashtabellen oder B-Bäumen für die Berechnung einer Anfrage lohnt sich allgemein nur, wenn die Indexstruktur im Hauptspeicher Platz findet.
- Man geht bei einer größeren Hashtabelle als Indexstruktur davon aus, dass jedes Nachschlagen aufgrund der nicht vorhandenen Ballung (siehe Kapitel 7) mindestens einen Seitenzugriff erfordert. Daher ist die Ausnutzung einer Hashtabelle bei einem normalen Index-Join nur sinnvoll, wenn die nicht-indizierte Relation klein ist.

Die Idee des Hash-Joins besteht darin, die Eingabedaten so zu partitionieren, dass die Verwendung einer Hauptspeicher-Hashtabelle möglich ist. Die Wirkung der Partitionierung kann man sich mit Hilfe von Abbildung 8.22 verdeutlichen. Beim Nested-Loop Join muss jedes Element der Argumentrelation R mit jedem Element der Relation S verglichen werden, was einer vollständigen Schraffierung der Fläche im Bild entspricht (linke Abbildung). Mit der Partitionierung werden vorher die Tupel der Argumentrelationen so gruppiert, dass nur die schraffierten Rechtecke in der Diagonalen berücksichtigt werden müssen (rechte Abbildung). Die Vorgehensweise dazu ist in Abbildung 8.23 dargestellt.

Die kleinere der beiden Argumentrelationen wird zum sogenannten *Build Input*: Sie wird solange partitioniert, bis die Partitionen in den Hauptspeicher passen.

Stehen für einen Partitionierungsvorgang m Hauptspeicherseiten zur Verfügung, werden $m-1$ für die Ausgabe und eine für die Eingabe reserviert. Die Hashfunktionen h_i werden so gewählt, dass sie die Eingabe auf die $m-1$ Ausgabeseiten abbilden. Es wird jeweils eine Seite gelesen und mit der Hashfunktion h_i auf die restlichen $m-1$ Seiten verteilt. Läuft eine der Ausgabeseiten über, wird sie in die zugehörige Partition geschrieben. Am Ende werden alle verbleibenden Seiten in ihre Partitionen geschrieben. So entstehen in jedem Schritt aus jeder Partition rekursiv $m-1$ kleinere Partitionen. In der Abbildung ist dies für $m-1=3$ gezeigt.

Als nächstes wird die größere Argumentrelation bearbeitet, der sogenannte *Probe Input*. Sie wird mit den gleichen Hashfunktionen h_i partitioniert wie der Build Input.

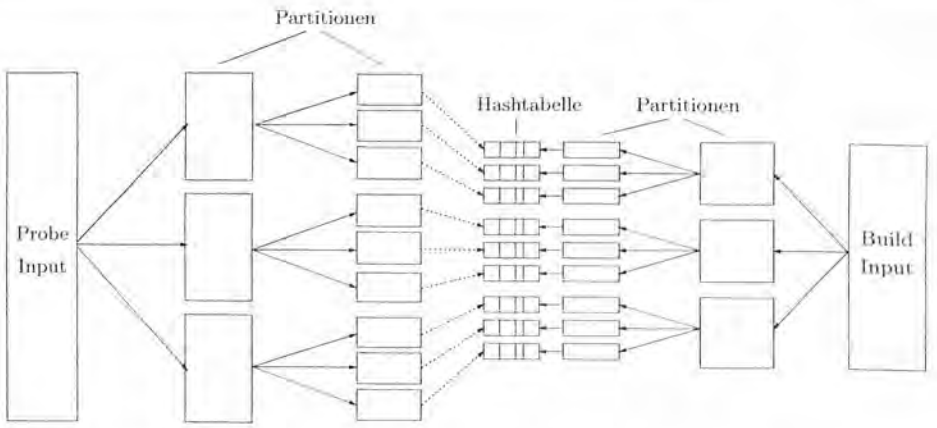


Abbildung 8.23: Partitionierung von Relationen mit einer Hash-Funktion

Die sich ergebenden Partitionen brauchen jedoch nicht unbedingt in den Hauptspeicher zu passen.

Nach der Partitionierungsphase befinden sich die potentiellen Joinpartner, anschaulich ausgedrückt, in „gegenüberliegenden" Partitionen. Jetzt wird immer jeweils eine Partition des Build Inputs in den Hauptspeicher gelesen und dort als normale Hauptspeicher-Hashtabelle organisiert. Die entsprechende Partition des Probe Inputs kann nun Seite für Seite eingelesen werden. Mit Hilfe der Hashtabelle sind alle potentiellen Joinpartner im Hauptspeicher schnell zu finden. Dieser abschließende Bearbeitungsschritt ist in Abbildung 8.23 durch gestrichelte Pfeile dargestellt.

Verdeutlichen wir das noch einmal an dem etwas konkreteren Beispiel in Abbildung 8.24. Dort wird ein Join von gleichaltrigen Männern und Frauen durchgeführt. Die Relation *Frauen* ist etwas größer, daher wird sie als Probe Input verwendet. Der Anschaulichkeit halber sei eine Hash-Funktion gewählt, die nach dem Alter ordnet – in der Praxis wäre das sicher nicht sinnvoll (siehe Aufgabe 8.4). So sind nach der ersten Partitionierung des Build Inputs die $20-39$ Jahre alten Männer in der ersten Partition, die $40-59$ Jahre alten in der zweiten Partition usw. Im zweiten Partitionierungsschritt werden diese Partitionen weiter zerlegt. Danach seien alle Partitionen klein genug, um in den Hauptspeicher zu passen. Das Gleiche wird für die Frauen durchgeführt.

Jetzt kann die Partition mit den $20-26$ Jahre alten Männern in den Hauptspeicher geladen werden, um sie auf die Hashtabelle zu verteilen. Auf der anderen Seite wird die Partition mit den $20-26$ Jahre alten Frauen durchgegangen, und es werden die entsprechenden Joinpartner gesucht, die sich ja im Hauptspeicher befinden müssen.

Man beachte, dass ein *HashJoin*-Iterator, den wir hier nicht mehr detaillierter spezifizieren, beim **open** schon einen Großteil der „Arbeit verrichtet". Bei der Initialisierung (Aufruf von **open**) wird schon die gesamte Partitionierung und der Aufbau der Hashtabelle für die erste Partition des Build-Inputs durchgeführt. Erst danach kann dieser Iterator sukzessive (durch Aufruf von **next**) Ergebnistupel des Joins liefern.

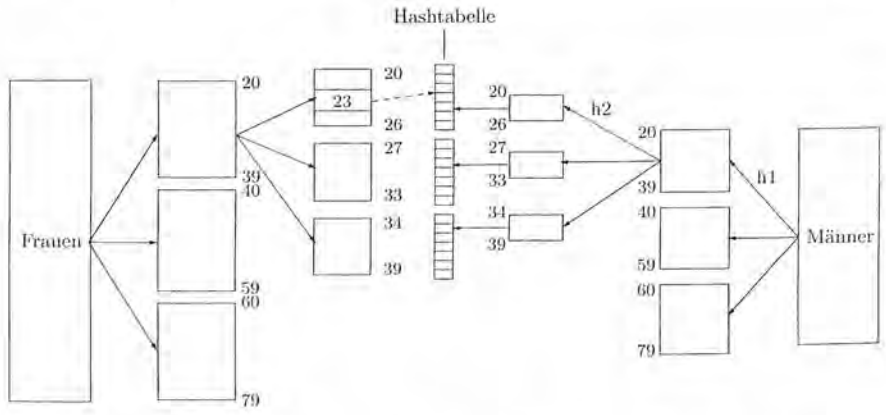


Abbildung 8.24: Berechnung von Frauen $\bowtie_{\text{Frauen.Alter=Männer.Alter}}$ Männer

8.2.3 Gruppierung und Duplikateliminierung

Auch Gruppierung und Duplikateliminierung sind miteinander verwandt. Ihre Gemeinsamkeit ist ähnlich der von Join und Differenz bzw. Schnitt. Während bei der Gruppierung Tupel zusammengefasst werden, bei denen ein bestimmtes Attribut im Wert übereinstimmt, werden bei der Duplikateliminierung diejenigen Tupel zusammengefasst, die vollständig übereinstimmen.

Es können wieder die schon im vorigen Abschnitt benutzten drei Methoden eingesetzt werden (daher geben wir hier auch nicht mehr die Iterator-Schreibweise an). Die Brute-Force Methode vergleicht einfach analog zum Nested-Loop Join in einer geschachtelten Schleife jedes Tupel mit jedem. Liegt eine Sortierung vor, braucht die Eingabe lediglich von Anfang bis Ende einmal durchsucht und alle Duplikate eliminiert bzw. Gruppen bearbeitet zu werden. Alternativ kann ein vorliegender Sekundärindex ausgenutzt werden. Wurde beispielsweise ein B^+-Baum verwendet, befinden sich in den Blättern des Baums entweder die Tupel oder Zeiger auf die Tupel der Eingaberelation in sortierter Reihenfolge. Im Normalfall ist es sinnvoll, für die Duplikateliminierung eine Partitionierung ähnlich wie beim Hash-Join oder eine Sortierung durchzuführen, wenn diese nicht vorliegt.

8.2.4 Projektion und Vereinigung

Projektion und Vereinigung sind sehr einfach zu implementieren, es wird daher nur kurz auf das Vorgehen eingegangen.

Da der Projektionsoperator der physischen Algebra keine Duplikateliminierung vornimmt (dafür ist ja ein spezieller Operator vorgesehen), braucht er lediglich jedes Tupel der Eingabe auf die entsprechenden Attribute zu reduzieren und an die Ausgabe weiterzureichen.

Bei der Vereinigung werden nur nacheinander alle Tupel der linken und rechten Eingabe ausgegeben, da auch hier in der physischen Algebra keine automatische Duplikateliminierung durchgeführt wird.

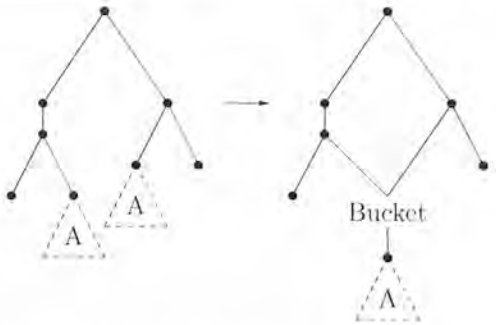


Abbildung 8.25: Eliminierung gemeinsamer Teilausdrücke

8.2.5 Zwischenspeicherung

Es ist durchaus möglich, Iteratoren so zu verwenden, dass zwischendurch kein einziges Tupel auf dem Hintergrundspeicher abgelegt werden muss. Sie brauchen lediglich einzeln nach oben weitergegeben zu werden. Das ist aber sicherlich nicht immer der effizienteste Weg. Besteht die innere Schleife eines Nested-Loop-Joins beispielsweise wieder aus einem Nested-Loop-Join, müsste für jedes Tupel des äußeren Arguments der innere Join komplett neu berechnet werden. In diesem Fall ist es vielfach effizienter einen Operator zur Zwischenspeicherung einzufügen, den wir *Bucket* nennen wollen.[2]

Der Bucket-Operator legt einfach alle Tupel der Eingabe temporär auf dem Hintergrundspeicher ab, quasi als eine Art „Auffangbecken". Bei später folgenden Durchläufen kann er dann auf diese Daten zugreifen.

Eine weitere Anwendungsmöglichkeit gibt es bei der Eliminierung (bzw. Faktorisierung) gemeinsamer Teilausdrücke. Sollte in einer Anfrage ein Ausdruck mehrmals vorkommen, ist es oft sinnvoll, ihn nur einmal auszuwerten und zwischenzuspeichern. Der Auswertungsplan wird dann zu einem Graph, wie es Abbildung 8.25 skizziert.

Eine Verfeinerung des Bucket-Operators sind die Operatoren *Sort*, *Hash* und *BTree*. Auch sie führen eine Zwischenspeicherung durch, bearbeiten die Eingabe jedoch vorher. Im ersten Fall wird sie sortiert, im zweiten und dritten wird eine Hashtabelle bzw. ein B-Baum mit der Eingabe als Inhalt angelegt. Dadurch ist es möglich, die effizienteren Sort- und Index-Algorithmen auch auf Zwischenergebnissen zu verwenden.

8.2.6 Sortierung von Zwischenergebnissen

Hier soll eine einfache Version des üblicherweise verwendeten Mergesorts vorgestellt werden. Das Problem bei der Sortierung ist wieder, dass Relationen im Allgemeinen wesentlich größer sind als der Hauptspeicher. Sie können also immer jeweils nur teilweise bearbeitet werden; Verfahren wie Quicksort sind daher nicht anwendbar.

[2] Aufgrund des hohen Speicherverbrauchs einer Joinberechnung ist es meistens auch notwendig, die Ausgabe abzuspeichern. Ansonsten kann bei mehreren Joinberechnungen jeder einzelnen nicht genügend Hauptspeicherplatz zugeteilt werden.

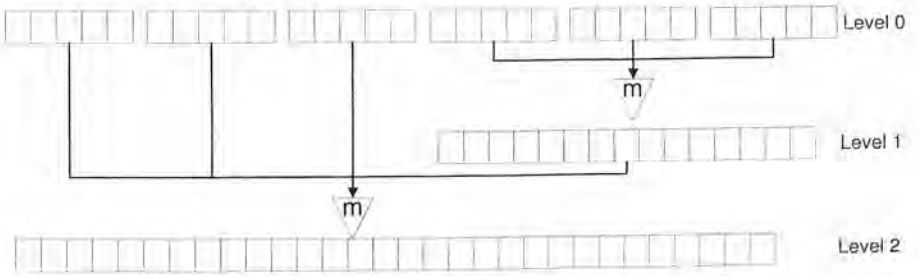


Abbildung 8.26: Demonstration eines einfachen Mergesorts

Die Idee des Mergesorts ist es, eine Relation in sortierte Stücke zu zerteilen, sogenannte *Läufe* (engl. *runs*). Zwei (oder mehrere) sortierte Läufe können dann zu einem größeren sortierten Lauf, ähnlich wie beim Merge-Join, zusammengemischt werden. Das wird solange fortgesetzt, bis nur noch ein Lauf vorhanden ist.

Die initialen, sogenannten *Level-0 Läufe* werden durch eine Hauptspeichersortierung, beispielsweise mit Hilfe von Quicksort, gebildet. Dazu wird die Relation stückweise eingelesen, sortiert und in eine temporäre Relation zurückgeschrieben.

Danach beginnt der Mischvorgang. Stehen m Seiten im Hauptspeicher zur Verfügung, werden $m-1$ Läufe (ähnlich wie beim Merge-Join) gemischt. Die freie Seite wird für die Ausgabe benötigt.

Sei b_R die Anzahl der Seiten, die die Relation R belegt. Betrachten wir als Beispiel für die Situation $m = 5$ und $b_R = 30$ die Abbildung 8.26. Nach der initialen Sortierung entstehen sechs Level-0 Läufe der Länge m. Jeder Mischvorgang kann maximal $m - 1 = 4$ Seiten lesen und sie in die verbleibende fünfte Seite mischen. Also muss man von den 6 Level-0 Läufen drei vorab mischen, damit man am Ende die verbleibenden drei Level-0 Läufe und den einen Level-1 Lauf auf einmal mischen kann. Man sollte in der Zwischenphase so wenige Läufe wie nötig mischen, da jeder Mischvorgang einen „Round-Trip" zur Platte bedeutet. Es ist relativ einfach, einen optimalen Algorithmus zu konzipieren, der eine minimale Anzahl von I/O-Vorgängen garantiert – siehe Aufgabe 8.8.

Ein einzelner Mischvorgang wird in Abbildung 8.27 gezeigt. Im Hauptspeicher M befindet sich je eine Seite jedes Laufs und die Ausgabeseite. Nun wird immer der kleinste der anliegenden Werte, hier die 1, in die Ausgabeseite geschrieben. Ist die Ausgabeseite voll, wird sie fortgeschrieben. Ist eine der Eingabeseiten leer, wird sie aus dem entsprechenden Lauf aufgefüllt.

Um die Anzahl der Durchgänge zu reduzieren, sollten die initialen Läufe so lang wie möglich sein. Eine Verbesserung lässt sich mit einer sogenannten *Replacement-Selection*-Strategie erreichen.

Bei der Bildung eines Level-0 Laufs werden die Daten dabei nicht direkt vollständig wieder zurückgeschrieben, sondern nur stückweise. Jedesmal, wenn wieder Platz frei wird, wird dieser mit neuen Elementen aus der Eingabe belegt. Sind die Elemente größer als die bereits zurückgeschriebenen, können sie in diesem Lauf mitverwendet werden. Ansonsten werden sie „gesperrt" und erst im nächsten Lauf verwendet. Ein Lauf endet, wenn nur noch gesperrte Einträge vorhanden sind. Mit einem solchen

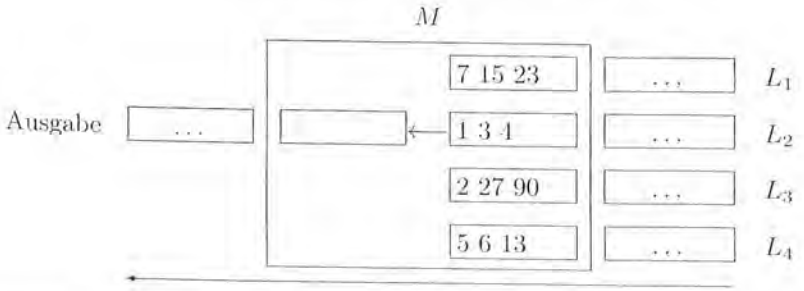


Abbildung 8.27: Demonstration des Mischvorgangs

Ausgabe	Speicher				Eingabe						
	10	20	30	40	25	73	16	26	33	50	31
10	20	25	30	40	73	16	26	33	50	31	
10 20	25	30	40	73	16	26	33	50	31		
10 20 25	(16)	30	40	73	26	33	50	31			
10 20 25 30	(16)	(26)	40	73	33	50	31				
10 20 25 30 40	(16)	(26)	(33)	73	50	31					
10 20 25 30 40 73	(16)	(26)	(33)	(50)	31						
16	26	31	33	50							

Abbildung 8.28: Berechnung der initialen Läufe mit Replacement-Selection

Verfahren lässt sich die Länge der initialen Läufe im Durchschnitt verdoppeln. Ein Beispiel ist in Abbildung 8.28 angegeben. Eingeklammerte Zahlen deuten gesperrte Elemente an.

Im ersten Schritt ist der Speicher mit den Zahlen 10, 20, 30 und 40 belegt. Die kleinste Zahl wird ausgegeben und durch den nächsten Wert der Eingabe ersetzt, die 25. Dieser Wert ist größer als die 10 und kann in diesem Lauf mitverwendet werden. Auf die gleiche Weise wird die 20 ausgegeben und die 73 von der Eingabe geholt. Der nächste Wert in der Eingabe, die 16, ist kleiner als das gerade kleinste Element 25. Daher kann sie nicht in diesem Lauf verwendet werden. Nach drei weiteren Schritten sind alle Werte im Speicher gesperrt und ein neuer Lauf muss begonnen werden.

Wir überlassen es den Lesern, den *Sort*-Iterator zu spezifizieren. Ähnlich wie beim *HashJoin*-Iterator muss auch der *Sort*-Iterator bei der Initialisierung (**open**) schon den Großteil der Arbeit verrichten. Nur die letzte *merge*-Phase wird während der sukzessiven Anforderung von Ergebnistupeln durch **next**-Aufrufe durchgeführt.

G-Join: Generalisierung von Sort-Merge- und Hash-Join

Es ist erstaunlich, dass man selbst fast drei Jahrzehnte nach der Erfindung des Hash-Join-Verfahrens immer noch algorithmische Verbesserungen für die Join-Berechnung erzielen kann. Beim sogenannten G-Join (Generalized Join) werden die Grundideen

des Sort-Merge-Joins (sortierte Runs) mit denen des Hash-Joins (temporärer Hash-Index für die Ermittlung der Join-Partner) kombiniert. Wir wollen dies an dem Beispiel $R \bowtie S$ demonstrieren, bei dem zunächst eine Run-Sortierung – am besten mittels Replacement-Selection – erfolgt:

- Sortiere Hauptspeicher-große Runs von R – falls R nicht schon nach dem Join-Attribut sortiert ist
- Gleichfalls wird S in Haupspeicher-große Runs sortiert, falls nicht ohnehin eine Sortierung von S gemäß dem Join-Attribut vorgegeben war

Der resultierende Zustand ist in Abbildung 8.29 an den äußeren Rändern gezeigt. Jedes Tupel wird durch seinen Join-Attributwert und die restlichen Daten repräsentiert. Wir gehen davon aus, dass R die kleinere der beiden Relationen ist (anderenfalls werden R und S vertauscht). Demzufolge besteht R aus zwei Runs, nachfolgend $R1$ und $R2$ genannt, und S aus den drei Runs $S1, S2, S3$. Anders als beim Sort-Merge-Join wird aber auf eine vollständige Sortierung verzichtet. Der G-Join „wandert" synchron durch die Runs von R und S. Dies ist schematisch in Abbildung 8.30 gezeigt. Es wird jeweils die noch nicht verarbeitete Seite mit dem kleinsten Anfangselement der Relation S verarbeitet. Dazu müssen alle Seiten der Runs von R in den Hauptspeicher-Puffer geladen sein bzw. werden, die sich mit der gerade aktiven S-Run-Seite überschneiden könnten. Diejenigen Seiten von R, die sich definitiv nicht mehr mit dem derzeitig aktiven Run von S überschneiden, weil deren Elemente kleiner sind als das kleinste Element der S-Seite, können schon wieder aus dem Hauptspeicher-Puffer verdrängt werden. Dies wird durch entsprechende Kontrollstrukturen bestehend aus vier sortierten Listen (implementiert als priority queues) für den Fortgang der Join-Berechnung gesteuert: A enthält die jeweils größten Join-Schlüssel der gepufferten R-Runs; B enthält die jeweils größten Join-Schlüssel der ältesten gepufferten Seiten der R-Runs; C enthält die kleinsten Join-Schlüssel der nächsten zu bearbeitenden Seite der S-Runs; D enthält deren größte Elemente. Es sei den Lesern überlassen, basierend auf diesen Kontrollstrukturen den Fortgang der Join-Berechnung und der Pufferbereinigung der R-Seiten zu konzipieren. Man kann davon ausgehen, dass jeweils ca. 2 bis 3 Seiten von jedem R-Run im Puffer sein müssen (dies ist aber keine Obergrenze – warum nicht?). Anders als in der Abbildung 8.29 gezeigt, kann man den Algorithmus so optimieren, dass immer nur eine einzige Seite von S, also nicht eine Seite pro S-Run, im Puffer sein muss. Zu diesem Zweck sollte jede S-Seite am Ende den kleinsten Schlüssel der nachfolgenden Run-Seite noch zusätzlich enthalten oder man verwendet für die Liste C den größten Wert der zuletzt schon bearbeiteten Seite des jeweiligen Runs. Wie auch in Abbildung 8.29 gezeigt, liefert der G-Join kein vollständig sortiertes Ergebnis, wie es der Sort-Merge-Join garantieren würde. Allerdings ist das Ergebnis „quasi-sortiert", so dass man in den meisten Fällen mit einer nachgeschalteten Replacement-Selection-Sortierung eine vollständige Sortierung in einer Phase erzielen kann.

8.2.7 Übersetzung der logischen Algebra

In diesem Abschnitt werden die einzelnen Operatoren der logischen Algebra in eine äquivalente Darstellung der physischen Algebra übersetzt. Dabei werden die phy-

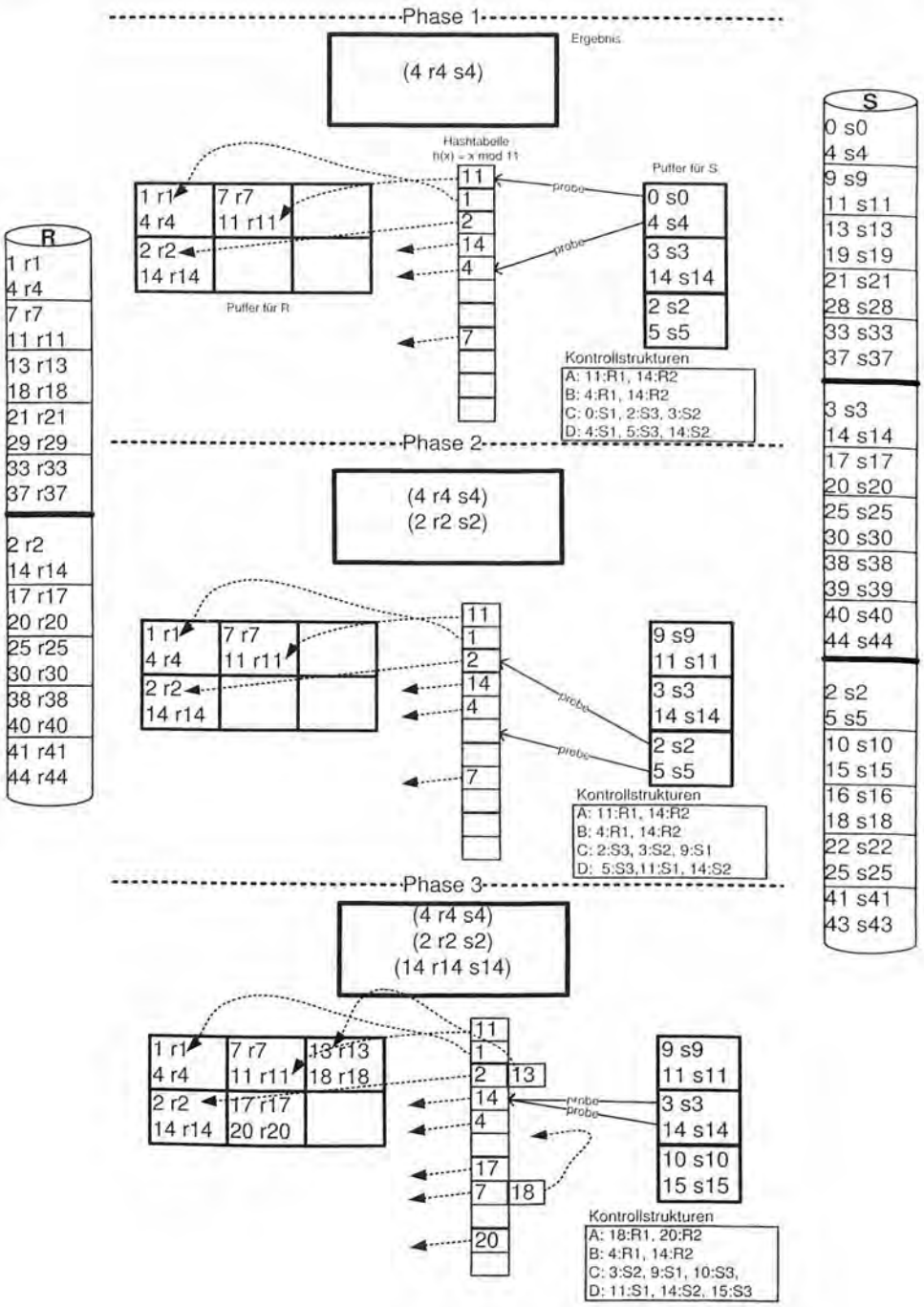


Abbildung 8.29: Die ersten drei Phasen des G-Joins

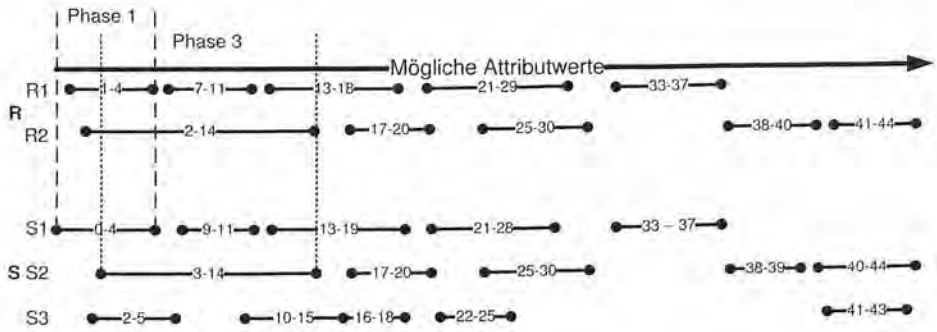


Abbildung 8.30: Die Verarbeitung der sortierten Runs im G-Join

sischen Eigenschaften der Daten ausgenutzt. Eine physische Eigenschaft kann z.B. „Eingabe befindet sich im Hauptspeicher" oder „Attribut A ist aufsteigend sortiert" sein. Diese Eigenschaften können von den Operatoren erhalten, neu eingeführt oder zerstört werden. Index-Join und Nested-Loop-Join erhalten beispielsweise die Sortierung in Attributen der äußeren Relation, nicht aber in Attributen der inneren. Ein Sort-Operator führt eine Sortierung neu ein.

Abbildung 8.31 zeigt einige Übersetzungsmöglichkeiten für relationale Operatoren. Die Argumente R und S deuten dabei nicht notwendigerweise abgespeicherte Relationen an, sondern beliebige weitere Teilbäume. Beispielsweise kann ein Join durch einen Merge-Join implementiert werden, wenn die Eigenschaft der Sortierung auf den Joinattributen vorhanden ist. Diese muss unter Umständen durch das Einfügen eines Sort-Operators erzeugt werden. Eine Projektion kann, wenn nötig, von einer Duplikateliminierung gefolgt werden. Setzt die Duplikateliminierungsmethode bestimmte physische Eigenschaften voraus, können diese durch Voranstellen eines entsprechenden Operators geschaffen werden. Eine „Möglichkeit" wird im Bild durch die eckigen Klammern $[\ldots]$ angedeutet. Eine Operation in eckigen Klammern kann fehlen, wenn sie nicht notwendig ist.

Ein möglicher Auswertungsplan für die Beispielanfrage aus Abbildung 8.7 könnte wie in Abbildung 8.32 aussehen. Dabei wurde angenommen, dass sich auf allen Primärschlüsseln des Schemas ein Primärindex in Form einer Hashtabelle befindet und auf dem Attribut *gelesenVon* der Relation *Vorlesungen* ein Sekundärindex (B^+-Baum). Für unsere Datenbasis ist die vorgestellte Auswertung sicherlich nicht die optimale, bei so kleinen Datenmengen würde sich ohnehin der Einsatz einer Indexstruktur nicht lohnen. Es wird jedoch demonstriert, wie ein guter Auswertungsplan bei einer realistischen (großen) Datenbasis aussehen könnte.

Zunächst wurde die Selektion in Ermangelung eines Indexes auf dem Namen der Professoren durch einen einfachen „Select" ersetzt. Der Sekundärindex auf dem Attribut *gelesenVon* ermöglicht den Einsatz eines Index-Joins, damit nicht die (normalerweise große) Relation *Vorlesungen* komplett gelesen werden muss.

Für den zweiten Join wurde ein Merge-Join ausgewählt. Dazu müssen beide Eingaben nach den Joinattributen sortiert sein. Alternativ wäre auch das Aufbauen einer temporären Indexstruktur möglich gewesen. Wie wir aber bereits gesehen ha-

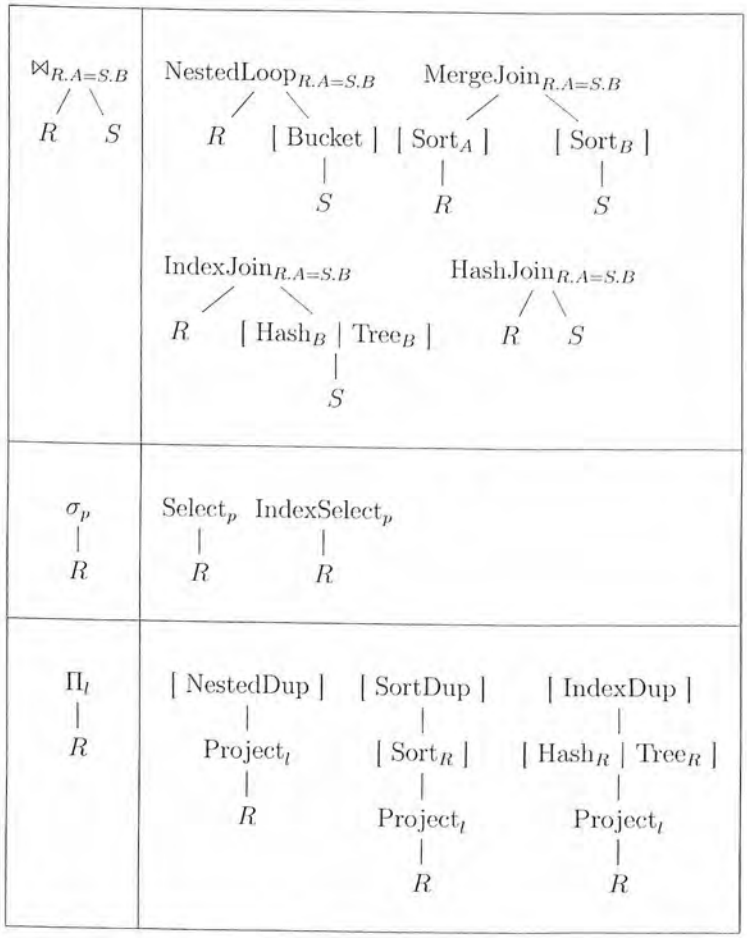


Abbildung 8.31: Mögliche Umsetzungen einiger relationaler Operatoren

Abbildung 8.32: Ein Auswertungsplan

ben, sind temporäre Indexstrukturen i.A. nur sinnvoll, wenn sie im Hauptspeicher Platz finden (Für „Hash" und „Tree" sollten demnach Hauptspeicher-Versionen verwendet werden). Es ist zu beachten, dass *VorlNr* nur ein Teil des Primärschlüssels der Relation *hören* ist und daher der Primärindex im Allgemeinen nicht ausgenutzt werden kann.

Im nächsten Join kann der Primärindex der Relation *Studenten* ausgenutzt werden. Das einzige, was noch übrig bleibt, ist die Projektion auf das Attribut *Semester* durchzuführen. Da die Projektion alleine keine Duplikateliminierung durchführt, wird diese nachträglich angewendet. Der Operator, der die Duplikateliminierung durchführt, wurde im Bild „IndexDup" genannt. Diese Wahl ist für das Beispiel wohl tatsächlich die effizienteste, weil wir, wegen des kleinen Wertebereichs von *Semester* (siehe Abschnitt 8.1.2), nur mit wenigen Tupeln in der Ausgabe rechnen müssen. Die Hashtabelle bleibt daher klein und kann im Speicher gehalten werden. Die Eingabe braucht lediglich einmal sequentiell gelesen zu werden.

8.3 Kostenmodelle

Heuristische Optimierungstechniken sind darauf ausgerichtet, in der Mehrzahl der Fälle innerhalb kurzer Laufzeit gute Ergebnisse – d.h. nahezu optimale Anfrageauswertungspläne – zu liefern. Leider generieren derartige Heuristiken manchmal eher schlechte Auswertungspläne. Um dies auszuschließen, ist ein Kostenmodell notwendig, mit dem man verschiedene alternative Auswertungspläne miteinander vergleichen kann, um den besten auszuwählen.

Ein Kostenmodell stellt Funktionen zur Verfügung, die den Aufwand, d.h. die

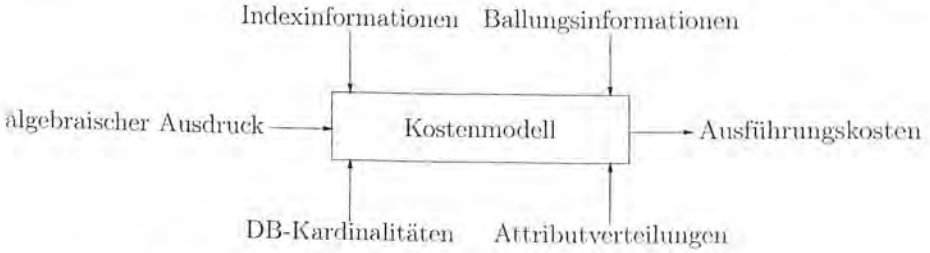


Abbildung 8.33: Funktionsweise des Kostenmodells

Laufzeit, der Operatoren der physischen Algebra abschätzen. Dazu werden diverse Parameter benötigt, die schon in der Einleitung dieses Kapitels erwähnt wurden, unter anderem Informationen über Indices, Ballungen, Kardinalitäten und Verteilungen (siehe Abbildung 8.33). Wie schon anfangs erwähnt, werden wir nur einige Varianten der Operatoren etwas detaillierter beschreiben und bewerten. Vorher muss aber noch einiges an Vorarbeit geleistet werden. Bei der Aufwandsbestimmung spielt in vielen Fällen eine Rolle, wieviele Tupel sich bei Auswertung einer Bedingung qualifizieren würden. Diesem Thema wollen wir uns zuerst widmen.

8.3.1 Selektivitäten

Der Anteil der qualifizierenden Tupel wird die *Selektivität sel*[3] genannt. Für die Selektion und den Join ist sie wie folgt definiert:

- Selektion mit Bedingung p:

$$sel_p := \frac{|\sigma_p(R)|}{|R|}$$

- Join von R mit S:

$$sel_{RS} := \frac{|R \bowtie S|}{|R \times S|} = \frac{|R \bowtie S|}{|R| \cdot |S|}$$

Die Selektivität der Selektion gibt also den relativen Anteil der Tupel an, die das Selektionskriterium p erfüllen. Beim Join wird der Anteil relativ zur Kardinalität des Kreuzprodukts angegeben.

Nun muss irgendwie die Selektivität abgeschätzt werden, damit man Rückschlüsse auf die Größe der Zwischenergebnisse ($|\sigma_p(R)|$ bzw. $|R \bowtie S|$) ziehen kann. Einfache Abschätzungen sind z.B.

- Die Selektivität der Operation $\sigma_{R.A=c}$, also des Vergleiches des Attributs A aller Tupel von R mit der Konstante c, beträgt $1/|R|$, falls A ein Schlüssel ist (es qualifiziert sich ein Tupel von $|R|$ möglichen).

[3] In der Literatur wird die Selektivität häufig als σ bezeichnet. Hier wird sie *sel* genannt, damit Verwechslungen mit dem Selektionsoperator ausgeschlossen sind.

- Bei einer Gleichverteilung der Werte von $R.A$ ist die Selektivität der Operation $\sigma_{R.A=c}$ $1/i$. Dabei ist i die Anzahl der unterschiedlichen Attributwerte (jedes i-te Tupel qualifiziert sich).

- Besitzt bei einem Equijoin $(R \bowtie_{R.A=S.B} S)$ das Attribut A Schlüsseleigenschaften, kann die Größe des Ergebnisses mit $|S|$ abgeschätzt werden. Jedes Tupel aus S findet nur maximal einen Joinpartner (abhängig von der Einhaltung der referentiellen Integrität), da B wohl einen Fremdschlüssel auf R darstellt. In diesem Falle ist die Selektivität $sel_{RS} = 1/|R|$.

Das sind jedoch nur Spezialfälle. Im Allgemeinen ist man auf anspruchsvollere Methoden zur Selektivitätsabschätzung angewiesen, denn vielfach sind Annahmen über Gleichverteilung bzw. Schlüsseleigenschaften nicht gegeben. In der Literatur sind drei Arten von Verfahren bekannt, mit deren Hilfe man testen kann, wieviele Tupel sich innerhalb eines bestimmten Wertebereiches befinden:

a) parametrisierte Verteilungen,

b) Histogramme und

c) Stichproben.

Die erste Methode versucht, zu der vorhandenen Werteverteilung die Parameter einer Funktion so zu bestimmen, dass diese die Verteilung möglichst gut annähert. Sie ist in Abbildung 8.34 a) vereinfacht dargestellt. Dort sind zwei Normalverteilungen mit unterschiedlichen Parametern und die tatsächliche Verteilung aufgetragen. Man sieht, dass beide Normalverteilungen die tatsächliche Verteilung in bestimmten Bereichen nicht gut annähern können.

Eine Abschätzung der Selektivität ist recht einfach zu berechnen, die Funktion liefert die Anzahl der Tupel im qualifizierenden Bereich. Unter Umständen können die Parameter bei Veränderungen (updates) der Datenbasis nachgezogen werden, so dass sich die Verteilungskurve angleicht.

Leider sind realistische Werteverteilungen oft nicht gut mit parametrisierten Funktionen annäherbar. Vor allem bei mehrdimensionalen Anfragen (also z.B. bei Selektionen, die sich auf mehrere Attribute beziehen) gestaltet sich dieses schwierig. In der Literatur wurden dafür flexiblere, aber auch wesentlich kompliziertere Verteilungen als in unserer Skizze vorgestellt. Auch muss eine sinnvolle und effiziente Möglichkeit der Parameterbestimmung bestehen. Hier kann auf Stichprobenverfahren zurückgegriffen werden.

Bei Histogrammverfahren wird der Wertebereich der betreffenden Attribute in Intervalle unterteilt und alle Werte gezählt, die in ein bestimmtes Intervall fallen. Das ist in Abbildung 8.34 b) dargestellt. Auf diese Weise ist eine sehr flexible Annäherung der Verteilung möglich.

Normale Histogramme unterteilen den Wertebereich in äquidistante Stücke, wie es in der Abbildung zu sehen ist. Das hat den Nachteil, dass vergleichsweise viele Unterteilungen in spärlich besetzten Bereichen vorgenommen werden. Dafür werden sehr häufig vorkommende Werte nur ungenau abgeschätzt. Aus diesem Grund wurden sogenannte *Equi-Depth-Histogramme* vorgeschlagen, die den Wertebereich so in Abschnitte unterteilen, dass in jeden Abschnitt gleich viele Werte fallen. So sind

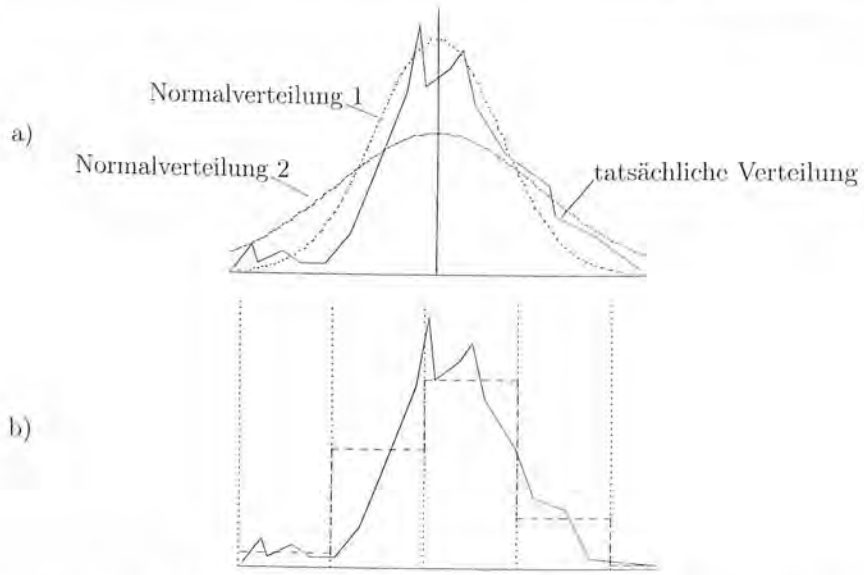


Abbildung 8.34: Schematische Darstellung der Selektivitätsabschätzung: a) parametrisierte Verteilungen und b) ein Histogramm

Abschnitte mit selten vorkommenden Werten sehr breit und solche mit häufig vorkommenden Werten sehr schmal. Mit dieser Methode ist eine genauere Annäherung an die tatsächliche Verteilung möglich. Nachteilig ist der höhere Verwaltungsaufwand, da Equi-Depth-Histogramme nur mit hohen Kosten an Veränderungen der Datenbasis anpassbar sind.

Stichprobenverfahren zeichnen sich durch ihre außerordentliche Einfachheit aus. Es wird einfach eine zufällige Menge von Tupeln einer Relation gezogen und deren Verteilung als repräsentativ für die ganze Relation angesehen. Das Lesen der Tupel erfordert jedoch „teure" Zugriffe auf den Hintergrundspeicher. Es ist daher wichtig, dass nicht mehr Zeit durch das Ziehen von Stichproben aufgewendet wird als für eine beliebige Abarbeitung der Anfrage. Insofern muss ein gutes Stichprobenverfahren adaptiv sein.

8.3.2 Kostenabschätzung für die Selektion

Wir gehen davon aus, dass Hintergrundspeicherzugriffe so dominierend sind, dass der CPU-Aufwand vernachlässigbar ist. Die Frage lautet also: Wieviele Hintergrundspeicherzugriffe verursacht eine Selektion?

Diese Frage ist im Falle des Select-Operators sehr einfach zu beantworten: Handelt es sich bei der Eingabe um eine auf dem Hintergrundspeicher abgelegte Relation, müssen alle zur Relation gehörenden Blöcke gelesen werden. Falls die Eingabe von einem anderen Iterator produziert wurde, kann sie einfach entsprechend der Selektionsbedingung „gefiltert" werden. Dabei entstehen keine weiteren Hintergrundspeicherzugriffe.

Im Falle der Selektion mit Index-Unterstützung müssen die Kosten durch den Indexzugriff berücksichtigt werden. Es ist eine gebräuchliche Vereinfachung, für ein Absteigen der Knoten eines B^+-Baums zwei Hintergrundspeicher-Zugriffe zu veranschlagen. Man geht davon aus, dass ein sinnvoll eingesetzter B^+-Baum eine Höhe von $h = 3$ oder $h = 4$ hat. Bei häufigem Zugriff befindet sich die Wurzel und zumindest ein Teil der zweiten Ebene im Datenbank-Puffer.

Betrachten wir die Operation $\sigma_{A\theta c}(R)$, wobei A ein Attribut, c eine Konstante und θ ein Vergleichsoperator ist. Die Relation R besitze einen Cluster-Index in Form eines B^+-Baums auf dem Attribut A, sie ist also nach A sortiert abgelegt. Dann wird für die Selektion der Wert c im B^+-Baum nachgeschlagen. Von dort aus kann – die Richtung wird durch den Vergleichsoperator bestimmt – die Relation sequentiell durchlaufen werden, bis die Bedingung nicht mehr zutrifft. Wenn die Relation also b_R Blöcke des Hintergrundspeichers belegt und für das Nachschlagen im Index t Zugriffe notwendig sind, ergeben sich die Gesamtkosten in etwa zu

$$t + \lceil sel_{A\theta c} \cdot b_R \rceil$$

Bei einer Hashtabelle geht man von einem Hintergrundspeicherzugriff aus, wenn ein verzeichnisloses Verfahren wie lineares Hashing verwendet wird. Beim erweiterbaren Hashing muss mit zwei Zugriffen gerechnet werden, da das Verzeichnis im Allgemeinen sehr groß ist. Es kann nicht davon ausgegangen werden, dass sich der gerade gesuchte Teil im Puffer befindet.

Da eine Hashtabelle normalerweise nicht ordnungserhaltend ist, geht man davon aus, dass jeder Wert, der das Selektionsprädikat erfüllt, auch nachgeschlagen werden muss. Wenn also ein Zugriff auf die Hashtabelle h Seitenfehler verursacht und d unterschiedliche Werte nachgeschlagen werden müssen, ergeben sich die Gesamtkosten zu $h \cdot d$. Hierbei nehmen wir vereinfachend an, dass alle Tupel mit gleichem Wert in denselben Behälter passen.

8.3.3 Kostenabschätzung für den Join

Betrachten wir den „verfeinerten" Nested-Loop-Join. Seien dabei b_R und b_S die Anzahl der Seiten, die R respektive S belegen. Für die innere Schleife wurden k Seiten reserviert, für die äußere $m - k$. Die Relation R wird einmal vollständig durchlaufen, es ergeben sich dabei b_R Seitenzugriffe. Die innere Schleife wird $\lceil b_R/(m-k) \rceil$-mal durchlaufen. Bei jedem Durchlauf ergeben sich $b_S - k$ Zugriffe, da durch das Zick-Zack-Vorgehen die letzten k Seiten des vorigen Durchlaufes wiederverwendet werden können. Lediglich der erste Durchlauf muss komplett alle b_S Seiten lesen. Insgesamt ergeben sich die Kosten zu

$$b_R + k + \lceil b_R/(m-k) \rceil \cdot (b_S - k)$$

Für realistische Größen des Puffers und der Relationen wird dieser Ausdruck minimal, wenn R die kleinere der beiden Relationen ist und für die innere Schleife nur ein Puffer von einer Seite ($k = 1$) verwendet wird (siehe Aufgabe 8.7).

8.3.4 Kostenabschätzung für die Sortierung

Zu Beginn werden die b_R Seiten der Eingaberelation in sortierte Level-0 Läufe geschrieben, die jeweils m Seiten groß sind. Von diesen gibt es $i = \lceil b_R/m \rceil$ Stück. Bei der Replacement-Selection-Strategie kann man im Durchschnitt von $2 \cdot m$ anstelle von m Seiten in jedem Level-0 Lauf ausgehen.

Während des Mischens werden auf jeder Stufe jeweils $m - 1$ Läufe gleichzeitig betrachtet. Insgesamt werden daher $l = \lceil \log_{m-1}(i) \rceil$ Stufen benötigt.

Jede Stufe liest und schreibt im schlimmsten Fall alle Tupel der Relation. Als Gesamtkosten ohne Berücksichtigung der möglichen Optimierungen ergeben sich

$$2 \cdot l \cdot b_R.$$

8.4 „Tuning" von Datenbankanfragen

Entwickler von zeitkritischen Datenbankanwendungen werden in der Regel nicht umhin kommen, ihre Anfragen bzw. die vom Optimierer generierten Auswertungspläne zu analysieren. Aus dieser Analyse kann man dann Rückschlüsse ziehen, ob unbefriedigende Antwortzeiten auf Fehler beim physischen Datenbankentwurf – wie z.B. fehlende Indices, ungünstige Objektballung – oder auf ungeeignete Auswertungspläne zurückzuführen sind.

Zunächst sollten Datenbankbenutzer darauf achten, dass viele DBMS-Produkte unterschiedliche Optimierungslevel anbieten. Die Optimierungslevel legen u.a. fest, wieviel Zeit der Optimierer verwenden sollte, nach einem guten (möglichst dem optimalen) Plan zu suchen. Möglicherweise wird aus Effizienzgründen des Optimierungsvorgangs ganz auf eine Kostenberechnung verzichtet, indem nur heuristische Regeln (Pushing Selections, Nutzung von Indices wann immer anwendbar, etc.) angewendet werden. Fast alle DBMS-Produkte haben heute aber (auch) einen kostenbasierten Optimierer, der viele mögliche Anfragepläne generiert und von diesen den billigsten – gemäß der Kostenabschätzung – auswählt. Das Kostenmodell des Optimierers kann aber nur dann vernünftig funktionieren, wenn entsprechende Statistikdaten über die Datenbank zur Verfügung stehen. Diese werden i.A. weder automatisch generiert noch bei Datenbankänderungen fortgeschrieben – das würde die Änderungsoperationen auf der Datenbank zu sehr „bestrafen". Die Datenbankadministratoren müssen die Generierung der Statistiken explizit anstoßen. Dazu dient z.B. in Oracle7 der Befehl

analyze table Professoren **compute statistics for table**;

Die Sprachkonzepte zum Tuning von Datenbanksystemen sind leider nicht standardisiert. So hat beispielsweise der Befehl zum Sammeln von Statistiken in DB2 folgende Form:

runstats on table ...

Auf diese Art müssen alle Datenbankstrukturen (Relationen und auch Indices) analysiert werden. Es ist darauf zu achten, dass man diese Analyse periodisch wiederholt, da der kostenbasierte Optimierer nur dann vernünftig arbeiten kann, wenn die Statistiken (einigermaßen) „up-to-date" sind. Die ermittelten Statistiken werden in speziellen Relationen des sogenannten *Data Dictionary*, wo auch die Schemainformation verwaltet wird, gespeichert. Hierauf hat man als autorisierter Benutzer auch Zugriff um diese Daten auswerten zu können.

Falls nach Erstellung bzw. „Auffrischung" der Statistiken einige Anfragen immer noch eine unbefriedigende Antwortzeit haben, kann man sich in den meisten Systemen die generierten Anfrageauswertungspläne anzeigen lassen. Dazu gibt es den **explain plan**-Befehl, der leider wiederum in uneinheitlicher Syntax von den Systemen benutzt wird. In Oracle7 könnte man sich mit

```
explain plan for
  select distinct s.Semester
  from Studenten s, hören h, Vorlesungen v, Professoren p
  where p.Name = 'Sokrates' and v.gelesenVon = p.PersNr and
        v.VorlNr = h.VorlNr and h.MatrNr = s.MatrNr;
```

den vom Optimierer generierten Plan ausgeben lassen. Präziser gesagt, der Auswertungsplan wird in einer speziellen Relation *plan_table* abgelegt, aus der man ihn sich mittels einer SQL-Anfrage ausgeben lassen kann. Beispielsweise hat unsere Datenbankinstallation folgenden Plan generiert:

```
SELECT STATEMENT   Cost = 37710
  SORT UNIQUE
    HASH JOIN
      TABLE ACCESS FULL STUDENTEN
      HASH JOIN
        HASH JOIN
          TABLE ACCESS BY ROWID PROFESSOREN
            INDEX RANGE SCAN PROFNAMEINDEX
          TABLE ACCESS FULL VORLESUNGEN
        TABLE ACCESS FULL HOEREN
```

Die Kostenbewertung beträgt 37710 Einheiten (was immer eine *Einheit* ist). Der Plan entspricht dem in Abbildung 8.35 gezeigten Auswertungsbaum. Die am weitesten nach innen eingerückten Operationen werden also zuerst ausgewertet. Viele Datenbankprodukte haben mittlerweile auch grafische Benutzerschnittstellen, in denen solche Auswertungspläne als Baumdarstellung angezeigt werden können.

Sollte man bei der Analyse eines generierten Auswertungsplans feststellen, dass der Optimierer einen sub-optimalen Plan ausgewählt hat (z.B. aufgrund fehlerhafter Selektivitätsabschätzungen), lassen sich in einigen DBMS-Produkten sogenannte „Hints" angeben, mit denen man dem Optimierer den „richtigen Weg weisen" kann, um zu einem besseren Auswertungsplan zu gelangen. Manchmal ist leider auch ein Umschreiben der Anfragen nötig, um gute Antwortzeiten zu erzielen. Dies ist z.B. oft bei tief geschachtelten (korrelierten) Unteranfragen der Fall – wie wir sie beispielsweise bei den Anfragen mit Allquantifizierung in Abschnitt 4.12 gesehen hatten.

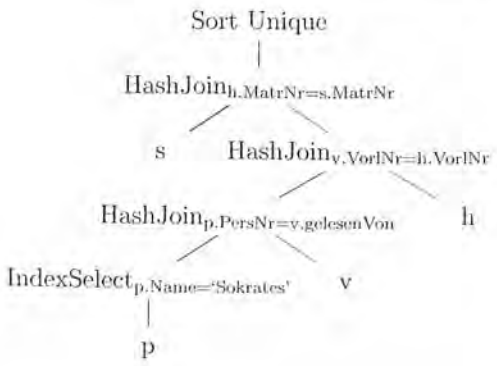


Abbildung 8.35: Baumdarstellung des Auswertungsplans

8.5 Kostenbasierte Optimierer

In den bisherigen Ausführungen hatten wir Heuristiken (wie „pushing selections") angewendet, um gute Anfrageauswertungspläne zu erhalten. Die so generierten Pläne sind allerdings in der Regel nicht optimal. Deshalb verwenden kommerzielle Datenbanksysteme heutzutage kostenbasierte Optimierer, die möglichst den gesamten Suchraum nach dem *optimalen* Anfrageauswertungsplan absuchen.

Wir werden uns nachfolgend auf die Optimierung der Joinreihenfolge konzentrieren, da dies die wichtigste Optimierungsaufgabe darstellt.

8.5.1 Suchraum für die Join-Optimierung

Der Suchraum für die Optimierung der Joinreihenfolge besteht aus der Menge aller Anfrageauswertungspläne (die als Operatorbäume dargestellt werden), welche die zu verknüpfenden Relationen als Blattknoten enthalten und deren innere Knoten den Join-Operationen entsprechen. Da die Join-Operation kommutativ und assoziativ ist, ist die resultierende Kardinalität des Suchraums sehr groß. In der Vergangenheit haben sich einige Datenbanksysteme deshalb auf einen Teilbereich des gesamten Suchraums beschränkt: die sogenannten *links-tiefen* Auswertungsbäume. Diese zeichnen sich dadurch aus, dass jeder Join als rechtes Argument eine Basis-Relation (und keine zwischenberechnete Relation) hat.

Links-tiefe Auswertungspläne

Wir wollen die unterschiedlichen Pläne anhand einer Bespielanfrage illustrieren:

select distinct s.Name
from Vorlesungen v, Professoren p, hören h, Studenten s
where p.Name = 'Sokrates' **and** v.gelesenVon = p.PersNr **and**
v.VorlNr = h.VorlNr **and** h.MatrNr = s.MatrNr **and** s.Semester > 13;

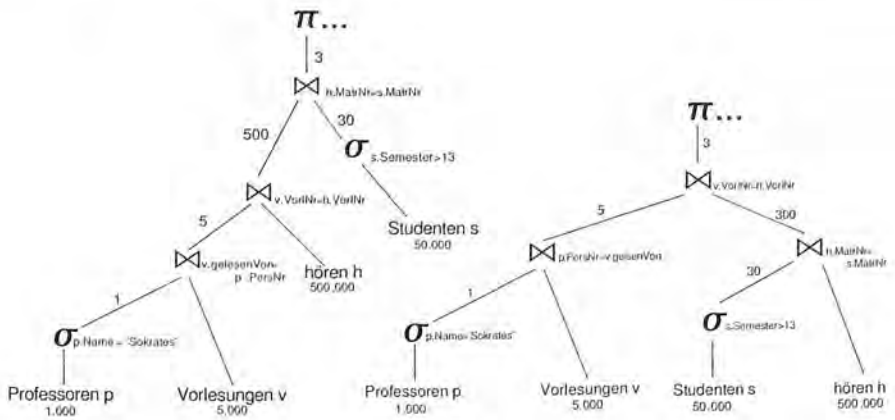


Abbildung 8.36: Links-tiefer bzw. buschiger Join-Auswertungsbaum

In dieser Anfrage sollen die Namen der „alten" Sokrates-Studenten ermittelt werden. Auf der linken Seite von Abbildung 8.36 ist ein links-tiefer Anfrageauswertungsplan gezeigt. Wir haben auch die Kardinalitäten für eine typische (Massen-) Universität annotiert.

Allgemein zeichnen sich solche links-tiefen Pläne dadurch aus, dass sie die n zu verknüpfenden Relationen R_1, R_2, ..., R_n wie folgt klammern:

$$((\cdots(R_{i_1} \bowtie R_{i_2}) \bowtie \cdots) \bowtie R_{i_n})$$

Offensicht gibt es $n!$ Möglichkeiten/Permutationen, die n Relationen auf diese Art und Weise zu verknüpfen. Hierbei gibt es allerdings auch Fälle, in denen die Joins zu Kreuzprodukten „degenerieren", weil es keine Joinprädikate für die zu kombinierenden Relationen gibt. Für unsere Sokrates-Anfrage wäre dies der Fall, wenn man beispielsweise mit dem Join/Kreuzprodukt

$$\text{Vorlesungen} \times \text{Studenten}$$

beginnen würde.

Die links-tiefen Auswertungspläne haben den Vorteil, dass sie den Datenfluss von einer Joinberechnung zur nächsten besonders gut unterstützen. Wenn man beispielsweise alle Joins als nested-loop-Joins realisiert, könnte man auf die Zwischenspeicherung gänzlich verzichten, da die Tupel von links direkt in die Joinberechnung „fließen" könnten.

Buschige Auswertungspläne

Auf der rechten Seite von Abbildung 8.36 ist ein buschiger (engl. *bushy*) Anfrageauswertungsplan gezeigt. Bei diesen Auswertungsplänen hat man maximale Flexibilität in der Auswahl der günstigsten Join-Reihenfolge. Wir haben in der Abbildung für beide Pläne auch die geschätzten Kardinalitäten der Zwischenergebnisse mit angegeben, die auf den Kardinalitäten einer typischen großen Universität mit 1000

Professoren, die jeweils 5 Vorlesungen halten, und 50.000 Studenten, die jeweils 10 Vorlesungen hören, basieren. Für unsere Sokrates-Anfrage ist der buschige Plan vermutlich nur geringfügig besser (wenn überhaupt); es gibt aber Fälle, in denen ein buschiger Auswertungsplan dem besten links-tiefen Plan bei Weitem überlegen ist (siehe dazu Übungsaufgabe 8.14). Der Suchraum der buschigen Anfrageauswertungspläne ist nochmals deutlich größer als derjenige für links-tiefe Bäume. Bei n zu verknüpfenden Relationen gibt es

$$\binom{2(n-1)}{n-1}(n-1)! = (2(n-1))!/(n-1)!$$

verschiedene buschige Auswertungsbäume (siehe dazu Übungsaufgabe 8.3).

In der nachfolgenden Tabelle sind die Suchraumgrößen für links-tiefe ($n!$) und buschige Auswertungspläne (rechte Spalte) im Vergleich zu der Exponentialfunktion dargestellt. Wir sehen, dass die Kardinalitäten jenseits von 10 Relationen geradezu „explodieren". Die Funktion c^n ist nur zum Vergleich aufgeführt.

Suchraum für links-tiefe bzw. buschige Pläne			
n: #Relationen	zum Vergleich: c^n	$n!$	$(2(n-1))!/(n-1)!$
2	7	2	2
5	146	120	1680
10	22026	3628800	$1,76 * 10^{10}$
20	$4,85 * 10^9$	$2.4 * 10^{18}$	$4,3 * 10^{27}$

8.5.2 Dynamische Programmierung

Wegen der „explodierenden" Größe des Suchraums ist es unmöglich, dass ein Datenbankoptimierer jeden Auswertungsbaum einzeln erstellt, um seine Kosten zu bewerten. Vielmehr benötigt man Algorithmen, die schon sehr frühzeitig alle Bäume mit „hoffnungslosen" Teilbäumen eliminieren (im Englischen *pruning* genannt). Genau dieses *Pruning* erreichen Optimierer, die auf dynamischer Programmierung basieren. Hierbei handelt es sich um den klassischen Optimierungsalgorithmus, der schon 1979 im relationalen Datenbanksystem *System R* von IBM eingebaut wurde. Die dynamische Programmierung, abgekürzt DP, ist anwendbar, wenn das Gesamtoptimierungsproblem nicht monolithisch als Ganzes zu lösen ist, sondern sich in kleinere Unterproblemen partitionieren lässt. Anstatt den gesamten Suchraum vollständig – also einen Auswertungsbaum nach dem anderen – zu durchsuchen, werden hierbei optimale Lösungen „kleinerer" Probleme zu optimalen Lösungen „größerer" Probleme zusammengefügt. Die kleineren Probleme wurden selbst wieder aus den optimalen Lösungen noch kleinerer Probleme zusammengefügt. Bei der dynamischen Programmierung fängt man somit an, die kleinstmöglichen Problemstellungen zu lösen, danach die nächstgrößeren Probleme, usw. – bis man bei der ursprünglichen Problemstellung angekommen ist. Diese Lösungen der Teilprobleme werden aber nicht jeweils bei Bedarf neu berechnet, sondern in einer geeigneten Datenstruktur (einer Tabelle oder einem Array) abgespeichert.

Die Voraussetzung für die Anwendung des algorithmischen Prinzips der dynamischen Programmierung basiert auf dem sogenannten Bellman'schen Optimalitätskriterium. Bellman (1957) ist der Erfinder der dynamischen Programmierung. Dieses

Bellman'sche Kriterium verlangt, dass die optimale Lösung selbst wieder aus optimalen Teillösungen besteht. Wir demonstrieren dies anhand des Optimierungsproblems, für die Menge $S = \{R_1, \ldots, R_m\}$ von Relationen die bestmögliche Joinreihenfolge zu bestimmen. Dies ist in Abbildung 8.37 visualisiert. Nehmen wir an, dass wir (woher auch immer) wissen, dass die bestmögliche Lösung auf der obersten Ebene einen Join ausführt, der als Argument von links die Kombination der Relationen $S - O$ und von rechts die Verknüpfung der Relationen O erhält. Das Bellman'che Optimalitätskriterium verlangt, dass die Teilprobleme selbst – für sich isoliert betrachtet – wieder optimale Lösungen darstellen. D.h. die linke Wolke des Joinplans entspricht dem optimalen Auswertungsbaum, um die in $S - O$ enthaltenen Relationen zu verknüpfen und Analoges gilt für die rechte Wolke.

Abbildung 8.37: Zerlegung des Optimierungsproblems in Teilprobleme

Die oben beschriebene Zerlegung des Problems in Teilprobleme erfolgt bei der klassischen dynamischen Programmierung aber nicht *top-down* sondern *bottom-up*. Dazu wird eine Tabelle (Array) – in unserem Algorithmus *BestePläneTabelle* genannt – erstellt, in der die Lösungen der schon bearbeiteten Teilprobleme abgelegt sind. Der Algorithmus fängt in der Tabelle mit den einzelnen in der Anfrage vorkommenden Basisrelationen an und bestimmt die bestmöglichen Zugriffsmethoden für die (benötigten) Tupel dieser Relationen. Dies geschieht in den Schritten 1 bis 2 des Algorithmus *DynProg* (siehe Abbildung 8.38). Wenn beispielsweise für eine Relation R ein Cluster-Index auf $R.A$ existiert kommt neben dem normalen *Scan*-Zugriff auch ein *Index-Scan* in Frage. Dieser sogenannte $iscan_A(R)$ hat den Vorteil, dass die Ergebnisse sortiert nach $R.A$ geliefert werden. Falls es eine Selektion basierend auf $R.B$ in der Anfrage q gibt, kommt eine Auswahl der Ergebnistupel über den Index $R.B$ (auch wenn es kein Cluster-Index ist) in Betracht. Dies bezeichnet man als *Index-Seek* oder abgekürzt $iseek_{B\Phi c}(R)$ bei einer Selektion $\sigma_{B\Phi c}(R)$. In Schritt (3) wird die *BestePläneTabelle* beschnitten, indem alle Zugriffspläne, für die es bessere äquivalente Pläne gibt, eliminiert werden.

In den Schritten (5) bis (9) werden die zuvor berechneten Teillösungen zu Plänen für größere Teillösungen zusammengefügt. Die möglichen Verküpfungen werden durch die Routine *joinPläne* generiert, die hier vereinfacht nur zwischen Joins und Kreuzprodukten (wenn es kein Joinprädikat gibt) unterscheidet. In der Praxis könnte man an dieser Stelle auch die physischen Ausführungsalgorithmen (Hash-Join, MergeJoin, etc.) durchprobieren. Da durch die Teilmengen-basierte Iteration alle Kombinationen aller Teillösungen versucht werden, handelt es sich hierbei um einen Optimierer für allgemeine buschige Pläne, die natürlich als Sonderfall die links-tiefen Pläne beinhalten. Zwischendurch wird mittel der *beschneidePläne*-Routine eine Be-

Function DynProg
input Eine Anfrage q über Relationen $R_1, \ldots, R_n$ *//zu joinende Relationen*
output Ein Anfrageauswertungsplan für q *//Auswertungsbaum*

```
1:  for i = 1 to n do {
2:      BestePläneTabelle({R_i}) = ZugriffsPläne(R_i)
3:      beschneidePläne(BestePläneTabelle({R_i}))   //Pruning
4:  }
5:  for i = 2 to n do {    //bilde i-elementige Teillösungen
6:      for all S ⊆ {R_1, ..., R_n} so dass |S| == i do {
7:          BestePläneTabelle(S) = ∅
8:          for all O ⊂ S do {    //probiere alle mgl. Joins zwischen Teilmengen
9:              BestePläneTabelle(S) = BestePläneTabelle(S) ∪
                        joinPläne(BestePläneTabelle(O), BestePläneTabelle(S − O))
10:             beschneidePläne(BestePläneTabelle(S))
11:         }
12:     }
13: }
14: beschneidePläne(BestePläneTabelle({R_1, ..., R_n}))
15: return BestePläneTabelle({R_1, ..., R_n})
```

Abbildung 8.38: Pseudocode für die Joinoptimierung mittels der dynamischen Programmierung

BestePläneTabelle	
Index (S)	Alternative Pläne
s, h, v	scan(s) ⋈ (scan(h) ⋈ $\text{iseek}_{\text{SWS=2}}$(v))
h, v	scan(h) ⋈ $\text{iseek}_{\text{SWS=2}}$(v), ...
s, v	scan(s) × $\text{iseek}_{\text{SWS=2}}$(v), ...
s, h	scan(s) ⋈ scan(h), ~~$\text{iscan}_{\text{VorlNr}}$(h) ⋈ $\text{iscan}_{\text{MatrNr}}$(s)~~, ...
Vorlesungen v	scan(v) , $\text{iseek}_{\text{SWS=2}}$(v) , ~~$\text{iscan}_{\text{VorlNr}}$(v)~~
hören h	scan(h) , ~~$\text{iseek}_{\text{MatrNr}}$(h)~~ , $\text{iscan}_{\text{VorlNr}}$(h)
Studenten s	scan(s) , ~~$\text{iseek}_{\text{Semester<5}}$(s)~~ , $\text{iscan}_{\text{MatrNr}}$(s)

Abbildung 8.39: *BestePläneTabelle* für unsere Beispiel-Anfrage

schneidung (Pruning) durchgeführt, um frühzeitig suboptimale Teillösungen zu eliminieren. Anders als bei der „reinen" dynamischen Programmierung überleben bei der Anfrageoptimierung u.U. mehrere alternative Pläne pro Zeile der *BestePläneTabelle*. Das ist darauf zurückzuführen, dass Pläne mit höheren Kosten möglicherweise andere interessante Eigenschaften aufweisen (engl *interesting properties*), die sich später auszahlen (amortisieren) könnten. Hierzu zählen insbesondere Sortierungen des Zwischenergebnisses, die bei einem späteren Merge-Join oder einer Aggregation hilfreich sein könnten. Sobald die äußere Schleife (5) terminiert, gibt es in der obersten Zeile mindestens eine Lösung für alle Relationen $\{R_1, \ldots, R_n\}$. Aus diesen wird dann im Zuge des *Prunings* (14) der billigste Plan ausgewählt; hier gibt es keine interessanten Eigenschaften mehr zu berücksichtigen. Warum?

Die Speicherung der Teillösungen in der *BestePläneTabelle* ist für eine Beispielanfrage in Abbildung 8.39 gezeigt. Die Tabelle basiert auf einer einfachen Anfrage, die die Studenten ermittelt, die schon im Grundstudium (also in den ersten vier Semestern) eine Spezialvorlesung (die man an dem geringen Umfang von 2 SWS erkennt) hören:

select distinct s.Name
from Vorlesungen v, hören h, Studenten s
where v.VorlNr = h.VorlNr **and** h.MatrNr = s.MatrNr **and**
s.Semester < 5 **and** v.SWS = 2;

Als Zugriffspläne für die Relation *Studenten* kommt ein vollständiger *scan* in Frage oder ein Clusterindex-basierter Zugriff nach *MatrNr*, der die Tupel in sortierter Reihenfolge liefert. Eine weitere Zugriffsmöglichkeit besteht über einen sekundären Index für das Attribut *Semester*. Dieser Zugriff ist eher ungünstig, da sehr viele Studenten in niedrigen Semester studieren – das Prädikat also eine geringe Selektivität aufweist. Deshalb fällt diese Möglichkeit „dem Pruning zum Opfer", was in der Tabelle durch das Durchstreichen notiert ist.

Analoge Zugriffsmöglichkeiten sind für die anderen beiden Basisrelationen *hören* und *Vorlesungen* gezeigt. Nur bei *Vorlesungen* ist ein *Index-Seek* eine sinnvolle Variante, falls wenige Vorlesungen das Prädikat *SWS=2* erfüllen. Die drei zweielementigen Teilmengen der drei Basisrelationen werden in der nächsten Phase des

Algorithmus bearbeitet, indem jeweils die besten Zugriffspläne mit einem Join oder einem Kreuzprodukt (falls es kein Joinprädikat gibt) verknüpft werden. Letztendlich wird die drei-elementige Gesamtlösung generiert, indem alle Basisrelationen mit den zwei-elementigen komplementären Teillösungen und dann alle zwei-elementigen Teillösungen mit der noch fehlenden Basisrelation „versucht" werden.

8.6 Übungen

8.1 Beweisen oder widerlegen Sie folgende Äquivalenzen:

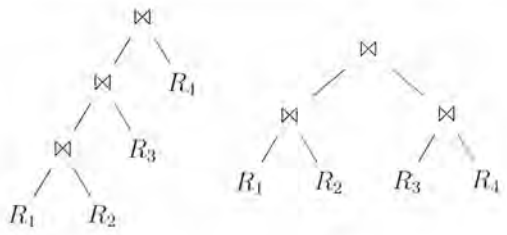
- $\sigma_{p_1 \wedge p_2 \wedge \ldots \wedge p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots(\sigma_{p_n}(R))\ldots))$
- $\sigma_p(R_1 \bowtie R_2) = \sigma_p(R_1) \bowtie R_2$ (falls p nur Attribute aus $\mathcal{R}_1$ enthält)
- $\Pi_l(R_1 \cap R_2) = \Pi_l(R_1) \cap \Pi_l(R_2)$
- $\Pi_l(R_1 \cup R_2) = \Pi_l(R_1) \cup \Pi_l(R_2)$
- $\Pi_l(R_1 - R_2) = \Pi_l(R_1) - \Pi_l(R_2)$

8.2 Überlegen Sie, wie der Semi-Join bei der algebraischen Optimierung eingesetzt werden könnte. Inwieweit wirkt sich die Verwendung von Semi-Joins auf das Einführen von Projektionen aus? Konzipieren Sie einen effizienten Auswertungsalgorithmus für Semi-Joins.

8.3 In Abschnitt 8.1.2 wurde eine sehr einfache Heuristik zur Bestimmung einer Anordnung der Joins eines algebraischen Ausdruckes vorgestellt. Mit dieser Heuristik werden allerdings nur Reihenfolgen von Joins berücksichtigt und nicht allgemeine Anordnungen. Es kann nicht passieren, dass das rechte Argument eines Joins aus einem anderen Join entstanden ist. Solche Auswertungspläne nennt man *Left-Deep Tree*. Allgemeine Auswertungspläne, von denen es natürlich wesentlich mehr gibt, nennt man *Bushy Trees*. Abbildung 8.40 zeigt ein Beispiel mit den abstrakten Relationen R_1, R_2, R_3 und R_4.

- Bestimmen Sie die Anzahl der möglichen Left-Deep Trees bzw. Bushy Trees für einen gegebenen algebraischen Ausdruck mit n Relationen, der nur Joinoperationen enthält.
- Diskutieren Sie, inwieweit Bushy-Trees effizientere Auswertungspläne bezüglich der Größe der Zwischenergebnisse liefern können. Ist es sinnvoll, Bushy Trees bei der Suche nach einem effizienten Auswertungsplan zu berücksichtigen, wenn man die Anzahl der Möglichkeiten in Betracht zieht?

8.4 Warum wurde immer die größere Relation als Probe Input beim Hash-Join verwendet? Warum wäre es in einer realen Umsetzung von dem Beispiel in Abbildung 8.24 keine gute Idee, die Hashfunktion nach dem Alter sortieren zu lassen?



Left-Deep Tree Bushy Tree

Abbildung 8.40: Zwei Klassen von Auswertungsplänen

8.5 Wenn der Wertebereich des Joinattributs bekannt ist, kann ein Bitvektor angelegt werden, in dem für jeden vorkommenden Wert in einer Relation eine Markierung gesetzt wird. Beschreiben Sie einen Algorithmus, in dem diese Methode zur Verbesserung des Hash-Joins eingesetzt wird.

8.6 In diesem Kapitel wurden nur Methoden für Equijoins vorgestellt, d.h. solchen Joins, bei denen auf Gleichheit getestet wird. Formulieren Sie die Joinimplementierungen so um, dass Sie auch für die Vergleichsoperatoren „$<$“, „$>$“ und „$\neq$“ funktionieren. Ist dies in allen Kombinationen möglich?

8.7 Erläutern Sie, warum die Kostenformel für den verfeinerten Nested-Loop Join minimal wird, wenn $k = 1$ und R die kleinere der beiden Relationen ist. Diese Aussage gilt übrigens nur für realistische (große) Größen des Systempuffers und der Relationen – wie Stohner und Kalinski (1998) nachgewiesen haben.

8.8 In Abbildung 8.26 wurde ein mehrstufiger Mischvorgang gezeigt. Generell sollte man in Zwischenphasen so wenige Daten wie nötig mischen. Konzipieren Sie den Auswahlalgorithmus, der darauf basiert Dummy-Läufe einzuführen und jeweils die kleinsten Läufe zuerst behandelt.

8.9 Geben Sie eine Pseudocode-Implementierung des Replacement-Selection an. Spezifizieren Sie den *Sort*-Iterator in Pseudocode.

8.10 Finden Sie eine Implementierung des Divisionsoperators. Eine einfache Möglichkeit besteht in einer geeigneten Sortierung von Dividend und Divisor und anschließendem wiederholten Durchlaufen des Divisors. Eine Alternative besteht im Anlegen einer Hashtabelle für den Divisor und den Quotienten.

8.11 Diskutieren Sie, inwieweit sich Integritätsbedingungen für die Anfrageauswertung ausnutzen lassen. Betrachten Sie u.a. auch den oben implementierten Divisionsoperator.

8.12 Geben Sie eine Kostenabschätzung für die Anzahl der Seitenzugriffe bei der Durchführung eines Hash-Joins an (abhängig von der Anzahl der Seiten der Eingaberelationen b_R und b_S sowie der reservierten Seiten im Hauptspeicher m).

8.13 Führen Sie eine Kostenabschätzung für die Ausdrücke in Abbildung 8.6 und 8.7 durch. Gehen Sie davon aus, dass die Selektion durch den Select-Operator implementiert wird und die Joins durch den verfeinerten Nested-Loop Join. Verwenden Sie folgende Parameter:

- Relationengrößen
  - $|p| = 800$
  - $|s| = 38000$
  - $|v| = 2000$
  - $|h| = 60000$
- durchschnittliche Tupelgrößen
  - p: 50 Bytes
  - s: 50 Bytes
  - v: 100 Bytes
  - h: 16 Bytes
- Selektivitäten
  - Join von s und h: $sel_{sh} = 2{,}6 \cdot 10^{-5}$
  - Join von h und v: $sel_{hv} = 5 \cdot 10^{-4}$
  - Join von v und p: $sel_{vp} = 1{,}25 \cdot 10^{-3}$
  - Selektion auf p: $sel_p = 1{,}25 \cdot 10^{-3}$
- Seitengröße 1024 Bytes
- Hauptspeicher 20 Seiten

8.14 Zeigen Sie anhand der Anfrage „*Finde alle Flüge von Berlin nach NY mit einmaligem Umsteigen*", dass ein "bushy" Anfrageauswertungsplan deutlich besser sein kann als ein links-tiefer Plan.

8.15 Man beweise die Korrektheit der Transformationsregeln für die Entschachtelung korrelierter Unteranfragen.

8.16 Man wende die Entschachtelungstechniken auf einigen selbst formulierten SQL-Anfragen mit korrelierten Unteranfragen an.

8.7 Literatur

Die Anfrageoptimierung ist immer noch ein sehr „heißes" Thema in der Datenbankforschung, leider gibt es aber keine aktuellen Übersichtsartikel. Ein „Klassiker" ist der Computing Surveys Artikel von Jarke und Koch (1984). Eine Übersicht über fortgeschrittene Techniken im Bereich Anfrageoptimierung findet sich in dem Sammelband von Freytag, Maier und Vossen (1994). Mitschang (1995) ist ein Buch zur Anfrageoptimierung – mit einem Schwerpunkt auf objekt-relationale Systeme. Lockemann und Dittrich (2002) haben ein Lehrbuch über Datenbankarchitekturen verfasst. Zur Auswertung von Anfragen gibt es einen sehr detaillierten Überblick von Graefe (1993), der auch viele praktische Probleme erläutert. Diesem Artikel ist

das Iteratorkonzept entnommen. Mishra und Eich (1992) diskutieren Auswertungstechniken für Joins. Shapiro (1986) stellt verschiedene Hash-Join-Verfahren vor.

Die ersten Techniken für Anfrageoptimierung findet man im INGRES Optimierer, beschrieben von Wong und Youssefi (1976). Er beruht auf der Zerlegung von Anfragegraphen. Selinger et al. (1979) veröffentlichten ein wegweisendes Papier über den System R Optimierer, in dem physische Eigenschaften von Relationen zur Auswahl einer günstigen Joinoperation ausgenutzt werden. Die Optimierung basiert auf Dynamic Programming, wobei alle Alternativen bewertet werden. Kossmann und Stocker (2000) haben darauf aufbauend eine Heuristik namens *iterative dynamic programming (IDP)* entwickelt, die bei komplexen Anfragen zum Einsatz kommen kann. Moerkotte und Neumann (2008) haben Optimierungsstrategien für die Enumeration der Anfrageplan-Alternativen in der dynamischen Programmierung entwickelt.

Die in diesem Kapitel beschriebene logische Optimierung korrelierter Unteranfragen wurde von Neumann und Kemper (2015) entwickelt. Es ist die erste Arbeit, die den Anspruch erhebt, alle Arten von korrelierten Unteranfragen effektiv entschachteln zu können. Das Verfahren wurde in das an der TUM von Kemper und Neumann (2011) konzipierte Datenbanksystem HyPer integriert.

Häufig werden regelbasierte Systeme eingesetzt, um viele Heuristiken in einem einheitlichen Rahmen verwenden zu können, was z.B. von Freytag (1987), Lohman (1988), Becker und Güting (1992) oder Lehnert (1988) erläutert wird. Den regelbasierten Optimierer von Starburst beschreiben Haas et al. (1990). Grust et al. (1997) verfolgen in ihrem *CROQUE*-Projekt den Ansatz, Anfragen auf der Basis von Monoid-Kalkülen zu optimieren. Dieser auf funktionaler Programmierung basierende Ansatz wird von Grust und Scholl (1999) detailliert beschrieben.

Berichte über generische Optimiererarchitekturen wurden von Kemper, Moerkotte und Peithner (1993) und Graefe und DeWitt (1987) verfasst. Kemper, Moerkotte und Peithner (1993) verwenden einen Blackboardansatz, der eine flexible und effiziente Optimierung auch unter Zeitbeschränkungen ermöglicht. Der Nachfolger des von Graefe und DeWitt (1987) beschriebenen Optimierergenerator Volcano, wird von Graefe und McKenna (1993) erläutert.

In diesem Kapitel konnten einige wichtige Themen nicht behandelt werden. Einen Überblick über Methoden zur Bestimmung von Joinreihenfolgen gibt Swami (1989). Interessante Techniken sind das sogenannte Simulated Annealing [Ioannidis und Wong (1987)], der KBZ-Algorithmus [Krishnamurthy, Boral und Zaniolo (1986)], der eine Teilklasse des Problems effizient lösen kann, und dessen Verallgemeinerung in [Swami und Iyer (1993)]. Eine vergleichende Bewertung verschiedener heuristischer Verfahren zur Bestimmung von (guten) Joinreihenfolgen wurde von Steinbrunn, Moerkotte und Kemper (1997) durchgeführt. Ibaraki und Kameda (1984) zeigen, dass die Bestimmung der Joinreihenfolge NP-hart ist. Cluet und Moerkotte (1995) und Scheufele und Moerkotte (1997) haben weitergehende Komplexitätsanalysen für Anfrageauswertungspläne mit Join- und Kreuzproduktoperatoren durchgeführt. Für die Optimierung von Anfragen mit Disjunktionen bieten sich sogenannte Bypass-Techniken an, wie sie von Kemper et al. (1994) und Steinbrunn et al. (1995) beschrieben wurden. Die Ausnutzung ähnlicher Unteranfragen für die Optimierung wird von Zhou et al. (2007) untersucht.

Bercken, Schneider und Seeger (2000) haben einen generischen Algorithmus für

die Join-Auswertung entwickelt. Dies wurde von Bercken et al. (2001) zu einer umfassenderen Java-Bibliothek von Algorithmen ausgebaut. Becker, Hinrichs und Finke (1993) hat eine Joinauswertung mit der mehrdimensionalen Indexstruktur Grid-File entwickelt. Güting et al. (2000) haben Verfahren für die Anfragebearbeitung auf so genannten beweglichen (hochgradig dynamischen Daten-) Objekten entworfen. Der G-Join wurde von Graefe (2011) erfunden. Man verspricht sich vom G-Join eine höhere Robustheit gegenüber Schieflagen der Datenverteilung, die beim Hash-Join zu unterschiedlich großen Partitionen führen könnten. Eine (entfernt) verwandte Join-Technik ist der progressive Merge-Join von Dittrich et al. (2002), bei der die frühzeitige Berechnung der ersten Ergebnisse im Vordergrund steht.

Helmer und Moerkotte (1997) haben spezielle Joinverfahren entwickelt, die anwendbar sind, wenn das Joinprädikat auf einem Mengenvergleich beruht. Dies kommt insbesondere in den objektorientierten und den sogenannten objektrelationalen Datenmodellen vor, da diese mengenwertige Attribute zulassen. Augsten et al. (2014) haben Techniken für die effiziente Auswertung von Ähnlichkeitsjoins („similarity joins") entwickelt. Claussen et al. (1997) behandeln die Optimierung von Anfragen mit Allquantifizierung. Carey und Kossmann (1997) zeigen Techniken für die Optimierung von Anfragen, bei denen die Datenbankbenutzer nur an den ersten N Ergebnistupeln interessiert sind. Eine interaktive Technik für die Ermittlung von ähnlichen Datenobjekten wurde von Waas, Ciaccia und Bartolini (2001) entwickelt. Kraft et al. (2003) haben Techniken für die Optimierung komplexer SQL-Ausdrücke – basierend auf „Query Rewriting" – entwickelt.

Christodoulakis (1983) gibt ein Verfahren zur Selektivitätsabschätzung mit Hilfe einer Verteilungsfunktion an. Lynch (1988) schlägt eine Möglichkeit zur Auswahl des Abschätzungsverfahrens durch den Benutzer vor und betrachtet zusätzlich nichtnumerische Schlüssel. Muralikrishna und DeWitt (1988) führen mehrdimensionale Equi-Depth-Histogramme als Verbesserung der Standardhistogramme ein. Poosala et al. (1996) beschreiben Histogramme zur Abschätzung der Ergebniskardinalität von Bereichsanfragen. Dieses Verfahren ist im Rahmen der DB2-Entwicklung bei IBM konzipiert worden. DB2 ist eines der wenigen derzeit verfügbaren DBMS-Produkte, das eine präzise Selektivitätsabschätzung mittels Histogrammen durchführt. Viele andere Produkte legen einfach eine Gleichverteilung der Attributwerte zugrunde – was natürlich zu fehlerhaften Abschätzungen und möglicherweise zu schlechten Auswertungsplänen führen kann. Lipton, Naughton und Schneider (1990) beschreiben ein adaptives Stichprobenverfahren.

9. Transaktionsverwaltung

Das Konzept der *Transaktion* wird oftmals als einer der größten Beiträge der Datenbankforschung für andere Informatikbereiche – z.B. Betriebssysteme und Programmiersprachen – angesehen. Unter einer Transaktion versteht man die „Bündelung" mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen als *Einheit* fehlerfrei ausgeführt werden sollen.

In diesem Kapitel werden wir die Eigenschaften von Transaktionen und die daraus ableitbaren Realisierungsanforderungen diskutieren. Wir werden sehen, dass es zwei grundlegende Anforderungen gibt:

1. Recovery, d.h. die Behebung von eingetretenen, oft unvermeidbaren Fehlersituationen.

2. Synchronisation von mehreren gleichzeitig auf der Datenbank ablaufenden Transaktionen.

Im nächsten Kapitel werden Realisierungsstrategien für die Fehlertoleranz und Fehlerbehebung (Recovery) diskutiert. In Kapitel 11 werden dann die Konzepte zur Mehrbenutzersynchronisation vorgestellt.

9.1 Begriffsbildung

Aus der Sicht des Datenbankbenutzer ist eine Transaktion eine *Arbeitseinheit* in einer Anwendung, die eine bestimmte Funktion erfüllt. Auf der Ebene des Datenbankverwaltungssystems sind natürlich derartige abstrakte Konzepte wie Arbeitseinheit unbekannt. Auf dieser Ebene stellt eine Transaktion eine Folge von Datenverarbeitungsbefehlen (lesen, verändern, einfügen, löschen) dar, die die Datenbasis von einem konsistenten Zustand in einen anderen – nicht notwendigerweise unterschiedlichen – konsistenten Zustand überführt. Das wesentliche dabei ist, dass diese Folge von Befehlen (logisch) ununterbrechbar, d.h. atomar ausgeführt wird.

Wir wollen diese abstrakten Begriffe an einem klassischen Transaktionsbeispiel erläutern. Dazu betrachten wir eine typische Transaktion in einer Bankanwendung: den Transfer von 50,– Euro von Konto A nach Konto B. Diese Transaktion besteht aus mehreren elementaren Operationen:

1. Lese den Kontostand von A in die Variable a: **read**(A,a);

2. Reduziere den Kontostand um 50,– Euro: $a := a - 50$;

3. Schreibe den neuen Kontostand in die Datenbasis: **write**(A,a);

4. Lese den Kontostand von B in die Variable b: **read**(B,b);

5. Erhöhe den Kontostand um 50,- Euro: $b := b + 50$;

6. Schreibe den neuen Kontostand in die Datenbasis: **write**(B,b);

Es sollte einleuchten, dass diese Folge von Befehlen, die eine Transaktion darstellen, atomar, also ununterbrechbar auszuführen ist. Anderenfalls könnte der Fall eintreten, dass nach Ausführung von Schritt 3. das System (z.B. aufgrund eines Stromausfalls) „abstürzt", und deshalb Konto A um 50,- Euro reduziert wurde, ohne dass Konto B jemals erhöht wurde. Es muss also gelten, dass entweder *alle* Befehle einer Transaktion ausgeführt werden oder gar keiner. Bei einer unkontrollierten Unterbrechbarkeit einer Transaktion kann es zu schwerwiegenden Konsistenzverletzungen aufgrund anderer parallel ablaufender Transaktionen kommen. Weiterhin wird die Konsistenzerhaltung gefordert. D.h., eine Transaktion beginnt mit einem konsistenten Zustand der Datenbasis und hinterlässt auch wieder einen konsistenten Zustand. Bezogen auf unser Beispiel wäre denkbar, dass es folgende Konsistenzbedingung gibt: Die Summe der Kontostände aller Konten eines Kunden darf den Dispositionskredit D nicht überschreiten. Bezogen auf unser Beispiel wäre es denkbar, dass beide Konten A und B einem Kunden gehören. Dann könnte nach Ausführung von Schritt 3. diese Konsistenzbedingung verletzt sein. Das ist durchaus zulässig, solange bei Abschluss der Transaktion die Konsistenz wiederhergestellt ist – was unter diesen Annahmen auf jeden Fall gewährleistet ist.

Wenn aber die beiden Konten unterschiedlichen Kunden gehören, könnte der Transfer zu einer Konsistenzverletzung führen, da durch die Reduzierung von A um 50,- Euro der entsprechende Dispositionskredit überschritten sein könnte. In dem Fall muss die gesamte Transaktion „ausgesetzt" werden – aus Sicht der Bank ist es natürlich nicht wünschenswert, den Kontostand von A beizubehalten, wenn gleichzeitig B erhöht wird.

9.2 Anforderungen an die Transaktionsverwaltung

Um den Durchsatz des Systems zu erhöhen, muss die Transaktionsverwaltung in der Lage sein, mehrere – i.A. sehr viele – gleichzeitig (nebenläufig) ablaufende Transaktionen zu verarbeiten. Dazu ist natürlich eine Synchronisation notwendig, die die anderenfalls durch unkontrollierte Nebenläufigkeit möglicherweise verursachten Konsistenzverletzungen ausschließt.

Weiterhin stellen Datenbanken i.A. einen ungeheuren Wert für die Unternehmen dar. Deshalb müssen Datenbanken gegen Soft- und Hardwarefehler geschützt werden. Diese Fehlertoleranz ist transaktionsorientiert durchzuführen: Abgeschlossene Transaktionen müssen auch nach einem Fehler hinsichtlich ihrer Wirkung erhalten bleiben, und noch nicht abgeschlossene Transaktionen müssen vollständig revidiert (zurückgesetzt) werden.

9.3 Operationen auf Transaktions-Ebene

Wie schon beschrieben, besteht eine Transaktion aus einer Folge von elementaren Operationen. Aus der „Sicht" des Datenbanksystems handelt es sich hierbei um die

Operationen **read** und **write**. Für die Steuerung der Transaktionsverarbeitung sind zusätzlich noch Operationen auf der Transaktionsebene notwendig:

- **begin of transaction (BOT)**: Mit diesem Befehl wird der Beginn einer eine Transaktion darstellenden Befehlsfolge gekennzeichnet.
- **commit**: Hierdurch wird die Beendigung der Transaktion eingeleitet. Alle Änderungen der Datenbasis werden durch diesen Befehl *festgeschrieben*, d.h. sie werden dauerhaft in die Datenbank eingebaut.
- **abort**: Dieser Befehl führt zu einem Selbstabbruch der Transaktion. Das Datenbanksystem muss sicherstellen, dass die Datenbasis wieder in den Zustand zurückgesetzt wird, der vor Beginn der Transaktionsausführung existierte.

In den klassischen Transaktionssystemen gibt es nur diese drei Befehle, die ja auch ausreichen, wenn man eine Transaktion als atomare Einheit betrachtet. In neueren Datenbankanwendungen – wie z.B. technischen Entwurfsvorgängen – sind Transaktionen jedoch von langer Dauer. Deshalb ist es dort sinnvoll, zwischenzeitlich Sicherungspunkte setzen zu können, auf die die laufende Transaktionsverarbeitung zurückgesetzt werden kann. Hierzu sind die beiden folgenden Befehle notwendig:

- **define savepoint**: Hierdurch wird ein Sicherungspunkt definiert, auf den sich die (noch aktive) Transaktion zurücksetzen lässt. Das DBMS muss sich dazu alle bis zu diesem Zeitpunkt ausgeführten Änderungen an der Datenbasis „merken". Diese Änderungen dürfen aber noch nicht in der Datenbasis festgeschrieben werden, da die Transaktion durch ein **abort** immer noch gänzlich aufgegeben werden kann.
- **backup transaction**: Dieser Befehl dient dazu, die noch aktive Transaktion auf den jüngsten – also den zuletzt angelegten – Sicherungspunkt zurückzusetzen. Es hängt von der Funktionalität des Systems ab, ob auch ein Rücksetzen auf weiter zurückliegende Sicherungspunkte möglich ist. Um diese Funktionalität zu realisieren, benötigt man selbstverständlich entsprechend mehr Speicherkapazität, um die Zustände mehrerer Sicherungspunkte temporär abzuspeichern – oder wie wir in Kapitel 10 sehen werden, mehr Zeit, um die ausgeführten Operationen rückgängig zu machen.

9.4 Abschluss einer Transaktion

Wie oben bereits angedeutet, gibt es zwei Möglichkeiten für den Abschluss einer Transaktion:

1. den erfolgreichen Abschluss durch ein **commit** und
2. den erfolglosen Abschluss durch ein **abort**.

Im ersten Fall wird eine Folge von elementaren Operationen durch die **BOT**- (**begin of transaction**) und **commit**-Befehle „geklammert":

BOT
op_1
op_2
⋮
op_n
commit

Für die Transaktionsverwaltung sind nur die Interaktionen mit der Datenbank relevant, d.h. alle Operationen auf z.B. lokalen Variablen sind in dieser Hinsicht nicht von Interesse. Wir werden uns deshalb in den nachfolgenden Diskussionen (Kapitel 10 und 11) nur mit den Befehlen **read** und **write** beschäftigen.

Für den erfolglosen Abschluss von Transaktionen mag es zwei Gründe geben: Zum einen könnten die Benutzer – d.h. die Transaktionen – selbst den Abbruch der noch aktiven Transaktion veranlassen. Dies geschieht explizit durch den Befehl **abort**:

BOT
op_1
op_2
⋮
op_j
abort

Der Grund für diesen Abbruch ist aus Sicht der Transaktionsverwaltung irrelevant. Die Transaktionsverwaltung muss gewährleisten, dass in der Datenbank der Zustand wieder „restauriert" wird, der vor Ausführung der ersten Operation op_1 existierte. Dieses Zurücksetzen der Transaktion nennt man auch „*rollback*".

Es gibt auch den Fall des außengesteuerten – also nicht „freiwilligen" – Zurücksetzens einer Transaktion:

BOT
op_1
op_2
⋮
op_k
~~~~~~ Fehler

Nach Ausführung des Befehls op_k tritt z.B. irgendein Fehler auf, der die weitergehende Bearbeitung der Transaktion unmöglich macht. In diesem Fall müssen die durch $op_1, \ldots, op_k$ getätigten Änderungen der Datenbasis rückgängig gemacht werden. Es gibt verschiedene Möglichkeiten für einen derartigen Fehler: Hardwarefehler, Stromausfall, Fehler im Programmcode der Transaktion oder auch eine aufgedeckte Verklemmung (Deadlock), die durch das Zurücksetzen dieser Transaktion vom Transaktionsverwalter gelöst werden soll. Es kann auch vorkommen, dass eine Transaktion nach Abarbeitung aller Operatoren wegen Verletzung von Konsistenzbedingungen zurückgesetzt werden muss. Diese dürfen – wie oben bereits ausgeführt – während der Transaktionsverarbeitung (teilweise) verletzt werden; aber bei Beendigung der
~~~~~~

Transaktion müssen alle auf der Datenbank definierten Konsistenzbedingungen erfüllt sein. Wenn dies nicht der Fall ist, muss die gesamte Transaktion zurückgesetzt werden.

9.5 Eigenschaften von Transaktionen

Wir haben mit der vorhergehenden Diskussion (hoffentlich) schon ein intuitives Verständnis des Transaktionsbegriffs erzielt. Die Eigenschaften des Transaktionskonzepts werden oft unter der Abkürzung *ACID* zusammengefasst. Das sogenannte *ACID-Paradigma* steht dabei für vier Eigenschaften:

Atomicity (Atomarität) Diese Eigenschaft verlangt, dass eine Transaktion als kleinste, nicht mehr weiter zerlegbare Einheit behandelt wird, d.h. entweder werden alle Änderungen der Transaktion in der Datenbasis festgeschrieben oder gar keine. Man kann sich dies auch als „alles-oder-nichts"-Prinzip merken.

Consistency Eine Transaktion hinterlässt nach Beendigung einen konsistenten Datenbasiszustand. Anderenfalls wird sie komplett (siehe *Atomarität*) zurückgesetzt. Zwischenzustände, die während der TA-Bearbeitung entstehen, dürfen inkonsistent sein, aber der resultierende Endzustand muss die im Schema definierten Konsistenzbedingungen (z.B. referentielle Integrität) erfüllen.

Isolation Diese Eigenschaft verlangt, dass nebenläufig (parallel, gleichzeitig) ausgeführte Transaktionen sich nicht gegenseitig beeinflussen. Jede Transaktion muss – logisch gesehen – so ausgeführt werden, als wäre sie die einzige Transaktion, die während ihrer gesamten Ausführungszeit auf dem Datenbanksystem aktiv ist. Mit anderen Worten, alle anderen parallel ausgeführten Transaktionen bzw. deren Effekte dürfen nicht sichtbar sein.

Durability (Dauerhaftigkeit) Die Wirkung einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank erhalten. Die Transaktionsverwaltung muss sicherstellen, dass dies auch nach einem Systemfehler (Hardware oder Systemsoftware) gewährleistet ist. Die einzige Möglichkeit, die Wirkungen einer einmal erfolgreich abgeschlossenen Transaktion ganz oder teilweise aufzuheben, besteht darin, eine andere sogenannte kompensierende Transaktion auszuführen.

Die Transaktionsverwaltung besteht aus zwei „großen" Komponenten: der *Mehrbenutzersynchronisation* und der *Recovery*. Die Aufgabe der Recovery besteht i.A. darin, die Atomarität und die Dauerhaftigkeit zu gewährleisten. Wir wollen dies an Abbildung 9.1 erläutern. In diesem Schaubild sind zwei Transaktionen T_1 und T_2 gezeigt, deren Ausführung zum Zeitpunkt t_1 bzw. t_2 beginnt. Aufgrund einer Fehlersituation kommt es zum Zeitpunkt t_3 zu einem Systemabsturz. Die Recovery-Komponente muss nach Wiederanlauf folgendes sicherstellen:

1. Die Wirkungen der zum Zeitpunkt t_3 abgeschlossenen Transaktion T_1 müssen in der Datenbasis vorhanden sein.

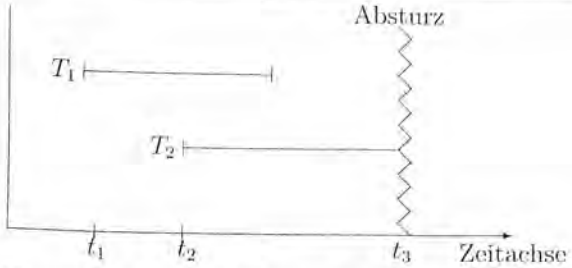


Abbildung 9.1: Transaktionsbeginn und -ende relativ zu einem Systemabsturz

2. Die Wirkungen der zum Zeitpunkt des Systemabsturzes noch nicht abgeschlossenen Transaktion T_2 müssen vollständig aus der Datenbasis entfernt sein. Diese Transaktion kann man nur durch ein erneutes Starten durchführen.

Die Aufgabe der Mehrbenutzersynchronisation besteht darin, die *Isolation* von parallel ablaufenden Transaktionen zu gewährleisten. Dazu müssen Mehrbenutzerkontrollkonzepte realisiert werden, die die Beeinflussung einer Transaktion durch andere Transaktionen ausschließt. Logisch gesehen gewährleistet die Mehrbenutzersynchronisation einen „Ein-Benutzer"- bzw. „Eine-Transaktion"-Betrieb, indem jeder Transaktion vorgetäuscht wird, die gesamte Datenbasis alleine zu besitzen. Mit anderen Worten wird den Benutzern der Eindruck einer seriellen Ausführung der Transaktionen vermittelt. Unter der seriellen Transaktionsausführung versteht man, dass eine Transaktion nach der anderen ausgeführt wird.

9.6 Transaktionsverwaltung in SQL

In SQL-92 – dem aktuellen SQL-Standard – werden Transaktionen implizit begonnen. Es gibt also keinen **begin of transaction**-Befehl, sondern mit Ausführung der ersten Anweisung wird automatisch eine Transaktion begonnen. Eine Transaktion wird durch einen der folgenden Befehle abgeschlossen:

- **commit work**: Die in der Transaktion vollzogenen Änderungen werden – falls keine Konsistenzverletzungen oder andere Probleme aufgedeckt werden – festgeschrieben. Das Schlüsselwort **work** ist optional, d.h. das Transaktionsende kann auch einfach mit **commit** „befohlen" werden.

- **rollback work**: Alle Änderungen sollen zurückgesetzt werden. Anders als der **commit**-Befehl muss das DBMS die „erfolgreiche" Ausführung eines **rollback**-Befehls immer garantieren können.

Als Beispiel betrachte man die folgende Sequenz von SQL-Befehlen auf Basis des in Abbildung 5.4 gezeigten Universitätsschemas:

```
insert into Vorlesungen
    values (5275, 'Kernphysik', 3, 2141);
insert into Professoren
    values (2141, 'Meitner', 'C4', 205);
commit work
```

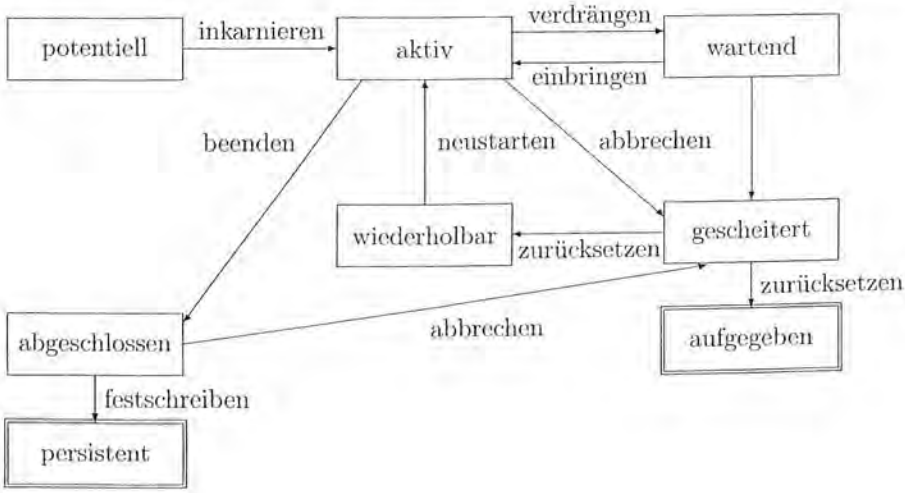


Abbildung 9.2: Zustandsübergangs-Diagramm für Transaktionen

Wegen der Integritätsbedingung für den Fremdschlüssel *gelesenVon* der Relation *Vorlesungen*, dürfte der **commit work**-Befehl nicht schon nach dem ersten **insert**-Befehl folgen. Dadurch wäre nämlich die referentielle Integrität der Datenbasis verletzt, weil es zu diesem Zeitpunkt ja noch keine Professorin namens Meitner mit *PersNr* 2141 in der *Professoren*-Relation gibt. Zwischenzustände der Datenbank während einer Transaktionsausführung dürfen aber sehr wohl inkonsistent sein – nur am Schluss der Transaktion muss die Konsistenz wiederhergestellt sein.

9.7 Zustandsübergänge einer Transaktion

In Abbildung 9.2 sind die möglichen Zustände und die Übergänge zwischen diesen Zuständen für Transaktionen dargestellt. Eine Transaktion befindet sich in einem der folgenden Zustände:

- *potentiell*: Die Transaktion ist codiert und „wartet darauf", in den Zustand *aktiv* zu wechseln. Diesen Übergang nennen wir *inkarnieren*.

- *aktiv*: Die aktiven (d.h. derzeit rechnenden) Transaktionen konkurrieren untereinander um die Betriebsmittel, wie z.B. Hauptspeicher, Rechnerkern zur Ausführung von Operationen, etc.

- *wartend*: Bei einer Überlast des Systems: (z.B. thrashing (Seitenflattern) des Puffers) kann die Transaktionsverwaltung einige aktive Transaktionen in den Zustand *wartend* verdrängen. Nach Behebung der Überlast werden diese wartenden Transaktionen sukzessive wieder eingebracht, d.h., wieder aktiviert.

- *abgeschlossen*: Durch den **commit**-Befehl wird eine aktive Transaktion beendet. Die Wirkung *abgeschlossener* TAs kann aber nicht gleich in der Datenbank

festgeschrieben werden. Vorher müssen noch möglicherweise verletzte Konsistenzbedingungen überprüft werden.

- *persistent*: Die Wirkungen abgeschlossener Transaktionen werden – wenn die Konsistenzerhaltung sichergestellt ist – durch *festschreiben* dauerhaft in die Datenbasis eingebracht. Damit ist die Transaktion *persistent*. Dies ist einer von zwei möglichen Endzuständen einer Transaktionverarbeitung.

- *gescheitert*: Transaktionen können aufgrund vielfältiger Ereignisse scheitern. Z.B. kann der Benutzer selbst durch ein **abort** eine aktive Transaktion abbrechen. Weiterhin können Systemfehler zum Scheitern aktiver oder wartender Transaktionen führen. Bei abgeschlossenen Transaktionen können auch Konsistenzverletzungen festgestellt werden, die ein Scheitern veranlassen.

- *wiederholbar*: Einmal gescheiterte Transaktionen sind u.U. *wiederholbar*. Dazu muss deren Wirkung auf die Datenbasis zunächst zurückgesetzt werden. Danach können sie durch Neustarten wiederum aktiviert werden.

- *aufgegeben*: Eine gescheiterte Transaktion kann sich aber auch als „hoffnungslos" herausstellen. In diesem Fall wird ihre Wirkung zurückgesetzt und die Transaktionsverarbeitung geht in den Endzustand *aufgegeben* über.

9.8 Literatur

Das Transaktionskonzept hat es zwar anscheinend schon früher gegeben; erstmals formalisiert wurde es aber in dem System R-Projekt am IBM-Forschungslabor San Jose von Eswaran et al. (1976). Gray (1981) gibt eine Retrospektive zu diesen Arbeiten und setzt sich mit den Grenzen des Transaktionskonzepts auseinander. Der Begriff ACID zur Charakterisierung von Transaktionen geht auf Härder und Reuter (1983) zurück – diese Arbeit war auch wegweisend für die Recoverykonzepte.

Das Konzept der Sicherungspunkte ist eine Vorform der geschachtelten Transaktionen, die von Moss (1985) systematisch behandelt werden. Walter (1984) hat diesen Ansatz zur Strukturierung komplexer Anwendungstransaktionen ausgenutzt.

Es gibt mittlerweile eine Reihe von Lehrbüchern zur Transaktionsverwaltung: Bernstein, Hadzilacos und Goodman (1987) ist eine zwar formale, aber dennoch sehr zugängliche Referenz. Das Buch von Papadimitriou (1986) ist sehr formal ausgerichtet. Gray und Reuter (1993) haben ein sehr umfangreiches Buch zur Realisierung der Transaktionskonzepte verfasst. Das Buch von Weikum und Vossen (2001) ist Pflichtlektüre aller Software-Entwickler, die sichere Mehrbenutzer-Informationssysteme realisieren wollen. Meyer-Wegener (1988) und Bernstein und Newcomer (1997) behandeln die Realisierung von Hochleistungs-Transaktionssystemen, insbesondere auch sogenannter *Transaction Processing Monitors*. Das Buch von Weikum (1988) behandelt Forschungs- und Realisierungsansätze zur Transaktionsverwaltung. Der Artikel von Schuldt et al. (2002) behandelt allgemein die Transaktionskonzepte für neuartige Anwendungen.

Wir werden zu den beiden grundlegenden Teilproblemen – nämlich Recovery und Mehrbenutzersynchronisation – noch zahlreiche Originalarbeiten zitieren.

10. Fehlerbehandlung

Die Datenbasis stellt im Allgemeinen einen immensen Wert für ein Unternehmen dar. Deshalb ist es unabdingbar, ihre Konsistenz auch im Fehlerfall wiederherstellen zu können. Ein vordringliches Ziel bei der Entwicklung eines Datenbankverwaltungssystems sollte es natürlich sein, Fehler, gleich welcher Art, (weitgehend) auszuschließen. Wie in jedem komplexen System lassen sich aber Fehler nie vollständig vermeiden – und selbst wenn es gelänge, das DBMS fehlerfrei zu codieren, so wären andere Komponenten (wie z.B. die Hardware) oder äußere Einflüsse (wie z.B. Bedienungsfehler, Feuer im Computerraum) unvermeidbare Fehlerquellen. Nach einem Systemfehler ist es die Aufgabe der *Recoverykomponente* des DBMS, den Wiederanlauf des Systems und die Rekonstruktion des *jüngsten* konsistenten Datenbasiszustands zu gewährleisten.

10.1 Fehlerklassifikation

Abhängig von der aufgetretenen Fehlersituation müssen unterschiedliche Recoverymechanismen eingesetzt werden. Wir unterscheiden grob drei Fehlerkategorien:

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

10.1.1 Lokaler Fehler einer Transaktion

In diese Kategorie fallen solche Fehler, die zwar zum Scheitern der jeweiligen Transaktion führen, aber den Rest des Systems hinsichtlich der Datenbankkonsistenz nicht beeinflussen. Typische Fehlerquellen sind:

- Fehler im Anwendungsprogramm,
- expliziter Abbruch (**abort**) der Transaktion durch den Benutzer, weil z.B. das gewünschte Ergebnis nicht zustandekommt,
- systemgesteuerter Abbruch einer Transaktion, um beispielsweise eine Verklemmung (engl. deadlock) zu beheben.

Diese lokalen Fehler werden behoben, indem alle Änderungen an der Datenbasis, die von dieser noch aktiven Transaktion verursacht wurden, rückgängig gemacht werden. Diesen Vorgang bezeichnet man auch als *lokales Undo*. Lokale Fehler treten relativ häufig auf und müssen deshalb sehr schnell und effizient behoben werden können. Mit anderen Worten, die Recoverykomponente sollte einen lokalen Fehler innerhalb weniger Millisekunden beheben können – sogar ohne dass das System für andere Transaktionen gesperrt werden muss.

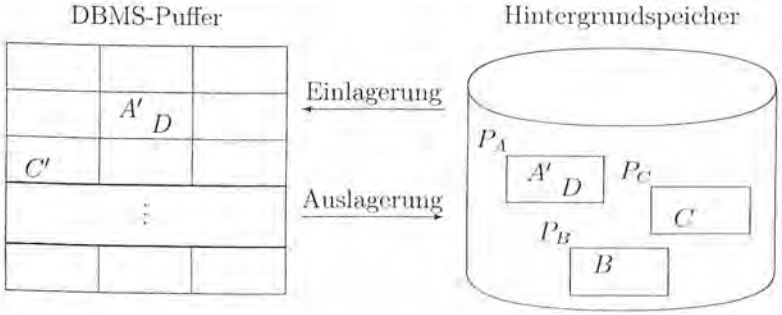


Abbildung 10.1: Schematische Darstellung der (zweistufigen) Speicherhierarchie

10.1.2 Fehler mit Hauptspeicherverlust

Ein Datenbankverwaltungssystem bearbeitet Daten innerhalb des sogenannten Datenbankpuffers. Der Puffer ist Teil des Hauptspeichers und ist in Seitenrahmen segmentiert, die jeweils genau eine Seite fassen können (vgl. Abschnitt 7.4). Diese Situation ist grafisch in Abbildung 10.1 beschrieben. Alle Datensätze (Tupel) – hier in unserem Beispiel abstrakt mit A, B, C und D bezeichnet – müssen auf Seiten abgebildet werden, die dauerhaft auf dem Hintergrundspeicher (in der materialisierten Datenbasis) gespeichert werden. Die Seiten sind hier mit P_A, P_B und P_C bezeichnet, wobei die Seite P_A neben dem Datum A auch noch das Datum D enthält – i.A. enthalten Seiten viele Datensätze. Beim Zugriff auf ein Datum, das sich (noch) nicht im Puffer befindet, muss die Seite, auf der sich das Datum befindet, *eingelagert* werden. Die Änderungen werden auf dieser Kopie der Seite im Puffer vollzogen. Sie werden dann „früher oder später" durch ein Rückkopieren der Pufferkopie auf den Hintergrundspeicher in die materialisierte Datenbasis eingebracht. In unserem Beispiel ist die Änderung des Datums A, die durch A' gekennzeichnet ist, bereits zurückgeschrieben, wohingegen die Änderung auf C noch nicht in den Hintergrundspeicher eingebracht ist.

Ein Problem ergibt sich dadurch, dass bei sehr vielen Fehlern (wie z.B. Stromausfall) der Inhalt des Puffers verlorengeht. Dadurch werden alle Änderungen an Daten vernichtet, die sich nur im Puffer aber noch nicht auf dem Hintergrundspeicher befinden. Das Transaktionsparadigma verlangt, dass

- alle durch nicht abgeschlossene Transaktionen schon in die materialisierte Datenbasis eingebrachten Änderungen rückgängig gemacht werden und
- alle noch nicht in die materialisierte Datenbasis eingebrachten Änderungen durch abgeschlossene Transaktionen nachvollzogen werden.

Den ersten Vorgang bezeichnet man als (globales) *Undo*, den zweiten Vorgang als (globales) *Redo*.

Diese Fehlerklasse geht also davon aus, dass die materialisierte Datenbasis nicht zerstört ist, sich aber nicht in einem transaktionskonsistenten Zustand befindet. Der transaktionskonsistente Zustand wird gerade durch das *Undo* und *Redo* wiederher-

gestellt. Dazu sind natürlich Zusatzinformationen aus einer sogenannten *Log-Datei* (Protokolldatei) notwendig.

Die Fehler dieser Fehlerklasse treten i.A. im Intervall von Tagen auf, da sie durch z.B. Stromausfall, Fehler im Betriebssystemcode, Hardwareausfall, etc. verursacht werden. Die Recoverydauer sollte hierbei in der Größenordnung von einigen Minuten liegen.

10.1.3 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 in Systemprogrammen (z.B. im Plattentreiber), die zu einem Datenverlust führen.

Obwohl solche Situationen im Durchschnitt sehr selten (etwa im Zeitraum von Monaten oder Jahren) auftreten, muss man **unbedingt** Vorkehrungen treffen, um die Datenbasis nach einem derartigen Fehler wieder in den jüngsten konsistenten Zustand bringen zu können. Dazu sind – wie wir später noch detaillierter ausführen werden – eine Archivkopie der materialisierten Datenbasis und ein Log-Archiv mit allen seit Anlegen dieser Datenbasis-Archivkopie vollzogenen Änderungen notwendig. Die beiden Archivkopien sollten natürlich räumlich getrennt von dem Plattenspeicher aufbewahrt werden, um z.B. bei Feuer nicht sämtliche Informationen zu verlieren.

10.2 Die Speicherhierarchie

In Abbildung 10.1 haben wir die zweistufige Speicherhierarchie bestehend aus dem DBMS-Puffer und dem Hintergrundspeicher mit der materialisierten Datenbasis skizziert. Wir wollen hier die Wechselwirkungen zwischen Transaktionsbearbeitung und der Ein- und Auslagerung von Datenseiten in den Puffer bzw. zurück auf den Hintergrundspeicher behandeln.

10.2.1 Ersetzung von Puffer-Seiten

Eine Transaktion benötigt im Allgemeinen mehrere Datenseiten, die sich entweder schon (zufällig) im Puffer befinden oder aber eingelagert werden müssen. Für die Dauer eines Zugriffs bzw. einer Änderungsoperation wird die jeweilige Seite im Puffer *fixiert*. Durch das Setzen eines *FIX*-Vermerks wird verhindert, dass die betreffende Seite aus dem Puffer verdrängt wird. Werden Daten auf dieser Seite durch die Operation geändert, wird die Seite als modifiziert („dirty") gekennzeichnet. In diesem Fall stimmt der im Puffer gehaltene Zustand der Seite nicht mehr mit dem Zustand der betreffenden Seite auf dem Hintergrundspeicher überein. Nach Beendigung der Operation wird der *FIX*-Vermerk wieder gelöscht. Dadurch ist diese Seite prinzipiell wieder als mögliches „Opfer" für die Ersetzung freigegeben.

Salopp ausgedrückt herrscht im Datenbankpuffer ein „Kommen und Gehen" von Seiten. Es gibt zwei Strategien in Bezug auf aktive, also noch nicht festgeschriebene Transaktionen:

- *¬steal*: Bei dieser Strategie wird die Ersetzung von Seiten, die von einer noch aktiven Transaktion modifiziert wurden, ausgeschlossen.
- *steal*: Jede nicht fixierte Seite ist prinzipiell ein Kandidat für die Ersetzung, falls neue Seiten eingelagert werden müssen.

Bei der *¬steal*-Strategie kann es nie vorkommen, dass Änderungen einer noch nicht abgeschlossenen Transaktion in die materialisierte Datenbasis übertragen werden. Bei einem *rollback* einer (natürlich noch) aktiven Transaktion braucht man sich also nicht um den Zustand des Hintergrundspeichers kümmern, da die Transaktionen dort vor dem **commit** keine Spuren hinterlassen können. Anders sieht es bei *steal* aus: In diesem Fall muss man bei einem *rollback* einer Transaktion, u.U. auch schon in die materialisierte Datenbasis eingebrachte Seiten, durch ein *Undo* in den vor Transaktionsbeginn existierenden Zustand zurücksetzen.

10.2.2 Einbringen von Änderungen einer Transaktion

Die von einer abgeschlossenen Transaktion verursachten Änderungen – d.h. alle von ihr modifizierten Seiten – werden unter der *force*-Strategie beim **commit** in die materialisierte Datenbasis übertragen (durch Kopieren der Seiten). Die mit *¬force* bezeichnete Strategie erzwingt diese Einbringung aller Änderungen nicht. Deshalb können Änderungen einer abgeschlossenen Transaktion verlorengehen, da sie nur im Systempuffer vorhanden waren und erst zu einem späteren Zeitpunkt, z.B. wenn die betreffende Seite sowieso ersetzt werden sollte, in die materialisierte Datenbasis eingebracht werden sollten. Aus diesem Grund benötigt man bei einer *¬force*-Pufferverwaltung andere Protokolleinträge aus einer separaten Log-Datei, um diese noch nicht in die Datenbasis propagierten Änderungen nachvollziehen (*Redo*) zu können. Bei Einhaltung der *force*-Strategie ist dies nicht notwendig, da die materialisierte Datenbasis immer alle Änderungen abgeschlossener Transaktionen enthï¿½lt.

Es wäre auf den ersten Blick verlockend, die Stategien *force* und *¬steal* zu kombinieren: Es erscheint, dass man dann *alle* Änderungen abgeschlossener Transaktionen und *keine* Änderungen noch aktiver Transaktionen in der materalisierten Datenbasis dauerhaft gespeichert hat. Es gibt aber viele Gründe, die gegen diese Systemkonfiguration sprechen: Zum einen ist die erzwungene Propagierung aller Änderungen zum Transaktionende sehr teuer. Es gibt Seiten, die von vielen Transaktionen benötigt werden und deshalb über längere Zeit im Puffer residieren. Solche sogenannten „hot spot"-Seiten würden somit nur zur Propagation der Änderungen in die Datenbasis kopiert, ohne dass sie im Puffer ersetzt werden. Dort wird lediglich gekennzeichnet, dass die Seiten danach – wahrscheinlich nur für kurze Zeit – unmodifiziert sind. Außerdem muss die Propagation, also das Kopieren der von der abgeschlossenen TA modifizierten Seiten, *atomar* (also „alles oder nichts") erfolgen. Das System darf nicht mitten im Kopiervorgang abstürzen und einen inkonsistenten Datenbasis-Zustand hinterlassen. Um das zu erzielen, ist ein zusätzlicher Aufwand nötig. Weiterhin können die Strategien *force* und *¬steal* nicht kombiniert werden, wenn Transaktionen

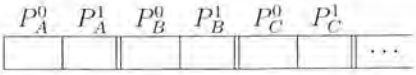


Abbildung 10.2: Twin-Block-Anordnung der Seiten P_A, P_B und P_C.

kleinere Objekte als ganze Seiten exklusiv bearbeiten (sperren) können – siehe dazu Übung 10.1 und das nachfolgende Kapitel über Mehrbenutzersynchronisation.

Wir fassen die vier Kombinationen von *force*/¬*force* und *steal*/¬*steal* hinsichtlich ihrer Anforderungen an die *Redo-* und *Undo*-Recovery wie folgt zusammen:

	force	¬force
¬steal	• kein Redo • kein Undo	• Redo • kein Undo
steal	• kein Redo • Undo	• Redo • Undo

10.2.3 Einbringstrategie

Unter der Einbringstrategie versteht man die Methodik, nach der Änderungen in die materialisierte Datenbasis propagiert werden. Die heute mit Abstand gängigste Methode heißt „*update-in-place*" und kann als *direkte* Einbringstrategie klassifiziert werden. Bei dieser Strategie wird jeder Seite genau ein Speicherplatz im Hintergrundspeicher zugeordnet. Wenn die Seite aus dem DBMS-Puffer verdrängt wird (und modifiziert war), wird sie direkt an diesen Speicherplatz kopiert, so dass der vorhergehende Zustand der Seite verlorengeht. Diese Vorgehensweise war auch in Abbildung 10.1 zugrundegelegt. Falls der vorhergehende Zustand im Rahmen eines *Undo* wiederhergestellt werden muss, benötigt man zusätzliche Protokollinformationen.

Bei den indirekten Einbringstrategien werden geänderte Seiten an einem separaten Platz gespeichert, und nur zu bestimmten, vom System initiierten Zeitpunkten, werden die alten Zustände durch die neuen ersetzt. Die einfachste Methode besteht darin, für jede Seite zwei Blöcke im Hintergrundspeicher freizuhalten. Die Situation ist für unser Beispiel in Abbildung 10.2 skizziert. Jeder Seite, wie z.B. P_A sind zwei Blöcke P_A^0 und P_A^1 zugeordnet. Es gibt ein globales Bit *aktuell*, das angibt, welche der Blöcke gerade aktuell ist. Also werden Änderungen jeweils nach $P_A^{aktuell}$, $P_B^{aktuell}$ und $P_C^{aktuell}$ geschrieben. Kommt es zu einem Fehler, kann das System sehr effizient auf die in $P_A^{\neg aktuell}$, $P_B^{\neg aktuell}$ und $P_C^{\neg aktuell}$ noch verfügbaren Zustände „zurückschalten". Durch dieses sogenannte Twin-Block-Verfahren wird die *atomare* Propagation des gesamten Pufferinhalts sehr gut unterstützt, da man zunächst alle modifizierten Seiten aus dem Puffer in ihre jeweiligen aktuellen Twin-Blöcke kopieren kann. Wenn dies erfolgt ist, wird das Bit *aktuell* auf den komplementären Wert gesetzt. Geht zwischenzeitlich, also während des Kopierens, etwas „schief", hat man immer noch die alten Zustände aller Seiten.

Das Twin-Block-Verfahren hat den großen Nachteil, dass der Speicherbedarf sich verdoppelt. Das *Schattenspeicherkonzept* leistet hier eine gewisse Abhilfe, da dabei nur die tatsächlich modifizierten Seiten verdoppelt werden. Aber es weist in der

Praxis etliche Nachteile auf, die seinen Einsatz fast immer ausschließen.

10.2.4 Hier zugrunde gelegte Systemkonfiguration

Die nachfolgende Diskussion der Recoverykomponente eines DBMS geht von der allgemeinsten und für die Recovery schwierigsten (und auch aufwendigsten) Konfiguration aus:

- *steal*: nicht-fixierte Seiten können jederzeit ersetzt bzw. auch nur propagiert (d.h. ohne Ersetzung zurückgeschrieben) werden.

- *¬force*: geänderte Seiten werden auf kontinuierlicher Basis in die Datenbasis propagiert, aber nicht notwendigerweise alle geänderten Seiten zum Ende einer Transaktion.

- *update-in-place*: Jede Seite hat einen Heimatplatz (Block) auf dem Hintergrundspeicher. Wird sie – auch vor dem **commit** einer Transaktion, die sie verändert hat – aus dem Puffer verdrängt, muss sie auf diesen Block kopiert werden.

- *Kleine Sperrgranulate*: Transaktionen können auch kleinere Objekte als eine komplette Seite exklusiv sperren und verändern. D.h., bezogen auf unser in Abbildung 10.1 dargestelltes Beispiel, kann eine Transaktion T_1 das Datum A auf Seite P_A verändern und eine parallele Transaktion T_2 könnte gleichzeitig das Datum D auch auf Seite P_A modifizieren. Das Problem aus der Sicht der Recoverykomponente besteht darin, dass eine Seite im Datenbankpuffer zu einem gegebenen Zeitpunkt sowohl Änderungen einer abgeschlossenen Transaktion als auch Änderungen einer noch nicht abgeschlossenen Transaktion enthalten kann.

Wir werden uns jetzt mit den Recoverykonzepten beschäftigen, die bei der oben beschriebenen Systemkonfiguration notwendig sind, um die Konsistenz der Datenbasis nach einem Fehlerfall wiederherzustellen.

10.3 Protokollierung von Änderungsoperationen

Die materialisierte Datenbasis enthält meist nicht den jüngsten konsistenten Zustand der Datenbasis – sie enthält i.A. nicht einmal einen konsistenten Zustand. Deshalb benötigt man Zusatzinformationen, die an anderer Stelle gespeichert werden als die Datenbasis – nämlich in einer sogenannten *Log*-Datei (oder auch *Protokolldatei* genannt). Wir haben im vorangegangenen Abschnitt gesehen, dass Änderungen noch nicht abgeschlossener Transaktionen in die materialisierte Datenbasis eingebracht werden können. Gleichzeitig können in der materialisierten Datenbasis auch Änderungen von bereits erfolgreich abgeschlossenen Transaktionen fehlen, da die modifizierten Seiten noch nicht aus dem Puffer in die Datenbasis propagiert wurden.

10.3.1 Struktur der Log-Einträge

Man benötigt für jede Änderungsoperation, die von einer Transaktion durchgeführt wird, zwei Protokollinformationen:

1. Die *Redo*-Information gibt an, wie die Änderung nachvollzogen werden kann.
2. Die *Undo*-Information beschreibt, wie die Änderung rückgängig gemacht werden kann.

Bei dem von uns vorgestellten Recoveryverfahren enthält jeder normale *Log*-Eintrag zusätzlich zur *Redo* und *Undo*-Information noch die folgenden Komponenten:

- *LSN* (*Log Sequence Number*), eine eindeutige Kennung des Log-Eintrags. Es wird verlangt, dass die *LSNs* monoton aufsteigend vergeben werden, so dass man die chronologische Reihenfolge der Protokolleinträge ermitteln kann.
- *Transaktionskennung TA* der Transaktion, die die Änderung durchgeführt hat.
- *PageID*, die Kennung der Seite, auf der die Änderungsoperation vollzogen wurde. Wenn eine Änderung mehr als eine Seite betrifft, müssen entsprechend viele Log-Einträge generiert werden.
- *PrevLSN*, einen Zeiger auf den vorhergehenden Log-Eintrag der jeweiligen Transaktion. Diesen Eintrag benötigt man aus Effizienzgründen.

10.3.2 Beispiel einer Log-Datei

Wir wollen jetzt die Log-Einträge für zwei parallel ablaufende Transaktionen T_1 und T_2 in Abbildung 10.3 demonstrieren. Die *Log-Einträge* für die **BOT**- und **commit**-Operationen haben eine besondere Struktur, da sie nur die *LSN*, die *TA* und den Operationsnamen enthalten. Der *PrevLSN*-Zeiger der **BOT**-Einträge ist auf 0 gesetzt, da es natürlich keinen vorhergehenden Eintrag zu der jeweiligen Transaktion geben kann. Mithilfe der *PrevLSN*-Zeiger kann man sehr effizient die *Log-Einträge* einer Transaktion in Rückwärtsrichtung durchlaufen.

Der Log-Eintrag mit der *LSN* #3 beispielsweise bezieht sich auf Transaktion T_1 und die Seite P_A. Wenn ein *Redo* durchgeführt werden muss, ist das Datum A auf Seite P_A um 50 zu erniedrigen (durch die der Sprache C angelehnte Notation A-=50 ausgedrückt). Wenn ein *Undo* auszuführen ist, muss A um 50 erhöht werden. Der vorhergehende Log-Eintrag hat die *LSN* #1.

10.3.3 Logische oder physische Protokollierung

In unserem Beispiel (siehe Abbildung 10.3) wurden die *Redo*- und die *Undo*-Informationen logisch protokolliert, d.h. es wurden jeweils Operationen angegeben. Eine andere Alternative besteht in der physischen Protokollierung. Dann wird für das *Undo* das sogenannte *Before-Image* und für das *Redo* das *After-Image* des Datenobjekts abgespeichert. Bezogen auf den Log-Eintrag #3 hätte man also den ursprünglichen Wert von A, sagen wir 1000, als *Undo*-Information, und den neuen Wert, nämlich 950, als *Redo*-Information abgespeichert.

Bei der logischen Protokollierung wird

Schritt	T_1	T_2	Log [LSN,TA,PageID,Redo,Undo,PrevLSN]
1.	**BOT**		$[\#1, T_1, \textbf{BOT}, 0]$
2.	$r(A, a_1)$		
3.		**BOT**	$[\#2, T_2, \textbf{BOT}, 0]$
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**		$[\#6, T_1, \textbf{commit}, \#5]$
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**	$[\#8, T_2, \textbf{commit}, \#7]$

Abbildung 10.3: Verzahnte Ausführung zweier Transaktionen und das erstellte Log

- das *Before-Image* durch Ausführung des *Undo*-Codes aus dem *After-Image* generiert und
- das *After-Image* durch Ausführung des *Redo*-Codes aus dem *Before-Image* berechnet.

Dazu ist es natürlich notwendig, dass man weiß (bzw. erkennen kann), ob das *Before-Image* oder das *After-Image* in der materialisierten Datenbasis enthalten ist. Hierzu dient die LSN: Beim Anlegen jedes neuen Log-Eintrags wird die neu generierte, eindeutige LSN in einen reservierten Bereich der betreffenden Seite geschrieben. Man beachte, dass die neu generierte LSN die derzeit größte LSN darstellt, da wir ein monotones Wachsen der LSNs verlangt haben. Wenn die Seite auf den Hintergrundspeicher propagiert wird, wird der derzeitige LSN-Eintrag dieser Seite mitkopiert. Daran kann man dann 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 war schon das *After-Image* bezüglich der protokollierten Änderungsoperationen auf den Hintergrundspeicher propagiert worden.

10.3.4 Schreiben der Log-Information

Bevor eine Änderungsoperation ausgeführt wird, muss der zugehörige Log-Eintrag angelegt werden. Bei physischer Protokollierung muss man das *Before-Image* vor

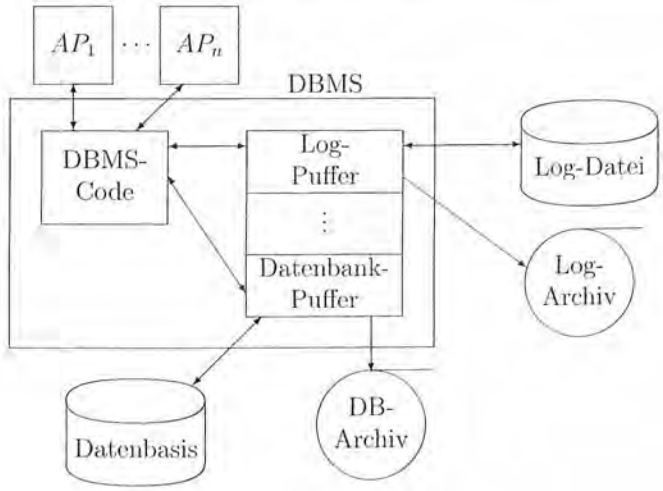


Abbildung 10.4: Speicherhierarchie eines Datenbanksystems [Härder und Reuter (1983)]

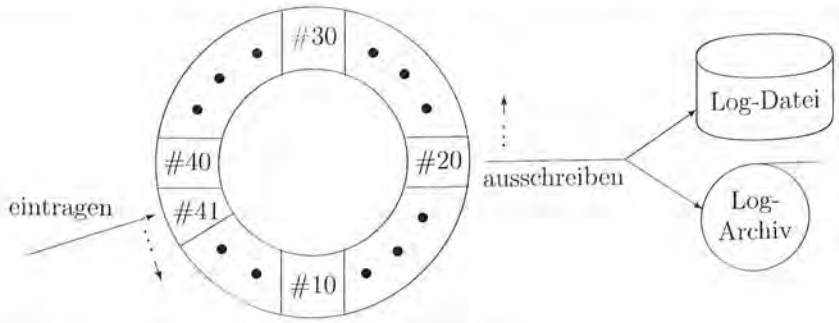


Abbildung 10.5: Anordnung des Log-Ringpuffers

Ausführung der Änderungsoperation und das *After-Image* nach Ausführung der Änderungsoperation in den Log-Record eintragen. Bei logischer Protokollierung kann man gleich beide Informationen – also *Redo-* und *Undo-*Code – in den Log-Record eintragen. Die Log-Einträge werden im sogenannten *Log-Puffer* im Hauptspeicher zwischengelagert. Diese Anordnung ist in Abbildung 10.4 schematisch gezeigt. Demnach befinden sich im Hauptspeicher getrennte Pufferbereiche für die Datenbankseiten und die Log-Einträge. Der Log-Puffer ist i.A. sehr viel kleiner als der Datenbankpuffer. *Spï¿½testens* sobald er voll ist, muss er auf den Hintergrundspeicher geschrieben werden. In modernen Datenbankarchitekturen ist der Log-Puffer als Ringpuffer organisiert: An einem Ende wird kontinuierlich ausgeschrieben und am anderen Ende kommen laufend neue Einträge hinzu. Dadurch kommt es zu einer gleichmäßigen Auslastung. Es wird also verhindert, dass man stoßweise sehr große Auslagerungsvorgänge bearbeiten muss, die die Transaktionsverarbeitung behindern würden. Die Anordnung des Log-Ringpuffers ist in Abbildung 10.5 gezeigt. Die Log-

Einträge werden gleich zweifach ausgeschrieben:

1. auf das temporäre Log und
2. auf das Log-Archiv.

Das temporäre Log befindet sich i.A. auf einem Plattenspeicher und wird online gehalten. Das Log-Archiv befindet sich normalerweise auf einem Magnetband, um es weitestgehend vor Hardwaredefekten zu schützen. Das Log-Archiv wird für die Recovery nach einem Hintergrundspeicherverlust benötigt – vergleiche dazu auch den Abschnitt 10.9.

10.3.5 Das WAL-Prinzip

Wir hatten gerade bemerkt, dass die Log-Einträge spätestens dann geschrieben werden müssen, wenn sich der zur Verfügung stehende Ringpuffer gefüllt hat. Bei der von uns zugrundegelegten Systemkonfiguration (¬*force*, *steal*, und *update-in-place*) ist aber zusätzlich unabdingbar, das sogenannte WAL-Prinzip (Write Ahead Log) einzuhalten. Dafür gibt es zwei Regeln, die *beide* befolgt werden müssen:

1. Bevor eine Transaktion festgeschrieben (**committed**) wird, müssen alle „zu ihr gehörenden" Log-Einträge ausgeschrieben werden.
2. Bevor eine modifizierte Seite ausgelagert werden darf, müssen alle Log-Einträge, die zu dieser Seite gehören, in das temporäre und das Log-Archiv ausgeschrieben werden.

Die erste Regel des WAL-Prinzips ist notwendig, um erfolgreich abgeschlossene Transaktionen nach einem Fehler nachvollziehen (*Redo*) zu können. Die zweite Regel wird benötigt, um im Fehlerfall die Änderungen nicht abgeschlossener Transaktionen aus den modifizierten Seiten der materialisierten Datenbasis entfernen zu können.

Beim WAL-Prinzip schreibt man natürlich alle Log-Einträge bis zu dem letzten notwendigen aus – d.h. man übergeht keine Log-Einträge, die von Regel 1. und 2. nicht erfasst sind. Dies ist essentiell, um die chronologische Reihenfolge der Log-Einträge im Ringpuffer zu wahren.

10.4 Wiederanlauf nach einem Fehler

Nach einem Fehler mit Verlust des Hauptspeicherinhalts hat man die in Abbildung 10.6 dargestellte Situation zu behandeln. Im Allgemeinen muss die Recovery natürlich viel mehr als zwei Transaktionen bearbeiten. T_1 und T_2 repräsentieren die zwei zu behandelnden Transaktionstypen:

- Transaktionen der Art T_1 müssen hinsichtlich ihrer Wirkung vollständig nachvollzogen werden. Transaktionen dieser Art nennt man *Winner*.
- Transaktionen, die wie T_2 zum Zeitpunkt des Absturzes noch aktiv waren, müssen rückgängig gemacht werden. Diese Transaktionen bezeichnen wir als *Loser*.

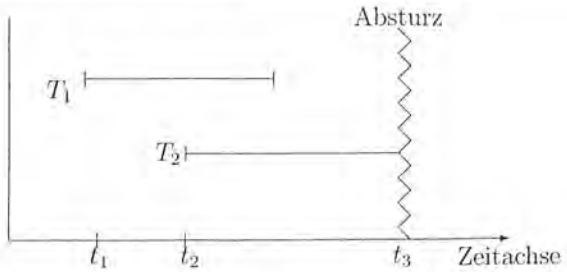


Abbildung 10.6: Transaktionsbeginn und -ende relativ zu einem Systemabsturz

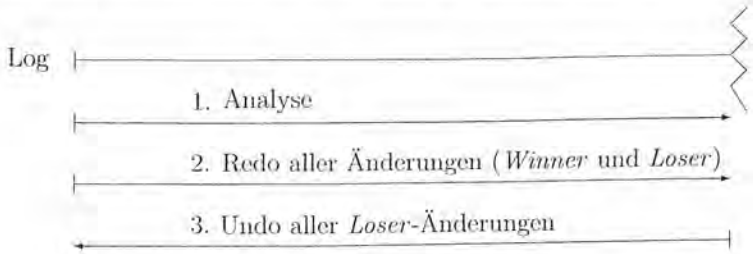


Abbildung 10.7: Wiederanlauf in drei Phasen

Nach dem von uns hier vorgestellten Recoverykonzept geschieht der Wiederanlauf in drei Phasen:

1. *Analyse*: Die temporäre Log-Datei wird von Anfang bis Ende analysiert, um die *Winner*-Menge von Transaktionen des Typs T_1 und die *Loser*-Menge von Transaktionen der Art T_2 zu ermitteln.

2. *Wiederholung der Historie*: Es werden *alle* protokollierten Änderungen in der Reihenfolge ihrer Ausführung in die Datenbasis eingebracht.

3. *Undo der Loser*: Die Änderungoperationen der *Loser*-Transaktionen werden in umgekehrter Reihenfoge ihrer ursprünglichen Ausführung rückgängig gemacht.

Diese Vorgehensweise ist in Abbildung 10.7 schematisch dargestellt. Wir wollen die drei Phasen des Wiederanlaufs nachfolgend etwas detaillierter beschreiben.

10.4.1 Analyse des Logs

Die Analyse der Log-Datei geht sehr einfach vonstatten. Die **BOT**-Einträge geben Aufschluss über alle Transaktionen, die in der betreffenden Zeitspanne gestartet wurden. Die **commit**-Einträge im Log kennzeichnen die *Winner*-Transaktionen. Alle begonnenen Transaktionen, zu denen kein **commit** in der Log-Datei gefunden werden kann, sind *Loser*. Jetzt dürfte nochmals nachhaltig die Bedeutung der Regel 1. des WAL-Prinzips (Ausschreiben aller Log-Einträge vor Abschluss einer Transaktion) klar werden.

10.4.2 Redo-Phase

In der *Redo*-Phase wird die Log-Datei sequentiell – in der Reihenfolge des Anlegens der Log-Einträge – durchlaufen. Für jeden Log-Eintrag wird die betroffene Seite aus der materialisierten Datenbasis in den Datenbankpuffer geholt (falls sie nicht schon aufgrund vorhergehender *Redo*-Vorgänge dort ist) und deren LSN ermittelt. Falls die in der Seite stehende LSN gleich oder größer ist als die LSN des Log-Eintrags, braucht nichts gemacht zu werden – das After-Image zu der protokollierten Änderungsoperation befindet sich schon auf der Seite. Anderenfalls, wenn die Seiten-LSN kleiner als die Log-Record-LSN ist, muss die im Log-Record gespeicherte *Redo*-Operation ausgeführt werden. Außerdem muss in diesem Fall (also bei durchgeführter *Redo*-Operation) die LSN der Seite durch die LSN des gerade bearbeiteten Log-Records ersetzt werden. Dies ist wichtig für den erneuten Wiederanlauf nach einem Fehler in der Wiederanlaufphase (siehe Abschnitt 10.5). Man beachte, dass auch die Änderungen der *Loser*-Transaktionen nachvollzogen werden – siehe Übung 10.6.

10.4.3 Undo-Phase

Nachdem die *Redo*-Phase abgeschlossen ist, wird die Log-Datei in umgekehrter Richtung (von hinten nach vorne) für die *Undo*-Phase durchlaufen. Wir übergehen alle Log-Einträge, die zu einer *Winner*-Transaktion gehören. Aber für jeden Log-Eintrag einer *Loser*-Transaktion wird die *Undo*-Operation ausgeführt. Anders als beim *Redo* wird das *Undo* auf jeden Fall ausgeführt, egal welche LSN auf der Seite steht. Dies ist notwendig, da auf jeden Fall das After-Image der protokollierten Operation auf der Seite steht: Entweder wurde es noch vor dem Absturz in die materialisierte Datenbasis geschrieben oder in der vorangegangenen *Redo*-Phase wiederhergestellt.

Wir werden im nächsten Abschnitt sehen, dass in der *Undo*-Phase zusï¿½tzlich noch Protokoll-Einträge – sogenannte Kompensationseinträge – generiert werden müssen.

10.5 Fehlertoleranz des Wiederanlaufs

Bei der Entwicklung der Recoverykomponente muss man natürlich auch die Fehlertoleranz bei einem Absturz innerhalb der Wiederanlaufphasen gewährleisten. Anders ausgedrückt: die *Redo*- und die *Undo*-Phasen müssen *idempotent* sein, d.h. sie müssen auch bei mehrfacher Ausführung (hintereinander) immer wieder dasselbe Ergebnis liefern. Zu jeder ausgeführten (Änderungs-)Aktion a gilt:

$$\begin{aligned} undo(undo(\cdots(undo(a))\cdots)) &= undo(a) \\ redo(redo(\cdots(redo(a))\cdots)) &= redo(a) \end{aligned}$$

Die Idempotenz der *Redo*-Phase wird dadurch erreicht, dass die LSN des Log-Records, für den ein *Redo* (tatsächlich) ausgeführt wird, in die Seite eingetragen wird. Dadurch wird sichergestellt, dass auch nach einem Absturz während des Wiederanlaufs ein *Redo* einer Operation nicht „versehentlich“ auf dem After-Image der Operation ausgeführt wird. Wieso? – siehe dazu Übungsaufgabe 10.7.

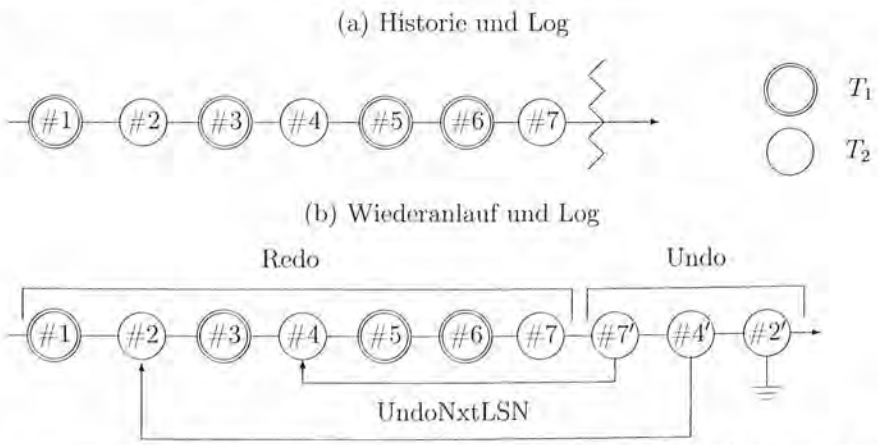


Abbildung 10.8: Wiederanlauf nach einem Absturz: (a) das vorgefundene Log und (b) die Fortschreibung der Log-Datei aufgrund der *Undo*-Aktionen

Für die Gewährleistung der Idempotenz der *Undo*-Phase müssen wir noch das Konzept der Kompensations-Protokolleinträge (CLR, compensation log record) einführen. Für jede ausgeführte *Undo*-Operation wird ein CLR angelegt, der genau wie „normale" Log-Records eine eindeutige LSN zugeteilt bekommt. Der CLR enthält die folgenden Informationen:

⌈ LSN, Transaktions-ID, Seiten-ID, *Redo*-Information, *PrevLSN*, *UndoNxtLSN* ⌋

Die *Redo*-Information eines CLR entspricht der während der *Undo*-Phase des Wiederanlaufs ausgeführten *Undo*-Operation. Bei einem erneuten Wiederanlauf wird diese Operation aber innerhalb der *Redo*-Phase ausgeführt. Kompensationseinträge benötigen keine *Undo*-Informationen, da sie – obwohl sie den *Loser*-Transaktionen zugeordnet werden – während nachfolgender *Undo*-Phasen übersprungen werden. Um das Überspringen zu gewährleisten, enthalten sie das Feld *UndoNxtLSN*. Dieses Feld enthält die LSN der zu dieser Transaktion gehörenden Änderungsoperation, die der kompensierten Operation vorausging. Diese Information ist relativ effizient aus der Rückwärtsverkettung (*PrevLSN*) der Protokolleinträge einzelner Transaktionen zu ermitteln. Die Abbildung 10.8 skizziert diese Situation für unsere Beispielausführung aus Abbildung 10.3. Wir nehmen an, dass der Absturz kurz vor dem **commit** von Transaktion T_2 stattfindet. In Abbildung 10.8 (a) ist die nach dem Absturz vorgefundene Log-Datei skizziert. Sie enthält alle Log-Records bis zur LSN #7.[1] Der untere Teil (b) der Abbildung zeigt den Zustand der Log-Datei nach dem vollständigen Wiederanlauf. Die drei Einträge #2, #4 und #7 der *Loser*-Transaktion T_2 – gekennzeichnet durch einfache Kreise, wohingegen T_1 Log-Einträge durch doppelte Kreise markiert sind – sind durch die CLRs #7', #4' und #2' kompensiert worden. Wir haben die (hoffentlich) anschauliche Notation #*i*' als LSN des Kompensationseintrages für den Log-Record #*i* verwendet, obwohl natürlich auch die LSNs der

[1] Was würde passieren, wenn die Log-Datei nur die Einträge bis LSN #6 enthielte? – Siehe Übungsaufgabe 10.5.

$$[\#1, T_1, \mathbf{BOT}, 0]$$
$$[\#2, T_2, \mathbf{BOT}, 0]$$
$$[\#3, T_1, P_A, A-=50, A+=50, \#1]$$
$$[\#4, T_2, P_C, C+=100, C-=100, \#2]$$
$$[\#5, T_1, P_B, B+=50, B-=50, \#3]$$
$$[\#6, T_1, \mathbf{commit}, \#5]$$
$$[\#7, T_2, P_A, A-=100, A+=100, \#4]$$
$$\langle \#7', T_2, P_A, A+=100, \#7, \#4 \rangle$$
$$\langle \#4', T_2, P_C, C-=100, \#7', \#2 \rangle$$
$$\langle \#2', T_2, -, -, \#4', 0 \rangle$$

Abbildung 10.9: Logeinträge nach abgeschlossenem Wiederanlauf

Kompensationseinträge fortlaufend sein müssen. Also muss die LSN $\#4'$ einen größeren Wert als $\#7'$ haben und $\#7'$ einen größeren Wert als $\#7$. Weiterhin sind in der Abbildung die *UndoNxtLSN*-Verweise der Kompensationseinträge gezeigt.

Nach Abschluss des Wiederanlaufs hätte die Log-Datei – bzw. die Log-Datei einschließlich des Log-Ringpuffers – die Gestalt, wie sie in Abbildung 10.9 gezeigt wird (vgl. Abbildung 10.3). Die Log-Einträge mit spitzen Klammern sind die CLR-Einträge. Der letzte davon – mit der Kennung $\#2'$ – enthält keine Seiten- bzw. *Redo*-Information, da er sich nur auf das **BOT** der Transaktion T_2 bezieht. An dem NULL-Eintrag des *UndoNxtLSN*- Feldes dieses CLR ist außerdem erkennbar, dass zu diesem Zeitpunkt die Transaktion T_2 vollständig zurückgesetzt ist, so dass zumindest für diesen „*Loser*" keine weiteren *Undo*-Operationen mehr ausgeführt werden müssen.

Was passiert jetzt bei einem erneuten Systemabsturz und dadurch initiierten Wiederanlauf? In der *Redo*-Phase würde die gesamte Log-Datei von $\#1$ bis $\#2'$ in Vorwärtsrichtung bearbeitet. Es wird sichergestellt, dass alle protokollierten Änderungen – einschließlich der Kompensationen – in die Datenbasis eingebracht werden. In der nachfolgenden *Undo*-Phase wird die Log-Datei dann in umgekehrter Richtung bearbeitet. Da der Zeiger *UndoNxtLSN* von $\#2'$ aber *NULL* ist, wäre dadurch kenntlich gemacht, dass die Transaktion T_2 vollständig zurückgesetzt ist.

Nehmen wir aber mal den Fall an, dass die Log-Datei nur die Einträge bis einschließlich $\#7'$ enthält. In diesem Fall werden in der *Redo*-Phase alle Operationen von $\#1$ bis $\#7$ und die Kompensation $\#7'$ nachvollzogen. Bei der anschließenden *Undo*-Phase wird $\#7'$ nicht rückgängig gemacht (es ist ja eine Kompensation, die in der Datenbasis erhalten bleiben soll), sondern anhand des *UndoNxtLSN*-Zeigers ermittelt, dass die nächste zu kompensierende Operation von T_2 in $\#4$ protokolliert ist. Die dort gespeicherte *Undo*-Operation wird ausgeführt und der CLR $\#4'$ angelegt. In $\#4$ steht ein *PrevLSN*-Verweis auf $\#2$, die als nächstes kompensiert und mit $\#2'$ protokolliert wird. Danach hat man dann den Zustand, der in Abbildung 10.8 (b) beschrieben ist.

10.6 Lokales Zurücksetzen einer Transaktion

Wir können jetzt auch das isolierte Zurücksetzen einer einzelnen Transaktion behandeln. Dazu muss man lediglich die zu dieser Transaktion gehörenden Log-Einträge

in chronologisch umgekehrter Reihenfolge abarbeiten. Man beachte, dass wir jetzt einen intakten Hauptspeicher – also sowohl Datenbankpuffer als auch Log-Puffer – voraussetzen. Man ermittelt den zuletzt angelegten Log-Record der zurückzusetzenden Transaktion. Einen Zeiger auf diesen jüngsten Log-Record könnte man für jede Transaktion im Haupspeicher unterhalten.

Durch die Rückwärtsverkettung der Log-Records über den *PrevLSN*-Eintrag kann man dann alle Protokolleinträge einer Transaktion sehr effizient rückwärts durchlaufen. Jede protokollierte Operation wird durch Ausführung der *Undo*-Information zurückgesetzt. Vor der Ausführung wird ein entsprechender Kompensations-Record (CLR) protokolliert. Die LSN des Kompensations-Records wird in die betreffende Seite eingetragen, damit man bei einem später evtl. notwendigen Wiederanlauf erkennen kann, ob die Kompensation in der Seite enthalten ist oder nicht. Bei einer vernünftigen Größe des Log-Ringpuffers kann man davon ausgehen, dass die meisten Protokolleinträge einer noch aktiven Transaktion im Hauptspeicher verfügbar sind. Deshalb ist das lokale Rollback einer Transaktion sehr effizient durchführbar.

Die beim lokalen Rollback angelegten Kompensationseinträge unterscheiden sich nicht von denen, die beim globalen *Undo* infolge eines Wiederanlaufs angelegt werden. Auch die beim lokalen Rollback angelegten CLRs werden bei einem später eventuell durchgeführten Wiederanlauf in der *Undo*-Phase übersprungen. Deshalb muss auch hier jeweils der *UndoNxtLSN*-Zeiger gesetzt werden – so wie das in Abbildung 10.8 (b) angedeutet war.

Beim isolierten Rollback müssen zusätzlich noch die von der Transaktion gesetzten Sperren (siehe Kapitel 11) aufgegeben werden. Dies war beim Wiederanlauf nicht notwendig, da die im Hauptspeicher verwalteten Sperren durch den Absturz sowieso verloren gingen, also zwangsläufig freigegeben wurden.

10.7 Partielles Zurücksetzen einer Transaktion

Im vorangegangen Kapitel hatten wir gesehen, dass einige Datenbanksysteme die Definition von Rücksetzpunkten innerhalb einer Transaktion erlauben. Dadurch soll das Rollback der Transaktion bis zu einem definierten Rücksetzpunkt ermöglicht werden. Dieses Konzept kann mit unserer bislang vorgestellten Recoverymethode leicht unterstützt werden. Die chronologisch nach der Definition des Rücksetzungspunkts erfolgten Änderungsoperationen werden lokal zurückgenommen, indem die *Undo*-Operation ausgeführt wird und entsprechende Kompensationseinträge in das Log angehängt werden. Grafisch ist dies in Abbildung 10.10 gezeigt.

In dieser Skizze werden zunächst die Operationen 1, 2, 3 und 4 ausgeführt. Dazu gibt es entsprechende Log-Eintäge #1, #2, #3 und #4. Die letzten beiden Operationen – also 3 und 4 – werden zurückgesetzt. Die Kompensationen sind in den CLRs #4′ und #3′ dokumentiert – wiederum muss natürlich gelten, dass die LSN #4′ größer als #4 und #3′ größer als #4′ ist. Danach wird die Operation 5 ausgeführt und im Log mit der LSN #5 dokumentiert. Somit hat die Transaktion nur die Operationen 1, 2 und 5 ausgeführt, wohingegen 3 und 4 wieder rückgängig gemacht wurden.

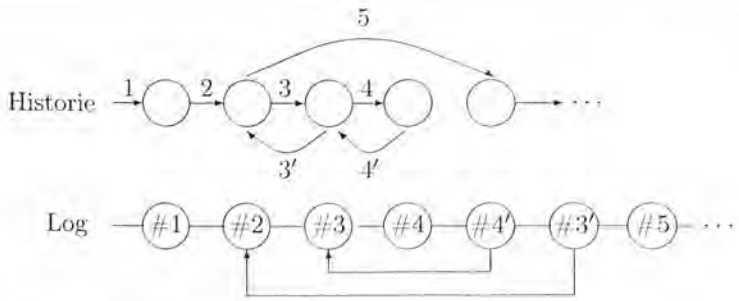


Abbildung 10.10: Partielles Rücksetzen einer Transaktion

10.8 Sicherungspunkte

Die bislang vorgestellte Recoverymethodik hat einen entscheidenden Nachteil: Der Wiederanlauf des Systems wird mit zunehmender Betriebszeit des Datenbanksystems immer langwieriger, da die zu verarbeitende Log-Datei immer umfangreicher wird. Abhilfe schaffen *Sicherungspunkte*, die sozusagen als Brandmauer (engl. fire wall) für den Wiederanlauf dienen. Durch einen Sicherungspunkt wird eine Position im Log markiert, über die man beim Wiederanlauf nicht hinausgehen muss. Alle älteren Log-Einträge sind irrelevant und könnten demnach auch aus der temporären Log-Datei entfernt werden. Aber schon an dieser Stelle sei betont, dass nicht unbedingt der Zeitpunkt des Sicherungspunktes den „cut-off" bildet – bei einigen Sicherungspunktverfahren muss man auch ältere Log-Einträge beachten. Auf jeden Fall wird aber beim Anlegen des Sicherungspunkts der „cut-off" – also die kleinste noch benötigte LSN – für den Wiederanlauf festgelegt. Wir behandeln drei Arten von Sicherungpunkten:

1. (globale) transaktionskonsistente Sicherungspunkte,
2. aktionskonsistente Sicherungspunkte und
3. unscharfe (fuzzy) Sicherungspunkte.

10.8.1 Transaktionskonsistente Sicherungspunkte

Bei der vorangegangenen Diskussion der Recoveryverfahren hatten wir stillschweigend angenommen, dass die Log-Datei zu einem Zeitpunkt begonnen wurde, als sich die Datenbasis in einem transaktionskonsistenten Zustand auf der Platte befand. Man kann dann nach einer bestimmten Betriebszeit des Systems einen neuen transaktionskonsistenten Sicherungspunkt anmelden. Daraufhin wird das Datenbanksystem in den „Ruhezustand" überführt. Alle neu ankommenden Transaktionen müssen warten und alle noch aktiven Transaktionen können zu Ende geführt werden. Sobald letzteres geschehen ist, werden alle modifizierten Seiten auf den Hintergrundspeicher ausgeschrieben. Jetzt enthält die materialisierte Datenbasis einen transaktionskonsistenten Zustand, d.h. die Seiten enthalten ausschließlich die Änderungen von erfolgreich abgeschlossenen Transaktionen. Somit kann man jetzt die Log-Datei

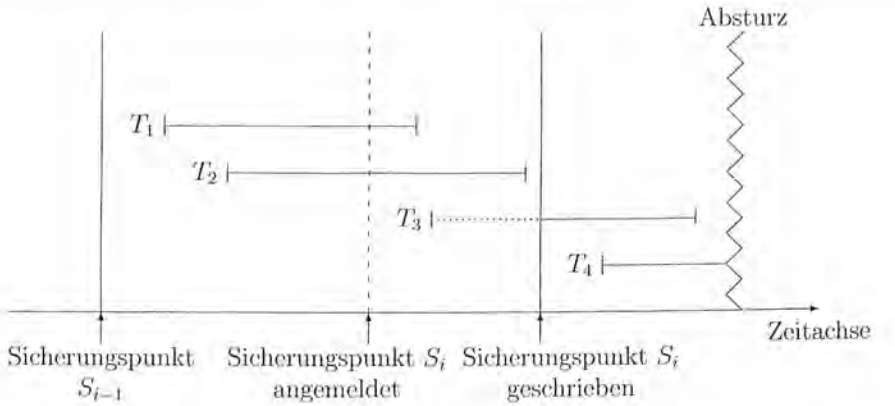


Abbildung 10.11: Transaktionsausführung relativ zu einem transaktionskonsistenten Sicherungspunkt und einem Systemabsturz

wieder von neuem beginnen. Abbildung 10.11 zeigt dieses Vorgehen schematisch. Der Beginn der Transaktion T_3 wird verzögert (gepunktete Linie), bis der angemeldete Sicherungspunkt S_i geschrieben ist. Bei einem späteren Absturz benötigt man dann nur die Log-Information, die seit dem Schreiben des Sicherungspunkts S_i angelegt wurde. Sie enthält alle Einträge, um T_3 nachvollziehen (*Redo*) und T_4 zurücksetzen zu können (*Undo*). Die Änderungen von T_1 und T_2 sind in der materialisierten Datenbasis enthalten und benötigen deshalb keinerlei Aufmerksamkeit seitens der Recovery. Die Vorgehensweise ist in Abbildung 10.12 (a) skizziert. Das Anlegen transaktionskonsistenter Sicherungspunkte ist sehr zeitaufwendig und deshalb nur in Ausnahmefällen durchführbar – z.B. am Wochenende, wenn das System sowieso heruntergefahren wird. Zum einen muss man den Beginn neu ankommender Transaktionen verzögern, zum anderen muss man die gesamten modifizierten Pufferinhalte auf den Hintergrundspeicher ausschreiben. Bei der update-in-place Einbringstrategie muss man natürlich auch beim Schreiben eines Sicherungspunkts das WAL-Prinzip befolgen und folglich den gesamten Log-Ringpuffer ausschreiben (siehe dazu Übungsaufgabe 10.8).

10.8.2 Aktionskonsistente Sicherungspunkte

Aufgrund der unzumutbaren Verzögerung neu eintreffender Transaktionen beim Anlegen transaktionskonsistenter Sicherungspunkte ist man gezwungen, Abstriche hinsichtlich der Qualität der Sicherungspunkte in Kauf zu nehmen. Die aktionskonsistenten Sicherungspunkte verlangen nur, dass vor dem Anlegen eines Sicherungspunkts alle (elementaren) Änderungsoperationen vollständig abgeschlossen sind. Abbildung 10.13 zeigt schematisch das Anlegen eines solchen Sicherungspunkts. Die Punkte (•) repräsentieren Änderungsoperationen der Transaktionen. Nach Anmeldung des Sicherungspunkts (gestrichelte senkrechte Linie) werden gerade bearbeitete Operationen abgeschlossen – wie z.B. die zweite Operation von T_4. Danach werden alle modifizierten Seiten aus dem Puffer in den Hintergrundspeicher über-

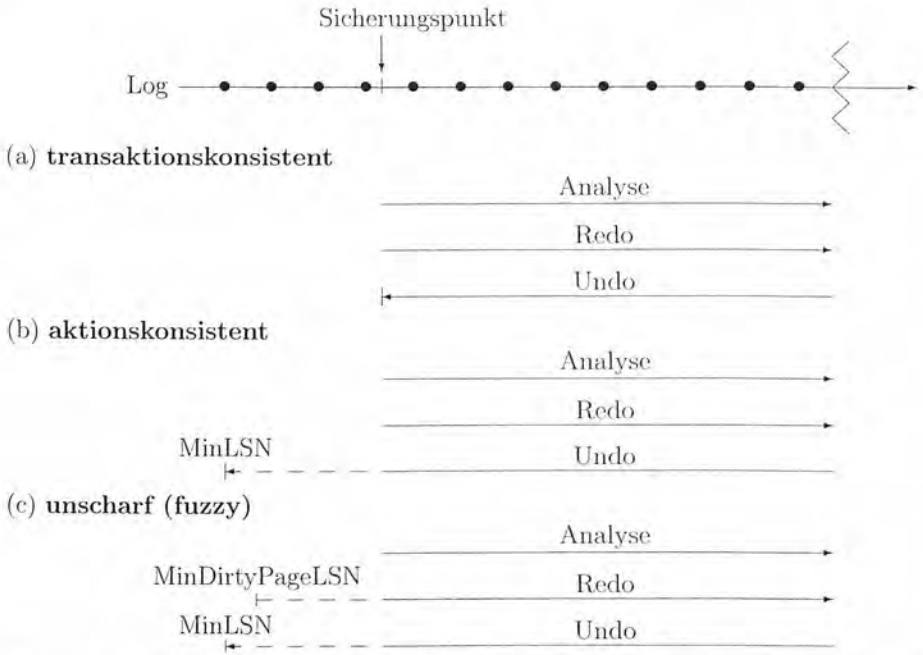


Abbildung 10.12: Wiederanlauf bei den drei unterschiedlichen Sicherungspunkt-Qualitäten

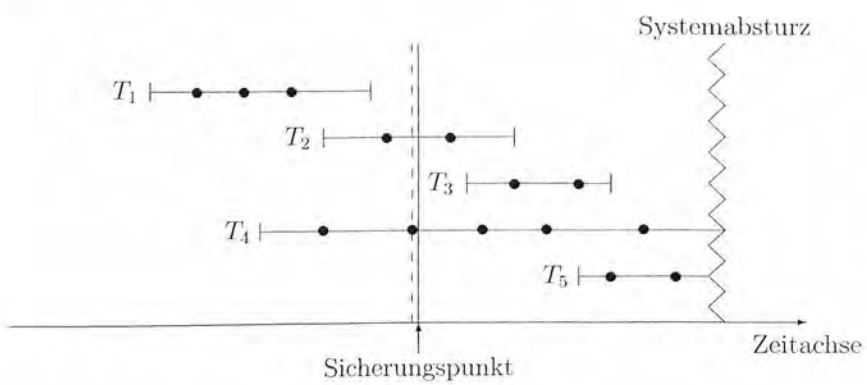


Abbildung 10.13: Transaktionsausführung relativ zu einem aktionskonsistenten Sicherungspunkt und einem Systemabsturz

tragen – wiederum muss nach dem WAL-Prinzip die Log-Information zuerst ausgeschrieben werden. Nach Abschluss des Sicherungspunkts braucht man bei einem späteren Wiederanlauf garantiert keine *Redo*-Informationen, die älter sind als der Sicherungspunkt. Wohl aber benötigt man u.U. *Undo*-Informationen, die älter sind als der Zeitpunkt, zu dem der Sicherungspunkt geschrieben wurde. Dies ist in Abbildung 10.13 z.B. für T_4 der Fall, da die ersten beiden Änderungen von T_4 auf jeden Fall in der materialisierten Datenbasis vorhanden sind.

Beim Anlegen eines aktionskonsistenten Sicherungspunkts kann man die kleinste LSN aller zu diesem Zeitpunkt aktiven Transaktionen ermitteln. In unserem Beispiel entspricht diese dem Protokolleintrag der ersten Operation von T_4. Diese LSN bezeichnen wir mit *MinLSN*. Weiterhin wird in der Protokolldatei eine Liste aller zum Zeitpunkt des Sicherungspunkts aktiven Transaktionen abgelegt. Diese Liste wird in der *Analyse*-Phase des Wiederanlaufs benötigt, um alle *Loser*-Transaktionen ermitteln zu können. Warum?

Beim Wiederanlauf setzen die *Analyse*- und *Redo*-Phase auf dem Sicherungspunkt auf. Die *Undo*-Phase muss allerdings über den Sicherungspunkt hinausgehen – und zwar bis zur Position *MinLSN* der Log-Datei. Diese Vorgehensweise ist in Abbildung 10.12 (b) gezeigt.

10.8.3 Unscharfe (fuzzy) Sicherungspunkte

Das Anlegen eines aktionskonsistenten Sicherungspunkts verlangt, dass der gesamte modifizierte Teil des Datenbankpuffers und die gesamte Log-Information „auf einen Schwung" ausgeschrieben wird. Dies führt zu einer starken Systembelastung. Normalerweise sollte man versuchen, modifizierte Seiten kontinuierlich auszuschreiben, da man dadurch CPU- und Ein/Ausgabe-Vorgänge überlappen kann. Dadurch entfällt das untätige Warten des Prozessors auf den langsameren Hintergrundspeicher, was letztlich zu einem sehr viel größeren Durchsatz (hinsichtlich Anzahl bearbeiteter Transaktionen) führt.

Die Idee der unscharfen (fuzzy) Sicherungspunkte besteht darin, anstatt die modifizierte Seite auszuschreiben, nur deren Kennungen in der Log-Datei zu notieren – nennen wir diese Menge *DirtyPages*. Für die Seiten in *DirtyPages* muss zusätzlich noch die minimale LSN – nennen wir sie *MinDirtyPageLSN* – ermittelt werden, mit der die am längsten nicht mehr ausgeschriebene Seite in den Hintergrundspeicher propagiert wurde. Diese LSN steht natürlich nicht in den Pufferseiten, sondern könnte höchstens durch Inspektion der Seiten im Hintergrundspeicher ermittelt werden. Das ist natürlich viel zu teuer, weshalb man sich zu jeder Seite im Puffer die LSN der ältesten noch nicht ausgeschriebenen Änderungsoperation merken muss. Die kleinste aller dieser LSNs bildet also die *MinDirtyPageLSN*. Diese *MinDirtyPageLSN* legt den Aufsetzpunkt für die *Redo*-Phase fest. Innerhalb der Spanne von der *MinDirtyPageLSN* bis zum Sicherungspunkt braucht man nur die Seiten aus *DirtyPages* zu beachten.

Den „cut-off" für die rückwärts gerichtete *Undo*-Phase bildet wieder die analog zu oben ermittelte *MinLSN* der beim Sicherungspunkt aktiven Transaktionen.

Das Vorgehen des Wiederanlaufs bei unscharfen Sicherungspunkten ist in Abbildung 10.12 (c) skizziert.

Die Effizienz des Wiederanlaufs bei unscharfen Sicherungspunkten hängt von der

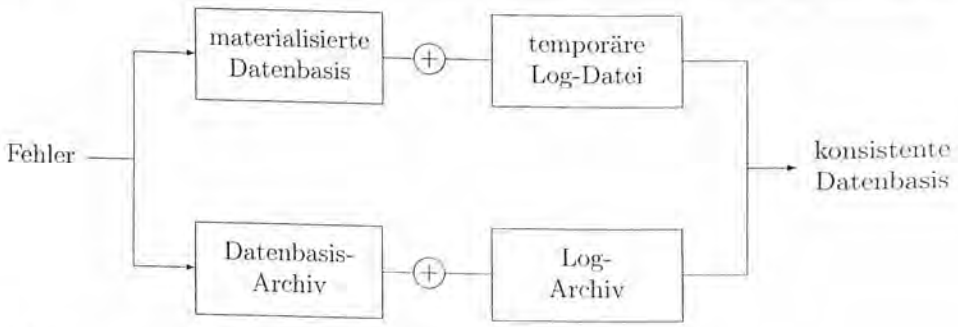


Abbildung 10.14: Die zwei Recovery-Alternativen

Pufferverwaltung ab. Wenn einige „hot-spot"-Seiten dauerhaft im Puffer verbleiben, ohne dass ihre Änderungen je ausgeschrieben werden, muss in der *Redo*-Phase die Log-Datei von ganz vorne bis hinten durchlaufen werden. Deshalb ist es unabdingbar, dass modifizierte Seiten kontinuierlich ausgeschrieben werden. Einige Systeme erzwingen das Ausschreiben der Seiten, die bei zwei aufeinanderfolgenden unscharfen Sicherungspunkten in der Menge *DirtyPages* enthalten sind und zwischenzeitlich noch nicht ausgeschrieben wurden.

10.9 Recovery nach einem Verlust der materialisierten Datenbasis

Der bislang vorgestellte Mechanismus für den Wiederanlauf setzte voraus, dass die materialisierte Datenbasis *und* die Log-Datei intakt vorgefunden werden. Wenn eine (oder sogar beide) dieser Dateien zerstört sind, benötigt man für die Recovery sogenannte Archivkopien, die auf ein Archivmedium – i.A. ein Magnetband – kopiert werden. Wir setzen hier voraus, dass sich die Datenbasis dabei in einem transaktionskonsistenten Zustand befindet. Die Log-Information wird kontinuierlich auf ein Archivband geschrieben – d.h. bei jedem Ausschreiben von Log-Einträgen aus dem Ringpuffer werden die temporäre Log-Datei *und* das Log-Archiv geschrieben. Diese Archivierungsvorgänge waren in Abbildung 10.4 eingezeichnet.

Bei Zerstörung der materialisierten Datenbasis oder der Log-Datei kann man somit aus der Archivkopie und dem Log-Archiv, dessen Einträge zu dem Zeitpunkt des Anlegens der DB-Archivkopie beginnen, den jüngsten konsistenten Zustand wiederherstellen.

Man kann auch aktionskonsistente Zustände der Datenbasis archivieren. Dann muss aber das Log-Archiv auch ältere Einträge beinhalten – siehe Aufgabe 10.9.

In Abbildung 10.14 sind die zwei möglichen Recoveryarten nach einem Systemabsturz eingezeichnet:

1. Der obere (schnellere) Weg wird bei intakten Hintergrundspeichern (sowohl materialisierte Datenbasis als auch Log-Datei) beschritten.

2. Der untere (langsamere) Pfad muss bei zerstörtem Hintergrundspeicherinhalt gewählt werden.

Es ist natürlich auch denkbar, dass man bei zerstörter Log-Datei die materialisierte Datenbasis und das Log-Archiv als Ausgangsinformationen wählt. Kann man auch das DB-Archiv und die Log-Datei für den Wiederanlauf kombinieren?

10.10 Übungen

10.1 Demonstrieren Sie anhand eines Beispiels, dass man die Strategien *force* und $\neg$*steal* nicht kombinieren kann, wenn parallele Transaktionen gleichzeitig Änderungen an Datenobjekten innerhalb einer Seite durchführen. Betrachten Sie dazu z.B. die in Abbildung 10.1 dargestellte Seitenbelegung, bei der die Seite P_A die beiden Datensätze A und D enthält. Entwerfen Sie eine verzahnte Ausführung zweier Transaktionen, bei denen eine Kombination aus *force* und $\neg$*steal* ausgeschlossen ist.

10.2 In Abbildung 10.3 ist die verzahnte Ausführung der beiden Transaktionen T_1 und T_2 und das zugehörige *Log* auf der Basis logischer Protokollierung gezeigt. Wie sähe das Log bei physischer Protokollierung aus, wenn die Datenobjekte A, B und C die Initialwerte 1000, 2000 und 3000 hätten?

10.3 Wie sähe die Log-Datei bei physischer Protokollierung nach Durchführung des Wiederanlaufs – also in dem in Abbildung 10.8 (b) skizzierten Zustand – aus?

10.4 In Abschnitt 10.7 ist das partielle Rücksetzen einer Transaktion beschrieben worden. Es wurde an der in Abbildung 10.10 skizzierten Transaktion demonstriert. Wie würde sich das vollständige Rollback dieser Transaktion gestalten, nachdem Operation 5 ausgeführt ist? Wie sieht das Log nach dem vollständigen Rollback aus?

10.5 Betrachten Sie Abbildung 10.8. In Teil (a) ist das Log bis LSN #7 skizziert. Was passiert, wenn die auf der Platte stehende Log-Datei nur die Einträge bis LSN #6 enthielte? Demonstrieren Sie den Wiederanlauf des Systems unter diesem Gesichtspunkt.

Könnte auch der Eintrag#5 in der temporären Log-Datei fehlen, obwohl der Absturz erst nach Schritt 15 in Abbildung 10.3 stattfand? Welches Prinzip wäre dadurch verletzt?

10.6 [Mohan et al. (1992)] Weisen Sie nach, dass die Idempotenz des Wiederanlaufs das *Redo aller* protokollierten Änderungen – also auch der von *Losern* durchgeführten Änderungen – verlangt.

Hinweis: Betrachten Sie zwei Transaktionen T_L und T_W, wobei T_L ein *Loser* und T_W ein *Winner* ist.

T_L modifiziere ein Datum A auf Seite P_1 und anschließend modifiziere T_W ein Datum B, auch auf Seite P_1. Diskutieren Sie unterschiedliche Zustände der Seite P_1 auf dem Hintergrundspeicher:

- Zustand vor Modifikation von A,
- Zustand nach Modifikation von A aber vor Modifikation von B,
- Zustand nach Modifikation von B.

Was passiert beim Wiederanlauf in Bezug auf diese drei möglichen Zustände der Seite P_1? Veranschaulichen Sie Ihre Diskussion grafisch.

10.7 Zeigen Sie, dass es für die Erzielung der Idempotenz der *Redo*-Phase notwendig ist, die – und nur die – LSN einer tatsächlich durchgeführten *Redo*-Operation in der betreffenden Seite zu vermerken.

Was würde passieren, wenn man in der *Redo*-Phase gar keine LSN-Einträge in die Datenseiten schriebe?

Was wäre, wenn man auch LSN-Einträge von Log-Records, für die die *Redo*-Operation nicht ausgeführt wird, in die Datenseiten übertragen würde?

Was passiert, wenn der Kompensationseintrag geschrieben wurde, und dann noch vor der Ausführung des *Undo* das Datenbanksystem abstürzt?

10.8 Warum muss beim Anlegen eines transaktionskonsistenten Sicherungspunkts der gesamte Log-Ringpuffer ausgeschrieben werden – wo man doch nach Fertigstellung des Sicherungspunkts wieder mit einer „leeren" Log-Datei anfangen kann?

10.9 Wie weit in die Vergangenheit müssen die Einträge des Log-Archives gehen, wenn man einen aktionskonsistenten Zustand der Datenbasis archiviert? Wie sieht in diesem Fall die Wiederherstellung des jüngsten konsistenten DB-Zustands nach Verlust des Hintergrundspeichers aus?

10.11 Literatur

Die ersten wegweisenden Recoverytechniken wurden von Gray et al. (1981) in der System R-Entwicklung bei IBM konzipiert. Härder und Reuter (1983) lieferten in ihrem vielbeachteten Aufsatz die erste systematische Klassifikation einzelner Techniken und die Einordnung existierender Systemlösungen. In diesem Aufsatz wurde der ACID-Begriff geprägt; ferner ist unsere Klassifizierung der Einbring- und Ersetzungsstrategien an diesen Aufsatz angelehnt. Reuter (1980) hat Protokollierungsverfahren für die Undo-Recovery beschrieben. Elhardt und Bayer (1984) entwickelten den sogenannten „Datenbank-Cache", der den Wiederanlauf eines Datenbanksystems nach einem Systemfehler beschleunigt. Reuter (1984) untersuchte die Leistungsfähigkeit verschiedener Recovery-Strategien. Die hier beschriebene Recoverytechnik lehnt sich eng an das ARIES-Verfahren von Mohan et al. (1992) an. Dieses Verfahren findet sich heute in vielen kommerziellen Systemen – insbesondere den IBM-Produkten, wie z.B. DB2. Das ARIES-Verfahren wurde von Franklin et al. (1992) und von Mohan und Narang (1994) für Client/Server-Architekturen weiterentwickelt. Lomet und Weikum (1998) behandeln die Recovery von Client/Server-Anwendungen. Härder und Rothermel (1987) haben die Recoverykonzepte auf geschachtelte Transaktionen ausgedehnt.

11. Mehrbenutzersynchronisation

Unter „*Multiprogramming*“ (Mehrbenutzerbetrieb) versteht man die gleichzeitige (nebenläufige, parallele) Ausführung mehrerer Programme. Der Mehrbenutzerbetrieb führt i.A. zu einer weitaus besseren Auslastung eines Computersystems als dies im Einzelbenutzerbetrieb möglich wäre. Dies liegt daran, dass Programme – insbesondere Datenbankanwendungen – sehr oft auf langsame Betriebsmittel (wie z.B. Hintergrundspeicher) oder interaktive Benutzereingaben warten müssen. In einem Einbenutzersystem wäre der Rechner (d.h. der Prozessor) während dieser Wartezeiten untätig, wohingegen im Mehrbenutzerbetrieb eine andere Anwendung während dieser Wartezeiten bedient werden kann – bis sie selbst auf ein Ereignis warten muss. In Abbildung 11.1 sind die Vorteile des Mehrbenutzerbetriebs bei der Ausführung von drei Transaktionen (T_1, T_2 und T_3) in idealisierter Weise dargestellt.

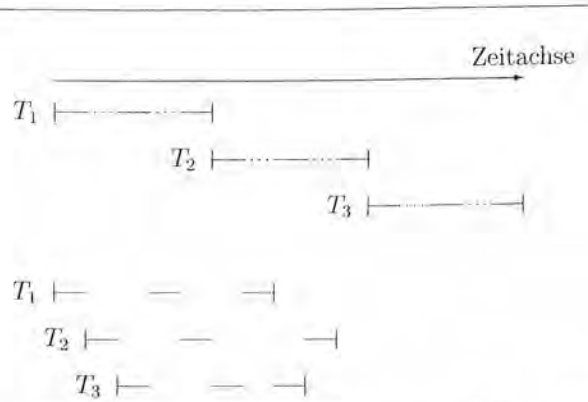


Abbildung 11.1: Ausführung der drei Transaktionen T_1, T_2 und T_3: (a) im Einbenutzerbetrieb und (b) im (verzahnten) Mehrbenutzerbetrieb (gestrichelte Linien repräsentieren Wartezeiten)

Es ist erkennbar, dass die Verzahnung (engl. *interleaving*) bei der Ausführung der drei Transaktionen zu einer wesentlich besseren Auslastung der CPU führt.[1]

Wir wollen uns in diesem Kapitel mit den Kontrollkonzepten beschäftigen, die für die Konsistenzerhaltung der Datenbank bei Mehrbenutzerbetrieb notwendig sind. Bezogen auf das ACID-Paradigma, beschäftigen wir uns in diesem Kapitel vorrangig mit dem *I* für *Isolation*. Unter der Anforderung *Isolation* versteht man, dass jeder Transaktion die Datenbank so erscheinen muss, als wenn sie die einzige Anwendung darauf wäre.

[1] In unserer idealisierten Darstellung ist die CPU 100 % ausgelastet und kein zusätzlicher Overhead, der durch den Mehrbenutzerbetrieb benötigt wird, berücksichtigt.

11.1 Fehler bei unkontrolliertem Mehrbenutzerbetrieb

Wir wollen uns zunächst mit den möglichen Fehlern, die im unkontrollierten (und nicht synchronisierten) Mehrbenutzerbetrieb auftreten können, beschäftigen. Wir werden diese Fehler in drei Fehlerklassen, denen jeweils ein Unterabschnitt gewidmet ist, aufteilen.

11.1.1 Verlorengegangene Änderungen (*lost update*)

Dieses Problem soll anhand der folgenden zwei Transaktionen aus dem Bankenbereich demonstriert werden:

- Transaktion T_1 transferiert 300,- Euro von Konto A nach Konto B, wobei zunächst Konto A belastet wird und danach die Gutschrift auf Konto B erfolgt.
- Die gleichzeitig ablaufende Transaktion T_2 schreibt dem Konto A die 3 % Zinseinkünfte gut.

Der Ablauf dieser beiden Transaktionen könnte wie folgt verzahnt ablaufen – wenn es keine Mehrbenutzersynchronisation gäbe:

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

Der Effekt dieser Ausführung ist, dass die in Schritt 5. dem Konto A gutgeschriebenen Zinsen verloren gehen, da der in Schritt 5. von T_2 geschriebene Wert in Schritt 6. gleich wieder von T_1 überschrieben wird. Deshalb geht der Effekt von Transaktion T_2 verloren.

11.1.2 Abhängigkeit von nicht freigegebenen Änderungen

Diese Fehlerklasse wird manchmal auch als „dirty read" bezeichnet, da ein Datum gelesen wird, das so niemals in einem gültigen (transaktionskonsistenten) Zustand der Datenbasis vorkommt. Wir wollen auch dies an unseren Beispieltransaktionen T_1 (Überweisung) und T_2 (Zinsgutschrift) demonstrieren:

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

In dieser verzahnten Ausführung liest T_2 in Schritt 4. einen Wert für Konto A, von dem T_1 schon 300,– Euro abgebucht hat. Aber T_1 wird in Schritt 9. mit **abort** abgebrochen, so dass die Wirkung von T_1 gänzlich zurückgesetzt werden muss. Leider hat aber T_2 in Schritt 5. die Zinsen basierend auf dem „falschen" Wert von A berechnet und in Schritt 6. dem Konto gutgeschrieben. Das heißt, die Transaktion T_2 wurde auf der Basis inkonsistenter Daten (*dirty data*) durchgeführt.

11.1.3 Phantomproblem

Das Phantomproblem taucht auf, wenn während der Abarbeitung einer Transaktion T_2 eine andere Transaktion T_1 ein neues Datum generiert, das T_2 eigentlich hätte mit berücksichtigen müssen. Wir wollen dies an einem konkreten Beispiel beleuchten. Dazu betrachten wir die folgenden beiden Transaktionen:

T_1	T_2
	select sum(KontoStand)
	from Konten
insert into Konten	
values $(C, 1000, \ldots)$	
	select sum(KontoStand)
	from Konten

Hierbei führt T_2 (innerhalb einer Transaktion) zweimal die SQL-Anfrage aus, um die Summe aller Kontostände zu ermitteln. Das Problem besteht nun darin, dass T_1 zwischenzeitlich ein neues Konto – nämlich das Konto mit der Kennung C und dem Kontostand 1000,– Euro einfügt, das aber erst beim zweiten „Durchgang" der SQL-Anfrage berücksichtigt wird. Die Transaktion T_2 berechnet also zwei unterschiedliche Werte, da zwischenzeitlich das „Phantom" C eingefügt wurde.

11.2 Serialisierbarkeit

In der Einleitung des Kapitels haben wir die Nachteile der seriellen (Nacheinander-) Ausführung von Transaktionen hinsichtlich der Leistungsfähigkeit des Gesamtsystems beschrieben.

Andererseits können die vorhin aufgeführten Fehler bei der seriellen Ausführung nicht auftreten, da sich Transaktionen nicht gegenseitig beeinflussen können. Beim Konzept der *Serialisierbarkeit* werden die Vorzüge der seriellen Ausführung –

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**

Abbildung 11.2: Serialisierbare Historie von T_1 und T_2

nämlich Isolation – mit den Vorteilen des Mehrbenutzerbetriebs – nämlich erhöhter Durchsatz – kombiniert. Intuitiv gesprochen entspricht die serialisierbare Ausführung einer Menge von Transaktionen einer kontrollierten, nebenläufigen, verzahnten Ausführung, wobei die Kontrollkomponente dafür sorgt, dass die (beobachtbare) Wirkung der nebenläufigen Ausführung einer möglichen seriellen Ausführung der Transaktionen entspricht.

11.2.1 Beispiele serialisierbarer Ausführungen (Historien)

Unter einer *Historie* versteht man die zeitliche Anordnung der einzelnen verzahnt ausgeführten Elementaroperationen einer Menge von nebenläufig bearbeiteten Transaktionen. Aus der Sicht der Mehrbenutzersynchronisation sind nur die elementaren Datenbankoperationen **read**, **write**, **insert** und **delete** relevant, da die Bearbeitung lokaler Variablen nicht von der Nebenläufigkeit beeinflusst wird.

Betrachten wir unsere zwei Beispieltransaktionen T_1 und T_2 aus Kapitel 10:

- T_1 transferiere einen bestimmten Betrag von A nach B.
- T_2 transferiere einen Betrag von C nach A.

Die nebenläufige Bearbeitung könnte zu der in Abbildung 11.2 gezeigten Historie führen. Da wir die Bearbeitung lokaler Variablen nicht mehr beachten, haben wir auch in den Lese- und Schreiboperationen auf die Angabe der lokalen Variablen verzichtet – also z.B. **read**(A) anstatt **read**(A, a_1).

Die oben gezeigte verzahnte Verarbeitung von T_1 und T_2 hat offensichtlich denselben Effekt wie die serielle Abarbeitung $T_1 \mid T_2$, die in Abbildung 11.3 gezeigt ist. Deshalb ist die Historie aus Abbildung 11.2 serialisierbar.

11.2.2 Nicht serialisierbare Historie

Die in Abbildung 11.4 gezeigte Verzahnung der zwei Transaktionen T_1 und T_3 ist nicht serialisierbar. Bezogen auf das Datenobjekt A kommt nämlich T_1 *vor* T_3; aber

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**

Abbildung 11.3: Serielle Ausführung von T_1 vor T_2, also $T_1 \mid T_2$

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**	

Abbildung 11.4: Nicht serialisierbare Historie

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**	

Abbildung 11.5: Zwei verzahnte Überweisungs-Transaktionen

hinsichtlich des Datums B kommt T_3 *vor* T_1. Deshalb ist diese Historie nicht äquivalent zu einer der beiden möglichen seriellen Ausführungen von T_1 und T_3, nämlich $T_1 \mid T_3$ oder $T_3 \mid T_1$.

Die aufmerksamen Leser werden sich an dieser Stelle fragen, wieso die in Abbildung 11.4 gezeigte Historie zu Inkonsistenzen führen sollte. Wenn T_1 und T_3 tatsächlich beide Überweisungen durchführen, wie in Abbildung 11.5 gezeigt, ist auch keine Inkonsistenz zu befürchten. Diese verzahnte Ausführung wäre auch tatsächlich äquivalent zu einer seriellen Ausführung. Wehe aber, wenn die Schritte 5. und 6. vor Schritt 4. ausgeführt worden wären. Dann hätten wir das Problem des „lost update" gehabt.

Warum wird die in Abbildung 11.5 dargestellte Historie dann aber nicht als serialisierbar betrachtet? Der Grund liegt darin, dass sie nur rein zufällig – wegen der speziellen Anwendungssemantik – äquivalent ist zu einer seriellen Historie. Aus der Sicht des Datenbanksystems ist diese Semantik jedoch nicht „erkennbar", denn das DBMS „sieht" nur die Lese- und Schreibvorgänge. Deshalb könnte die Historie aus Abbildung 11.4 sowohl zu der Ausführung in Abbildung 11.5 als auch zu der in Abbildung 11.6 gehören, wo T_3 einer Zinsgutschrift-Transaktion entspricht. Die Leser mögen verifizieren, dass diese Historie zu keiner der beiden möglichen seriellen Historien $T_1 \mid T_3$ oder $T_3 \mid T_1$ äquivalent ist (in jeder seriellen Ausführung hätte die Bank in der Summe 1,50 Euro mehr Zinsen bezahlen müssen). Das Datenbanksystem darf also bei der Mehrbenutzersynchronisation keine Annahmen hinsichtlich der Verarbeitung der Datenobjekte seitens der Anwendungstransaktionen machen. Die Konsistenz muss für jeden möglichen Datenbankzustand und für jede denkbare Verarbeitung garantiert werden.

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**	

Abbildung 11.6: Eine Überweisung (T_1) und eine Zinsgutschrift (T_3)

11.3 Theorie der Serialisierbarkeit

11.3.1 Definition einer Transaktion

Um die zugrundeliegende Theorie entwickeln zu können, benötigen wir zunächst eine formale Definition von Transaktionen. Eine Transaktion T_i besteht aus folgenden elementaren Operationen:

- $r_i(A)$ zum Lesen des Datenobjekts A,
- $w_i(A)$ zum Schreiben des Datenobjekts A,
- a_i zur Durchführung eines **abort**,
- c_i zur Durchführung des **commit**.

Eine Transaktion kann nur eine der beiden Operationen **abort** oder **commit** durchführen.

Weiterhin ist eine Reihenfolge (Ordnung) der Operationen einer Transaktion zu spezifizieren. Meistens gehen wir von einer streng sequentiellen Reihenfolge der Operationen aus, wodurch eine totale Ordnung gegeben wäre. Die Theorie lässt sich aber auch auf der Basis einer partiellen Ordnung entwickeln. Es müssen aber mindestens folgende Bedingungen hinsichtlich der auf den Operationen von T_i definierten partiellen Ordnung $<_i$ eingehalten werden:

- Falls T_i ein **abort** durchführt, müssen alle anderen Operationen $p_i(A)$ vor a_i ausgeführt werden, also $p_i(A) <_i a_i$.
- Analoges gilt für das **commit**, d.h. $p_i(A) <_i c_i$ falls T_i „**committed**".

$$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1$$

Abbildung 11.7: Historie einer Überweisungstransaktion T_1

- Wenn T_i ein Datum A liest und auch schreibt, muss die Reihenfolge festgelegt werden, also entweder $r_i(A) <_i w_i(A)$ oder $w_i(A) <_i r_i(A)$.

Wir können uns nun den Ablauf einer Überweisungstransaktion T_1 anschauen. Die Operationen und die zugehörige Ordnung (in diesem Fall ist es sogar eine totale Ordnung) sind in Abbildung 11.7 gezeigt. Beachten Sie bitte, dass wir auf die explizite Angabe des **BOT** verzichten – wir nehmen implizit ein **BOT** vor der ersten Operation der Transaktion an. Die sich aus der Transitivität ergebenden Reihenfolgen werden i.A. nicht explizit eingezeichnet, also z.B. die Reihenfolge $r_1(A) \rightarrow r_1(B)$, die aus $r_1(A) \rightarrow w_1(A)$ und $w_1(A) \rightarrow r_1(B)$ folgt.

11.3.2 Historie (Schedule)

Unter einer *Historie* (manchmal auch *Schedule* genannt) versteht man den Ablauf einer verzahnten Ausführung mehrerer Transaktionen. Jede einzelne Transaktion besteht aus den Elementaroperationen $r_i(A)$ und $w_i(A)$ für ein Datenobjekt A und a_i oder c_i. Die Historie spezifiziert die Reihenfolge, in der diese Elementaroperationen verschiedener Transaktionen ausgeführt werden. Man kann sich das intuitiv auch so vorstellen, dass eine Monitorkomponente (also ein „Geschichtsschreiber") protokolliert, welche Operationen der Prozessor in welcher Reihenfolge ausgeführt hat. Bei einem Einprozessorsystem werden alle Operationen sequentiell ausgeführt, so dass man eine totale Ordnung definieren kann. Es ist aber auch denkbar, dass einige Operationen „wirklich" parallel ausgeführt werden und man deshalb keine Reihenfolge festlegen kann bzw. will. Bei der Spezifikation der Historie muss aber mindestens für alle sogenannten *Konfliktoperationen* eine Reihenfolge festgelegt werden.

Was sind Konfliktoperationen? Das sind solche Operationen, die bei unkontrollierter Nebenläufigkeit potentiell Inkonsistenzen verursachen können. Das kann aber nur geschehen, wenn die Operationen auf dasselbe Datenobjekt zugreifen und mindestens eine davon das Datum modifiziert.

Betrachten wir zwei Transaktionen T_i und T_j, die beide auf das Datum A zugreifen. Dann sind folgende Operationen möglich:

- $r_i(A)$ und $r_j(A)$: In diesem Fall ist die Reihenfolge der Ausführungen irrelevant, da beide TAs in jedem Fall denselben Zustand lesen – wenn sie ohne zwischenzeitliche Änderung des Datums A aufeinander folgen. Diese beiden Operationen stehen also nicht in Konflikt zueinander, so dass in der Historie ihre Reihenfolge zueinander irrelevant ist

- $r_i(A)$ und $w_j(A)$: Hierbei handelt es sich um einen Konflikt, da T_i entweder den alten oder den neuen Wert von A liest. Es muss also entweder $r_i(A)$ *vor* $w_j(A)$ oder $w_j(A)$ *vor* $r_i(A)$ spezifiziert werden.

- $w_i(A)$ und $r_j(A)$: analog.

$$
\begin{array}{lllll}
 & & r_2(A) \rightarrow & w_2(B) \rightarrow & w_2(C) \rightarrow & c_2 \\
 & & \uparrow & \uparrow & \uparrow & \\
H = & r_3(B) \rightarrow & w_3(A) \rightarrow & w_3(B) \rightarrow & w_3(C) \rightarrow & c_3 \\
 & & \uparrow & & & \\
 & r_1(A) \rightarrow & w_1(A) \rightarrow & c_1 & &
\end{array}
$$

Abbildung 11.8: Historie für drei Transaktionen

- $w_i(A)$ und $w_j(A)$: Auch in diesem Fall ist die Reihenfolge der Ausführung entscheidend für den Zustand der Datenbasis; also handelt es sich um Konfliktoperationen, für die die Reihenfolge festzulegen ist.

Formal ist eine Historie H für eine Menge von Transaktionen $\{T_1, \ldots, T_n\}$ eine Menge von Elementaroperationen mit partieller Ordnung $<_H$, so dass gilt:

- $H = \cup_{i=1}^{n} T_i$,
- $<_H$ ist verträglich mit allen $<_i$, d.h. $<_H \supseteq \cup_{i=1}^{n} <_i$,
- für zwei Konfliktoperationen $p, q \in H$ gilt entweder $p <_H q$ oder $q <_H p$.

In Abbildung 11.8 ist eine Historie H für drei Transaktionen T_1, T_2 und T_3 gezeigt – dies sind „neue" abstrakte Transaktionen, die mit den vorher in diesem Kapitel beschriebenen nicht übereinstimmen. In diesem Beispiel ist nur eine partielle Ordnung gegeben. Es wird z.B. nicht spezifiziert, in welcher Reihenfolge $r_3(B)$ und $r_1(A)$ ausgeführt werden – dies ist natürlich nur für nicht in Konflikt stehende Operationen zulässig.

Im Allgemeinen werden wir aber eine totale Ordnung angeben, die wie folgt aussehen könnte:

$$r_1(A) \rightarrow r_3(B) \rightarrow w_1(A) \rightarrow w_3(A) \rightarrow c_1 \rightarrow w_3(B) \rightarrow \ldots$$

11.3.3 Äquivalenz zweier Historien

Zwei Historien H und H' über der gleichen Menge von Transaktionen sind äquivalent (in Zeichen $H \equiv H'$), wenn sie die Konfliktoperationen der nicht abgebrochenen Transaktionen in derselben Reihenfolge ausführen. Formaler ausgedrückt: Wenn p_i und q_j Konfliktoperationen sind und $p_i <_H q_j$ gilt, dann muss auch $p_i <_{H'} q_j$ gelten.

Die Anordnung der nicht in Konflikt stehenden Operationen ist also für die Äquivalenz zweier Historien irrelevant. Die Reihenfolge der Operationen innerhalb einer Transaktion bleibt invariant, d.h. zwei Operationen v_i und w_i der Transaktion T_i sind in H in derselben Reihenfolge auszuführen wie in H'; also $v_i <_H w_i$ impliziert $v_i <_{H'} w_i$. Betrachten wir als Beispiel zwei Überweisungstransaktionen mit der in Abbildung 11.2 gezeigten Historie. In unserer Kurznotation sieht das wie folgt aus:

$$r_1(A) \rightarrow r_2(C) \rightarrow w_1(A) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$$

Da $r_2(C)$ und $w_1(A)$ nicht in Konflikt stehen, ist der obige Schedule äquivalent zu

$$r_1(A) \rightarrow w_1(A) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$$

wobei lediglich $r_2(C)$ und $w_1(A)$ vertauscht wurden. Weiterhin steht $r_1(B)$ nicht in Konflikt mit $w_2(C)$ und $r_2(C)$, so dass wir durch zweifache Vertauschung folgenden Schedule erhalten:

$$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$$

Analog können wir $w_1(B)$ durch sukzessive Vertauschung an $w_2(C)$ und $r_2(C)$ vorbei propagieren. Letztendlich machen wir dasselbe mit c_1 und erhalten folgenden äquivalenten Schedule:

$$r_1(A) \rightarrow w_1(A) \rightarrow r_1(B) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(C) \rightarrow w_2(C) \rightarrow r_2(A) \rightarrow w_2(A) \rightarrow c_2$$

Jetzt werden die aufmerksamen Leser gemerkt haben, dass wir durch sukzessives (zielgerichtetes) Vertauschen von Operationen, die nicht in Konflikt stehen, aus dem in Abbildung 11.2 dargestellten verzahnten Schedule den seriellen Schedule aus Abbildung 11.3 generiert haben. Daraus folgt, dass diese beiden Schedules äquivalent sind.

11.3.4 Serialisierbare Historien

Die an unserem Beispiel dargestellte Vorgehensweise bildet die Grundlage der Serialisierbarkeit. Eine Historie H ist serialisierbar, wenn sie äquivalent zu einer seriellen Historie H_s ist.

11.3.5 Kriterien für Serialisierbarkeit

Wir haben oben an einem Beispiel gezeigt, wie man durch „zielgerichtetes" Vertauschen von Operationen aus einer verzahnten Historie eine serielle Historie generieren kann – falls das überhaupt möglich ist. Wir geben jetzt eine Methodik an, mit der man

1. effizient entscheiden kann, ob es eine äquivalente serielle Historie gibt und
2. in welcher Reihenfolge die Transaktionen in der äquivalenten seriellen Historie ausgeführt werden müssten.

Dazu konstruieren wir zu einer gegebenen Historie H über den in der Historie erfolgreich abgeschlossenen Transaktionen $\{T_1, \ldots, T_n\}$ den sogenannten Serialisierbarkeitsgraphen $SG(H)$. $SG(H)$ hat die Knoten $T_1, \ldots, T_n$. Für je zwei Konfliktoperationen p_i, q_j aus der Historie H mit $p_i <_H q_j$ (also p_i wird vor q_j ausgeführt) fügen wir die Kante $T_i \rightarrow T_j$ in den Graphen $SG(H)$ ein – falls es diese Kante nicht schon aus anderem Grund gibt. In Abbildung 11.9 ist eine Beispiel-Historie H mit zugehörigem Serialisierbarkeitsgraphen $SG(H)$ gezeigt. Die Kante $T_2 \rightarrow T_3$ in dem Graphen $SG(H)$ kommt z.B. durch die Anordnung der beiden Konfliktoperationen $w_3(A)$ nach $r_2(A)$ in der Historie H zustande.

$$
H = \begin{array}{ccccccc}
r_1(A) & \rightarrow & w_1(A) & \rightarrow & w_1(B) & \rightarrow & c_1 \\
 & & \uparrow & & \uparrow & & \\
 & \swarrow & r_2(A) & \rightarrow & w_2(B) & \rightarrow & c_2 \\
 & & \downarrow & & & & \\
r_3(A) & \rightarrow & w_3(A) & \rightarrow & c_3 & &
\end{array}
$$

$$
SG(H) = \begin{array}{ccc}
 & & T_3 \\
 & \nearrow & \\
T_2 & & \uparrow \\
 & \searrow & \\
 & & T_1
\end{array}
$$

Abbildung 11.9: Historie und zugehöriger Serialisierbarkeitsgraph

$$H = w_1(A) \rightarrow w_1(B) \rightarrow c_1 \rightarrow r_2(A) \rightarrow r_3(B) \rightarrow w_2(A) \rightarrow c_2 \rightarrow w_3(B) \rightarrow c_3$$

$$
SG(H) = \begin{array}{ccc}
 & & T_2 \\
 & \nearrow & \\
T_1 & & \\
 & \searrow & \\
 & & T_3
\end{array}
$$

$$
\begin{aligned}
H_s^1 &= T_1 \mid T_2 \mid T_3 \\
H_s^2 &= T_1 \mid T_3 \mid T_2 \\
H &\equiv H_s^1 \equiv H_s^2
\end{aligned}
$$

Abbildung 11.10: Serialisierbare Historie H mit zugehörigem Serialisierbarkeitsgraphen $SG(H)$ und den zwei äquivalenten seriellen Historien H_s^1 und H_s^2.

Serialisierbarkeitstheorem Das sogenannte Serialisierbarkeitstheorem besagt, dass eine Historie H genau dann *serialisierbar* ist, wenn der zugehörige Serialisierbarkeitsgraph $SG(H)$ azyklisch ist.

Weiterhin ist eine serialisierbare Historie H äquivalent zu all den seriellen Historien H_s, in denen die Anordnung der Transaktionen einer topologischen Sortierung von *SG(H)* entspricht. Unter einer topologischen Sortierung versteht man eine sequentielle Anordnung der Transaktionen des Serialisierbarkeitsgraphen in der Form, dass keine Transaktion T_i vor einer Transaktion T_j steht, wenn es einen gerichteten Pfad von T_j nach T_i im Serialisierbarkeitsgraphen gibt. Auch hierzu ist in Abbildung 11.10 ein Beispiel gegeben. Die beiden möglichen topologischen Sortierungen des Serialisierbarkeitsgraphen – nämlich $T_1 \mid T_2 \mid T_3$ und $T_1 \mid T_3 \mid T_2$ – entsprechen den beiden seriellen Historien H_s^1 und H_s^2.

11.4 Eigenschaften von Historien bezüglich der Recovery

Die Serialisierbarkeit ist eine minimale Anforderung an die im DBMS zugelassenen Schedules. Eine zusätzliche Anforderung ergibt sich aus der Recovery: Die in der Transaktionsverarbeitung zulässigen Historien sollten so gestaltet sein, dass jede Transaktion zu jedem Zeitpunkt vor Ausführung eines **commit** lokal zurückgesetzt werden kann – ohne dass davon andere Transaktionen beeinträchtigt werden.

11.4.1 Rücksetzbare Historien

Bezüglich der Recovery ist die Minimalanforderung, dass man noch aktive Transaktionen jederzeit abbrechen kann, ohne dass andere schon mit **commit** abgeschlossene Transaktionen in Mitleidenschaft gezogen werden können. Historien, die diese Eigenschaft erfüllen, nennen wir *rücksetzbare* Historien.

Um die rücksetzbaren Historien charakterisieren zu können, müssen wir zunächst die Schreib/Leseabhängigkeiten zwischen Transaktionen einführen. Wir sagen, dass in der Historie H T_i von T_j liest, wenn folgendes gilt:

1. T_j schreibt mindestens ein Datum A, das T_i nachfolgend liest, also:

$$w_j(A) <_H r_i(A)$$

2. T_j wird (zumindest) nicht vor dem Lesevorgang von T_i zurückgesetzt, also:

$$a_j \not<_H r_i(A)$$

3. Alle anderen zwischenzeitlichen Schreibvorgänge auf A durch andere Transaktionen T_k werden vor dem Lesen durch T_i zurückgesetzt. Falls also ein $w_k(A)$ mit $w_j(A) < w_k(A) < r_i(A)$ existiert, so muss es auch ein $a_k < r_i(A)$ geben.

Intuitiv ausgedrückt, besagen die drei Bedingungen, dass T_i ein Datum A in genau dem Zustand, den T_j geschrieben hat, liest.

Eine Historie heißt rücksetzbar, falls immer die schreibende Transaktion (in unserer Notation T_j) vor der lesenden Transaktion (T_i genannt) ihr **commit** durchführt, also: $c_j <_H c_i$. Anders ausgedrückt: Eine Transaktion darf erst dann ihr **commit** durchführen, wenn alle Transaktionen, von denen sie gelesen hat, beendet sind. Wäre diese Bedingung nicht erfüllt, könnte man die schreibende Transaktion womöglich nicht zurücksetzen, da die lesende Transaktion dann mit einem „offiziell" nie existenten Wert für A ihre Berechnung „**committed**" hätte – und nach Durchführung des **commit** ist eine Transaktion gemäß des ACID-Paradigmas nicht mehr rücksetzbar.

11.4.2 Historien ohne kaskadierendes Rücksetzen

Selbst rücksetzbare Historien können noch folgenden unangenehmen Effekt verursachen: Das Rücksetzen einer Transaktion setzt eine Lawine von weiteren Rollbacks in Gang. Die Historie in Abbildung 11.11 verdeutlicht dies. Die Transaktion T_1 schreibt

Schritt	T_1	T_2	T_3	T_4	T_5
0.	...				
1.	$w_1(A)$				
2.		$r_2(A)$			
3.		$w_2(B)$			
4.			$r_3(B)$		
5.			$w_3(C)$		
6.				$r_4(C)$	
7.				$w_4(D)$	
8.					$r_5(D)$
9.	a_1 **(abort)**				

Abbildung 11.11: Historie mit kaskadierendem Rücksetzen

in Schritt 1. ein Datum A, das T_2 liest. Abhängig vom gelesenen Wert von A – zumindest muss das DBMS dies annehmen – schreibt T_2 einen neuen Wert in B, der wiederum von T_3 gelesen wird. T_3 schreibt C, das von T_4 gelesen wird. T_4 schreibt D, das von T_5 gelesen wird. Jetzt, nachdem alle anderen Transaktionen T_2, T_3, T_4 und T_5 abhängig von dem von T_1 in A geschriebenen Wert geworden sind, kommt es in Schritt 9. zum **abort** der Transaktion T_1. Natürlich müssen dann auch alle anderen Transaktionen zurückgesetzt werden. In der Theorie ist dies kein Problem, da die Historie rücksetzbar ist; praktisch wird dadurch aber die Leistungsfähigkeit des Systems drastisch beeinträchtigt. Deshalb sind wir an Schedules interessiert, die kaskadierendes Rücksetzen vermeiden.

Eine Historie H vermeidet kaskadierendes Rücksetzen, wenn

- $c_j <_H r_i(A)$ gilt, wann immer T_i ein Datum A von T_j liest.

Anders ausgedrückt: Änderungen werden erst nach dem **commit** freigegeben.

11.4.3 Strikte Historien

Bei strikten Historien dürfen auch veränderte Daten einer noch laufenden Transaktion nicht überschrieben werden. Wenn also für ein Datum A die Ordnung $w_j(A) <_H o_i(A)$ mit $o_i = r_i$ oder $o_i = w_i$ gilt, dann muss T_j zwischenzeitlich mit **commit** oder **abort** abgeschlossen worden sein. Also muss entweder

- $c_j <_H o_i(A)$ oder
- $a_j <_H o_i(A)$

gelten.

11.4.4 Beziehungen zwischen den Klassen von Historien

Es gelten die in Abbildung 11.12 dargestellten Beziehungen (Inklusionen) zwischen den Historienklassen. Hierbei sind folgende Abkürzungen gebraucht worden [Bernstein, Hadzilacos und Goodman (1987)]:

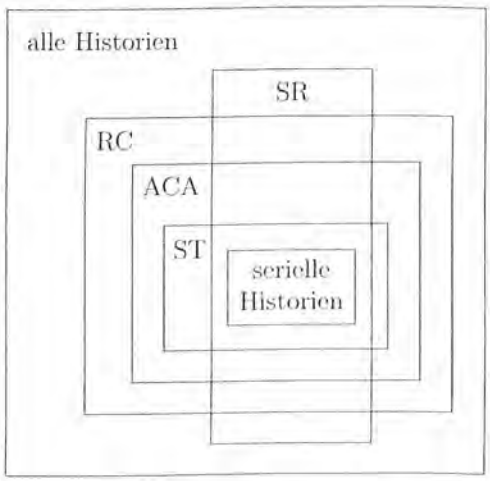


Abbildung 11.12: Beziehungen der Historienklassen zueinander

- *SR*: serialisierbare Historien (SeRializable),
- *RC*: rücksetzbare Historien (ReCoverable),
- *ACA*: Historien ohne kaskadierendes Rücksetzen (Avoiding Cascading Abort),
- *ST*: strikte Historien (STrict).

11.5 Der Datenbank-Scheduler

Hinsichtlich der Transaktionsverarbeitung können wir uns eine Datenbanksystem-Architektur mit einem *Scheduler* – stark vereinfacht – so vorstellen, wie in Abbildung 11.13 gezeigt.

Die Aufgabe des Schedulers besteht darin, die Operationen – d.h. Einzeloperationen verschiedener Transaktionen $T_1, \ldots, T_n$ – in einer derartigen Reihenfolge auszuführen, dass die resultierende Historie „vernünftig" ist. Unter einer „vernünftigen" Historie verstehen wir als Mindestanforderung die Serialisierbarkeit, aber i.A. wird vom Scheduler sogar verlangt, dass die resultierende Historie ohne kaskadierendes Rollback rücksetzbar ist. D.h., bezogen auf den vorangegangenen Abschnitt (siehe Abbildung 11.12), sollten die vom Scheduler zugelassenen Historien aus dem Bereich $ACA \cap SR$ sein.

Wir werden mehrere mögliche Techniken für die Realisierung eines Schedulers kennenlernen. Die mit Abstand bedeutendste ist die sperrbasierte Synchronisation, die in fast allen kommerziellen relationalen Systemen verwendet wird.

Weiterhin gibt es eine Zeitstempel-basierte Synchronisation, wobei jedes Datum einen Eintrag für den Zeitstempel derjenigen Transaktion erhält, die die letzte Modifikation vorgenommen hat. Diese beiden Methoden – sperrbasierte und zeitstempelbasierte Synchronisation – werden oft als *pessimistische* Verfahren eingestuft, da

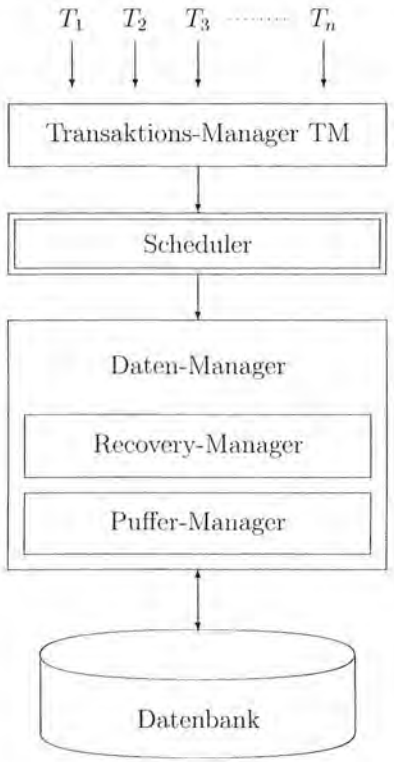


Abbildung 11.13: Die Stellung des *Schedulers* in der Datenbanksystem-Architektur

hier die Grundannahme herrscht, dass potentielle Konflikte auch tatsächlich zu einer nicht serialisierbaren Historie führen – was nicht immer der Fall ist.

Demgegenüber gibt es noch die *optimistische* Synchronisation. In diesem Fall führt der Scheduler erst mal alle Operationen aus – merkt sich aber für jede Transaktion, welche Daten gelesen und geschrieben wurden. Erst wenn die Transaktion ihr **commit** durchführen will, wird verifiziert, ob sie ein Problem – d.h. einen nicht serialisierbaren Schedule – verursacht hat. In diesem Falle wird die Transaktion zurückgesetzt.

11.6 Sperrbasierte Synchronisation

Bei der sperrbasierten Synchronisation wird während des laufenden Betriebs sichergestellt, dass die resultierende Historie serialisierbar bleibt. Dies geschieht dadurch, dass eine Transaktion erst nach Erhalt einer entsprechenden Sperre auf ein Datum zugreifen kann.

11.6.1 Zwei Sperrmodi

Je nach Operation (**read** oder **write**) unterscheiden wir zwei Sperrmodi:

- S (shared, read lock, Lesesperre): Wenn Transaktion T_i eine S-Sperre für ein Datum A besitzt, kann T_i **read**(A) ausführen. Mehrere Transaktionen können gleichzeitig eine S-Sperre auf demselben Objekt A besitzen.
- X (exclusive, write lock, Schreibsperre): Ein **write**(A) darf nur die *eine* Transaktion ausführen, die eine X-Sperre auf A besitzt.

Die Verträglichkeit von Sperranforderungen mit schon existierenden Sperren (auf demselben Objekt durch andere Transaktionen) fasst man in einer sogenannten *Verträglichkeitsmatrix* (auch *Kompatibilitätsmatrix* genannt) zusammen:

	NL	S	X
S	√	√	
X	√		

Hierbei ist in der Waagerechten die existierende Sperre – NL (no lock, also keine Sperre), S oder X – eingetragen und auf der Senkrechten die Sperranforderung. Existiert z.B. schon eine S-Sperre, dann kann eine weitere S-Sperre gewährt werden („√"-Eintrag) aber keine X-Sperre („ "-Eintrag).

11.6.2 Zwei-Phasen-Sperrprotokoll

Die Serialisierbarkeit ist bei Einhaltung des folgenden Zwei-Phasen-Sperrprotokolls (Engl. *two-phase locking*, 2PL) durch den Scheduler gewährleistet. Bezogen auf eine individuelle Transaktion wird folgendes verlangt:

1. Jedes Objekt, das von einer Transaktion benutzt werden soll, muss vorher entsprechend gesperrt werden.
2. Eine Transaktion fordert eine Sperre, die sie schon besitzt, nicht erneut an.
3. Eine Transaktion muss die Sperren anderer Transaktionen auf dem von ihr benötigten Objekt gemäß der Verträglichkeitstabelle beachten. Wenn die Sperre nicht gewährt werden kann, wird die Transaktion in eine entsprechende Warteschlange eingereiht – bis die Sperre gewährt werden kann.
4. Jede Transaktion durchläuft zwei Phasen:
   - Eine *Wachstumsphase*, in der sie Sperren anfordern, aber keine freigeben darf und
   - Eine *Schrumpfungsphase*, in der sie ihre bisher erworbenen Sperren freigibt, aber keine weiteren anfordern darf.
5. Bei EOT (Transaktionsende) muss eine Transaktion alle ihre Sperren zurückgeben.

In Abbildung 11.14 sind die beiden Phasen – Wachstums- und Schrumpfungsphase des 2PL-Protokolls visualisiert. Auf der y-Achse ist die Anzahl der von der Transaktion gehaltenen Sperren aufgezeichnet, die während der ersten Phase nur zunehmen (oder stagnieren) darf und in der zweiten Phase nur abnehmen darf.

In Abbildung 11.15 ist eine Historie zweier Transaktionen T_1 und T_2 gezeigt:

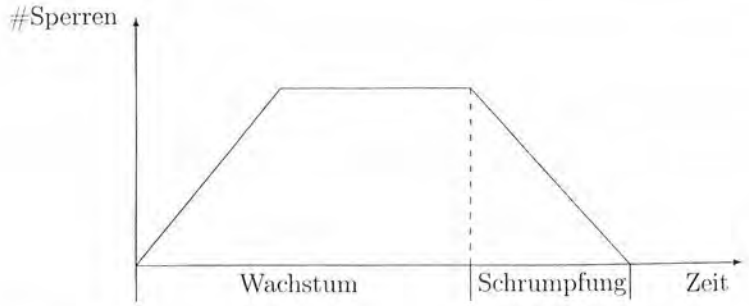


Abbildung 11.14: Zwei-Phasen Sperrprotokoll

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**	

Abbildung 11.15: Verzahnung zweier Transaktionen mit Sperranforderungen und Sperrfreigaben nach 2PL

- T_1 modifiziert nacheinander die Datenobjekte A und B (z.B. eine Überweisung)
- T_2 liest nacheinander dieselben Datenobjekte A und B (z.B. zur Aufsummierung der beiden Kontostände).

Mittels **lockX**(...) wird eine X-Sperre und mit **lockS**(...) eine S-Sperre angefordert. Mit **unlockX**(...) und **unlockS**(...) werden die Sperren wieder freigegeben. Dieser Schedule gehorcht dem Zwei-Phasen-Sperrprotokoll. Warum?

In Schritt 6. fordert T_2 eine S-Sperre für A an. Diese kann aber zu diesem Zeitpunkt nicht gewährt werden, so dass T_2 *blockiert* werden muss. Erst nach Freigabe der X-Sperre durch T_1 in Schritt 9. kann T_2 wieder aktiviert werden, indem ihre Sperranforderung erfüllt wird. Das Analoge geschieht in Schritt 11., wenn T_2 das Datum B sperren will.

Die gezeigte Historie ist natürlich serialisierbar (alle 2PL-Schedules sind serialisierbar) und entspricht der seriellen Ausführung von T_1 vor T_2 (also $T_1 \mid T_2$).

11.6.3 Kaskadierendes Rücksetzen (Schneeballeffekt)

Das Zwei-Phasen-Sperrprotokoll garantiert auf jeden Fall die Serialisierbarkeit. Aber es hat (in der bislang vorgestellten Form) gravierende Mängel: Es vermeidet nicht das kaskadierende Rollback – ja, es lässt sogar nicht-rücksetzbare Historien zu (siehe Übungsaufgabe 11.4). Schauen wir uns nochmals den Schedule aus Abbildung 11.15 an. Wenn T_1 z.B. direkt vor Schritt 15. scheitern würde, dann müsste auch T_2 zurückgesetzt werden – da T_2 von T_1 geschriebene Daten („dirty data") gelesen hat.

Die Lösung besteht darin, das 2PL-Protokoll zum sogenannten *strengen 2PL*-Protokoll wie folgt zu verschärfen:

- Die Anforderungen (1) bis (5) des 2PL-Protokolls bleiben erhalten.
- Es gibt keine Schrumpfungsphase mehr, sondern *alle* Sperren werden erst zum Ende der Transaktion (EOT) freigegeben.

Abbildung 11.16 zeigt diese verschärfte Anforderung grafisch. Unter Einhaltung des strengen 2PL-Protokolls entspricht die Reihenfolge, in der die Transaktionen beendet werden, einer äquivalenten seriellen Abarbeitungsreihenfolge (Engl. *commit order serializability*). Warum?

11.7 Verklemmungen (Deadlocks)

Leider gibt es ein schwerwiegendes und inherentes (also nicht vermeidbares) Problem mit den sperrbasierten Synchronisationsmethoden: Das Auftreten von *Verklemmungen* (Engl. *deadlocks*). Eine solche Verklemmung ist in Abbildung 11.17 gezeigt. Nach Schritt 9. ist die Ausführung der beiden Transaktionen T_1 und T_2 *verklemmt*, da T_1 auf die Freigabe einer Sperre durch T_2 wartet und umgekehrt T_2 auf die Freigabe einer Sperre durch T_1 wartet. Beide Transaktionen sind blockiert.

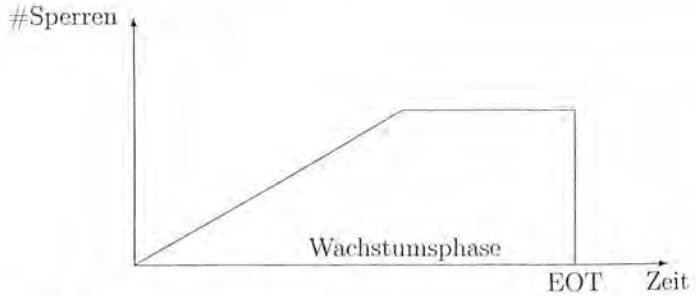


Abbildung 11.16: Strenges Zwei-Phasen Sperrprotokoll

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*

Abbildung 11.17: Ein verklemmter Schedule

11.7.1 Erkennung von Verklemmungen

Eine „brute-force"-Methode zur Erkennung von (potentiellen) Verklemmungen ist die *Time-out* Strategie. Hierbei wird ganz einfach die Ausführung der Transaktionen überwacht. Falls eine Transaktion innerhalb eines Zeitmaßes (z.B. 1 Sekunde) keinerlei Fortschritt erzielt, geht das System von einer Verklemmung aus und setzt die betreffende Transaktion zurück.

Diese Time-out Methode hat den Nachteil, dass wenn das Zeitmaß zu klein gewählt wird, zu viele Transaktionen abgebrochen werden, die tatsächlich gar nicht verklemmt waren – sondern nur auf Ressourcen (CPU, Sperren, etc.) gewartet haben. Andererseits, wenn das Zeitmaß zu groß gewählt wird, werden tatsächlich existierende Verklemmungen zu lange geduldet – wodurch die Systemauslastung beeinträchtigt werden könnte.

Eine präzise – aber auch teurere – Methode Verklemmungen zu erkennen, basiert auf einem sogenannten Wartegraphen. Die Knoten des Wartegraphen entsprechen den Kennungen der (derzeit im System aktiven) Transaktionen. Die Kanten sind gerichtet. Wann immer eine Transaktion T_i auf die Freigabe einer Sperre durch eine Transaktion T_j wartet, wird die Kante $T_i \rightarrow T_j$ eingefügt.

Eine Verklemmung liegt dann (und nur dann) vor, wenn der Wartegraph einen Zyklus aufweist. Ein solcher Zyklus muss natürlich nicht auf die Länge 2 (wie in unserem verklemmten Schedule von Abbildung 11.17) beschränkt sein, sondern kann

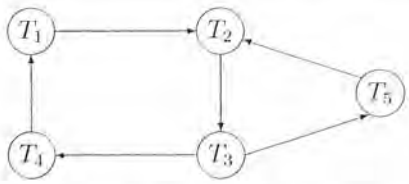


Abbildung 11.18: Wartegraph mit zwei Zyklen: $T_1 \to T_2 \to T_3 \to T_4 \to T_1$ und $T_2 \to T_3 \to T_5 \to T_2$

beliebig lang sein. Zwei derartige Zyklen sind in Abbildung 11.18 gezeigt. In der Praxis hat sich aber gezeigt, dass die weitaus größte Zahl von Verklemmungen in Datenbanken tatsächlich durch Zyklen der (minimalen) Länge 2 verursacht wird.

Eine Verklemmung wird durch das Zurücksetzen einer der im Zyklus vorkommenden Transaktionen aufgelöst. Welche der n Transaktionen man aus einem Zyklus der Länge n auswählt, kann von verschiedenen Kriterien abhängig gemacht werden:

- Minimierung des Rücksetzaufwands: Man wählt die jüngste Transaktion, oder diejenige mit den wenigsten Sperren, aus.
- Maximierung der freigegebenen Ressourcen: Man wählt die Transaktion mit den meisten Sperren aus, um die Gefahr eines nochmaligen Deadlocks zu verkleinern.
- Vermeidung von „Starvation"(Verhungern): Man muss verhindern, dass immer wieder die gleiche Transaktion zurückgesetzt wird. Man muss sich also merken, wie oft eine Transaktion schon wegen eines Deadlocks zurückgesetzt wurde und ihr ggf. einen „Freifahrschein" – also eine Markierung, die sie als zukünftiges Opfer ausschließt – geben.
- Mehrfache Zyklen: Manchmal ist eine Transaktion an mehreren Verklemmungszyklen beteiligt – wie z.B. Transaktion T_2 in Abbildung 11.18. Durch das Zurücksetzen dieser Transaktion löst man somit gleich mehrere (hier zwei) Verklemmungen auf einmal.

11.7.2 Preclaiming zur Vermeidung von Verklemmungen

Eine sehr einfache – aber leider in der Praxis meist nicht realisierbare – Methode zur Deadlock-Vermeidung besteht im sogenannten „Preclaiming". Transaktionen werden erst begonnen, wenn alle ihre Sperranforderungen schon bei Transaktionsbeginn (**BOT**) erfüllt werden können.

Dieses Preclaiming-Verfahren setzt natürlich voraus, dass eine Transaktion schon vorab „weiß", welche Datenobjekte sie benötigt und hier liegt die Krux des Verfahrens. Da die genaue Menge der benötigten Datenobjekte vom jeweiligen Kontrollfluss des Transaktionsprogramms – man denke an „**if** ... **then** ... **else** ..."-Anweisungen abhängt, muss man i.A. eine Obermenge der tatsächlich benötigten Objekte sperren. Das führt dann zu einer übermäßigen Ressourcenbelegung und damit zu einer Einschränkung der Parallelität. Abbildung 11.19 zeigt das Preclaiming in Verbindung mit dem *strengen* 2PL-Protokoll.

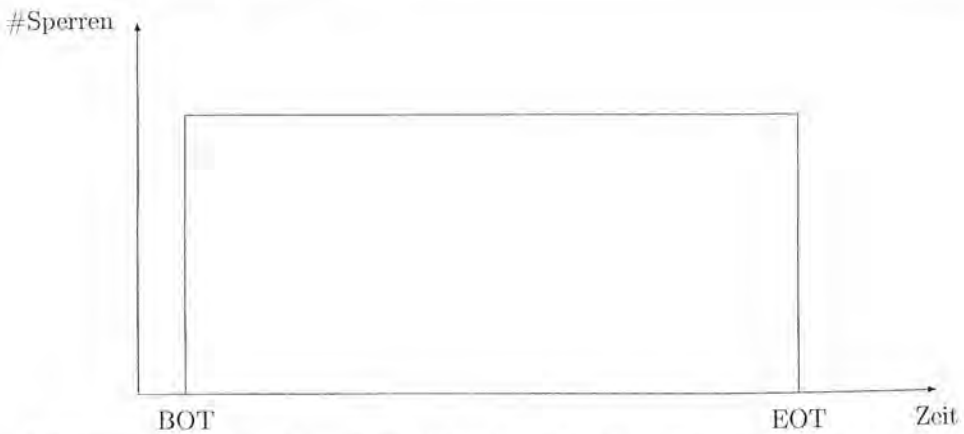


Abbildung 11.19: Preclaiming in Verbindung mit dem strengen 2 PL-Protokoll

11.7.3 Verklemmungsvermeidung durch Zeitstempel

Jeder Transaktion wird hierbei ein eindeutiger Zeitstempel TS (time stamp) zugeordnet. Die Zeitstempel werden monoton wachsend vom Transaktionsmanager vergeben, so dass eine ältere Transaktion T_a einen kleineren Zeitstempel als eine jüngere Transaktion T_j besitzt, also: $TS(T_a) < TS(T_j)$. Verklemmungen werden dadurch vermieden, dass Transaktionen nicht mehr „bedingungslos" auf die Freigabe einer Sperre durch eine andere Transaktion warten. Der Scheduler kann nach zwei – auf den ersten Blick ähnlich erscheinenden, aber hinsichtlich der Wirkung sehr unterschiedlichen – Strategien verfahren, wenn T_1 eine Sperre anfordert, die T_2 aber erst freigeben müsste:[2]

- *wound-wait*: Wenn T_1 älter als T_2 ist, wird T_2 abgebrochen und zurückgesetzt, so dass T_1 weiterlaufen kann. Sonst wartet T_1 auf die Freigabe der Sperre durch T_2.

- *wait-die*: Wenn T_1 älter als T_2 ist, wartet T_1 auf die Freigabe der Sperre. Sonst wird T_1 abgebrochen und zurückgesetzt.

Die Benennung der Strategien erfolgte jeweils aus der Sicht der Transaktion T_1, die eine Sperre anfordert. Diese Methode ist garantiert verklemmungsfrei. Warum? (siehe Übungsaufgabe 11.12).

Der Nachteil dieser Art der Verklemmungsvermeidung besteht darin, dass i.A. zu viele Transaktionen zurückgesetzt werden, ohne dass tatsächlich eine Verklemmung auftreten würde (siehe Übungsaufgabe 11.13).

Die beiden Strategien *wound-wait* und *wait-die* zeigen große Unterschiede hinsichtlich der Priorisierung einer älteren Transaktion. Bei *wound-wait* „bahnt" sich eine ältere Transaktion ihren Weg durch das System, wohingegen bei *wait-die* eine ältere Transaktion mit zunehmendem „Lebensalter" immer mehr Zeit in Warteschlangen zubringt, um auf die Freigabe von Sperren zu warten.

[2] Also fordert T_1 eine X-Sperre an oder T_2 besitzt eine X-Sperre (und T_1 fordert X oder S an).

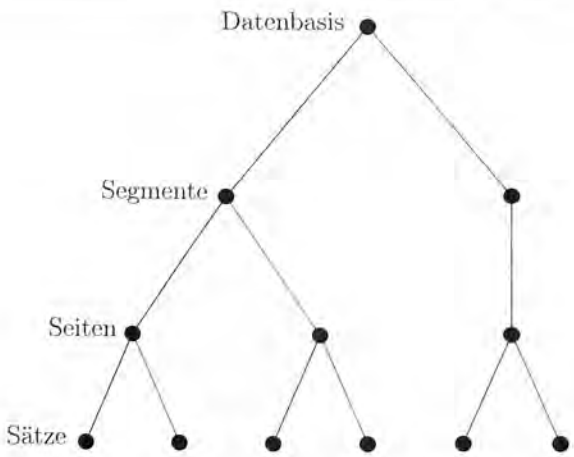


Abbildung 11.20: Hierarchische Anordnung der möglichen Sperrgranulate

11.8 Hierarchische Sperrgranulate

Bislang haben wir nur die zwei Sperrmodi S und X betrachtet. Weiterhin waren wir bislang davon ausgegangen, dass alle Sperren auf derselben „Granularität" erworben werden. Mögliche Sperrgranulate sind:

- *Datensätze*: Ein Datensatz (Tupel) ist i.A. die kleinste Sperreinheit, die in einem Datenbanksystem angeboten wird. Transaktionen, die auf sehr viele Datensätze zugreifen, müssen bei dieser Sperrgranularität einen hohen Sperraufwand in Kauf nehmen.

- *Seiten*: In diesem Fall werden durch Vergabe einer Sperre implizit alle auf der Seite gespeicherten Datensätze gesperrt.

- *Segmente*: Ein Segment ist eine logische Einheit von mehreren (i.A. vielen) Seiten. Werden Segmente als Sperreinheit gewählt, wird natürlich die Parallelität drastisch eingeschränkt, da bei einer X-Sperrung implizit alle Seiten innerhalb des betreffenden Segments exklusiv gesperrt werden.

- *Datenbasis*: Dies ist der Extremfall, da dadurch die serielle Abarbeitung aller Änderungstransaktionen erzwungen wird.

In Abbildung 11.20 sind diese hierarchisch miteinander in Beziehung stehenden Sperrgranulate grafisch dargestellt. Bezogen auf Abbildung 11.20 waren wir bislang davon ausgegangen, dass alle Transaktionen ihre Sperren auf derselben Hierarchiestufe – also auf der Ebene der Sätze, Seiten, Segmente oder Datenbasis – anfordern. Überlegen wir uns, welche Auswirkungen die Vermischung der Sperrgranulate hätte: Nehmen wir an, Transaktion T_1 will das „linke" Segment exklusiv sperren. Dann müsste man alle Sperren auf Seitenebene durchsuchen, um zu überprüfen ob nicht irgendeine andere Transaktion eine in dem Segment enthaltene Seite gesperrt hat. Gleichfalls muss man alle Sperren auf der Satz-Ebene durchsuchen, ob nicht ein

Satz, der in einer Seite des Segments steht, von einer anderen Transaktion gesperrt ist. Dieser Suchaufwand ist so immens, dass sich diese einfache Vermischung von Sperrgranulaten verbietet. Andererseits hat die Beschränkung auf nur eine Sperrgranularität für alle Transaktionen auch entscheidende Nachteile:

- Bei zu kleiner Granularität werden Transaktionen mit hohem Datenzugriff stark belastet, da sie viele Sperren anfordern müssen.
- Bei zu großer Granularität wird der Parallelitätsgrad des Systems unnötig eingeschränkt, da implizit zu viele Datenobjekte unnötigerweise gesperrt werden – es werden also Datenobjekte implizit gesperrt, die gar nicht benötigt werden.

Die Lösung des Problems besteht in der Einführung zusätzlicher Sperrmodi, wodurch die flexible Auswahl eines bestimmten Sperrgranulats pro Transaktion ermöglicht wird. Dieses Verfahren wird wegen der flexiblen Wahl der Sperrgranularität in der englischsprachigen Literatur als *multiple-granularity locking* (MGL) bezeichnet.

Die zusätzlichen Sperrmodi bezeichnet man als *Intentionssperren*, da dadurch auf höheren Ebenen der Sperrgranulatshierarchie die Absicht einer weiter unten in der Hierarchie gesetzten Sperre angezeigt wird. Die Sperrmodi sind:

- *NL*: keine Sperrung (no lock),
- *S*: Sperrung durch Leser,
- *X*: Sperrung durch Schreiber,
- *IS* (intention share): Weiter unten in der Hierarchie ist eine Lesesperre (*S*) beabsichtigt,
- *IX* (intention exclusive): Weiter unten in der Hierarchie ist eine Schreibsperre (*X*) beabsichtigt.

Die Kompatibilität dieser Sperrmodi zueinander ist in der folgenden Kompatibilitätsmatrix aufgeführt (in der Horizontalen ist die derzeitige Sperre eines Objekts angegeben, in der Vertikalen die – von einer anderen Transaktion – angeforderte Sperre):

	NL	*S*	*X*	*IS*	IX
S	√	√	–	√	–
X	√	–	–	–	–
IS	√	√	–	√	√
IX	√	–	–	√	√

Die Sperrung eines Datenobjekts muss dann so durchgeführt werden, dass erst geeignete Sperren in allen übergeordneten Knoten in der Hierarchie erworben werden. D.h. die Sperrung verläuft „top-down" und die Freigabe „bottom-up" nach folgenden Regeln:

1. Bevor ein Knoten mit *S* oder *IS* gesperrt wird, müssen alle Vorgänger in der Hierarchie vom Sperrer (also der Transaktion, die die Sperre anfordert) im *IX*- oder *IS*- Modus gehalten werden.

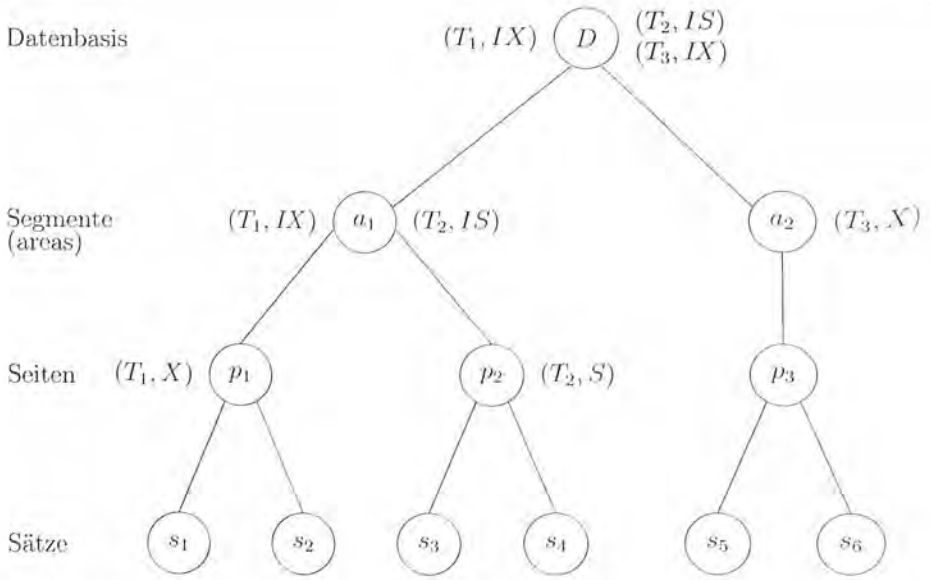


Abbildung 11.21: Datenbasis-Hierarchie mit Sperren

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 (bottom up) freigegeben, so dass bei keinem Knoten die Sperre freigegeben wird, wenn die betreffende Transaktion noch Nachfolger dieses Knotens gesperrt hat.

Wenn das strenge 2-Phasen-Sperrprotokoll befolgt wird, werden Sperren natürlich erst am Ende der Transaktion freigegeben. Anhand von Abbildung 11.21 wollen wir das Sperrprotokoll illustrieren. Sperren sind hier mit (T_i,M) bezeichnet, wobei T_i die Transaktion und M den Sperrmodus darstellt. Dazu betrachten wir drei Transaktionen:

- T_1 will die Seite p_1 exklusiv sperren und muss dazu zunächst IX-Sperren auf der Datenbasis D und auf a_1 (den beiden Vorgängern von p_1) besitzen.

- T_2 will die Seite p_2 mit einer S-Sperre belegen, wozu T_2 erst IS-Sperren oder IX-Sperren auf den beiden Vorgänger-Knoten D und a_1 anfordert. Da IS mit den an T_1 vergebenen IX-Sperren kompatibel ist, können diese Sperren gewährt werden.

- T_3 will das Segment a_2 mit X sperren und fordert IX für D an, um danach die X-Sperre auf a_2 zu bekommen. Damit hat T_3 dann alle Objekte unterhalb von a_2 – hier die Seite p_3 mit den Datensätzen s_5 und s_6 – implizit mit X gesperrt.

Die Abbildung 11.21 zeigt den Zustand zu diesem Zeitpunkt – nachdem alle Sperranforderungen der drei Transaktionen erfüllt wurden.

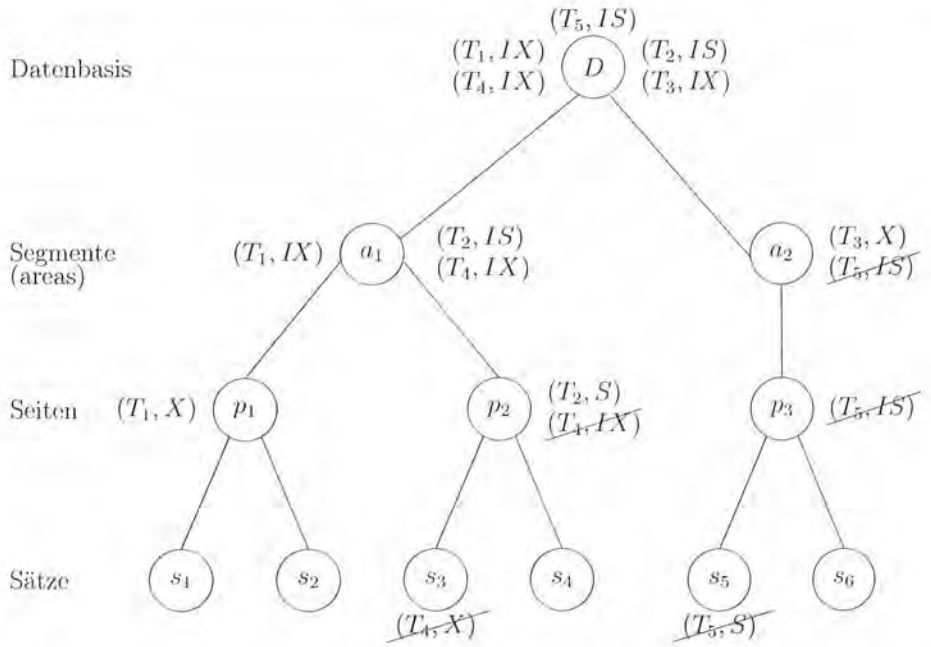


Abbildung 11.22: Datenbasis-Hierarchie mit zwei blockierten Transaktionen T_4 und T_5.

Wir wollen nun noch zwei weitere Transaktionen T_4 (Schreiber) und T_5 (Leser) betrachten, deren Sperranforderungen in dem aktuell herrschenden Zustand nicht gewährt werden können.

- T_4 will den Datensatz s_3 exklusiv sperren. Dazu wird T_4 zunächst IX-Sperren für D, a_1 und p_2 – in dieser Reihenfolge – anfordern. Die IX-Sperren für D und a_1 können gewährt werden, da sie mit den dort existierenden Sperren IX und IS kompatibel sind – laut Kompatibilitätsmatrix. Aber die IX-Sperre auf p_2 kann nicht gewährt werden, da IX nicht mit S verträglich ist.

- T_5 will eine S-Sperre auf s_5 erwerben. Dazu wird T_5 IS-Sperren auf D, a_2 und p_3 erwerben müssen. Nur die IS-Sperre auf D ist mit den existierenden Sperren verträglich, wohingegen die auf a_2 benötigte IS-Sperre nicht mit der von T_3 gesetzten X-Sperre kompatibel ist.

Die Abbildung 11.22 zeigt den Zustand nach den oben beschriebenen erfüllten Sperranforderungen. Die noch ausstehenden Sperren sind durch die Durchstreichung gekennzeichnet. Die Transaktionen T_4 und T_5 sind blockiert aber nicht verklemmt und müssen auf die Freigabe der Sperren (T_2, S) auf p_2 bzw. (T_3, X) auf a_2 warten. Erst danach können die beiden Transaktionen T_4 und T_5 mit ihren Sperranforderungen von oben nach unten fortfahren und sukzessive die „durchgestrichenen" Sperren erwerben.

Beim MGL-Sperrverfahren können – obwohl das in diesem Beispiel nicht der Fall ist – durchaus Verklemmungen auftreten (siehe Übungsaufgabe 11.16).

Aus den Beispielen sollte deutlich geworden sein, dass man zu einem gegebenen Knoten in der Datenbasis-Hierarchie alle Sperren verwalten muss. Wenn z.B. in Abbildung 11.21 T_1 die IX-Sperre auf a_1 freigibt, muss der Knoten weiterhin im IS-Modus für T_2 gesperrt bleiben. Deshalb muss man bei einer Sperranforderung im Prinzip die angeforderte Sperre mit allen am Knoten gesetzten Sperren hinsichtlich Kompatibilität überprüfen. Man kann dies aber beschleunigen, indem jedem Knoten ein Gruppenmodus zugewiesen wird. Dazu werden die zueinander kompatiblen Sperren geordnet. Es gilt:

$$S > IS$$
$$IX > IS$$

Alle anderen Sperrmodi können laut Kompatibilitätsmatrix nicht gleichzeitig an demselben Knoten gehalten werden – und brauchen demnach auch nicht geordnet zu werden. Der Gruppenmodus stellt dann die größte (d.h. schärfste) am Knoten gehaltene Sperre dar, und neu eintreffende Sperranforderungen brauchen nur gegen diesen Gruppenmodus auf Verträglichkeit überprüft zu werden.

In der Literatur wurde für das MGL-Sperrverfahren noch ein zusätzlicher Sperrmodus SIX vorgeschlagen, der einen Knoten im S-Modus und gleichzeitig im IX-Modus sperrt. Dieser Modus ist vorteilhaft für Transaktionen die einen Unterbaum der Hierarchie vollständig (oder zumindest zu großen Teilen) lesen, aber nur wenige Daten in diesem Unterbaum modifizieren. Der Sperrmodus SIX erlaubt parallel arbeitenden Transaktionen den Sperrmodus IS, so dass diese Transaktionen gleichzeitig die Daten lesen können, die von der „SIX-Transaktion" nicht modifiziert werden. Die Erweiterung des MGL-Sperrverfahren um diesen Sperrmodus ist Gegenstand der Übungsaufgabe 11.17.

Zusammenfassend erlaubt das MGL-Sperrverfahren den Transaktionen mit geringem Datenaufkommen auf niedriger Hierarchieebene – also in kleiner Granularität – zu sperren, um dadurch die Parallelität zu erhöhen. Transaktionen mit großem Datenvolumen erwerben ihre Sperren auf entsprechend höherer Hierarchieebene – also in größerer Granularität –, um dadurch den Sperraufwand zu reduzieren. Bei einigen Systemen wird automatisch von einer niedrigen Granularität auf die nächst-höhere Granularität umgeschaltet, sobald eine bestimmte Anzahl von Sperren in der kleineren Granularität erworben wurde. Diesen Vorgang nennt man im Englischen „lock escalation".

11.9 Einfüge- und Löschoperationen, Phantome

Es ist klar, dass man auch Einfüge- und Löschoperationen in die Mehrbenutzersynchronisation einbeziehen muss. Die naheliegende Methode besteht in folgendem Vorgehen:

- Vor dem Löschen eines Objekts muss die Transaktion eine X-Sperre für dieses Objekt erwerben. Man beachte aber, dass eine andere TA, die für dieses Objekt ebenfalls eine Sperre erwerben will, diese nicht mehr erhalten kann, falls die Löschtransaktion erfolgreich (mit **commit**) abschließt.

- Beim Einfügen eines neuen Objekts erwirbt die einfügende Transaktion eine X-Sperre.

In beiden Fällen muss die Sperre gemäß des strengen 2PL-Protokolls bis zum Ende der TA gehalten werden.

Diese einfache Erweiterung des Synchronisationsverfahrens schützt Transaktionen leider nicht gegen das sogenannte *Phantomproblem* – siehe dazu auch Abschnitt 11.1.3. Dieses Problem entsteht beispielsweise, wenn während der Abarbeitung einer Transaktion neue Datenobjekte in die Datenbank eingefügt werden. Als Beispiel betrachte man folgende SQL-Anweisungen:

T_1	T_2
select count(∗) **from** prüfen	
where Note **between** 1 **and** 2;	
	insert into prüfen
	values(29555, 5001, 2137, 1);
select count(∗) **from** prüfen	
where Note **between** 1 **and** 2;	

Falls Sperren nur tupelweise vergeben worden sind, kann T_2 diese Einfügeoperation verzahnt mit der Anfragebearbeitung von T_1 ausführen. Bei der zweiten Ausführung der Anfrage von T_1 würde dann ein anderer Wert ermittelt, da ja jetzt T_2's Einfügeoperation erfolgreich abgeschlossen ist. Dies widerspricht sicherlich der Serialisierbarkeit! Das gleiche Problem könnte auch dadurch auftreten, dass T_2 eine Note von bspw. 3,0 in 2,0 ändert – also ist das Phantomproblem nicht nur auf Einfügeoperationen begrenzt.

Das Problem lässt sich dadurch lösen, dass man zusätzlich zu den Tupeln auch den Zugriffsweg, auf dem man zu den Objekten gelangt ist, sperrt. Wenn man z.B. über einen Index die Objekte gefunden hat, muss man zusätzlich zu den Tupelsperren noch Indexbereichssperren setzen. Wenn also ein Index für das Attribut *Note* existiert, würde der Indexbereich $[1, 2]$ für T_1 mit einer S-Sperre belegt. Indexe müssen im Zuge von Einfüge- und Änderungsoperationen natürlich fortgeschrieben werden. Wenn jetzt also Transaktion T_2 versucht, das Tupel [29555, 5001, 2137, 1] in *prüfen* einzufügen, wird die TA blockiert, da sie die notwendige X-Sperre für den Indexbereich nicht erlangen kann – darauf hat T_1 ja schon eine S-Sperre. Es reicht aber nicht aus, nur diese Indexsperren zu erwerben; man muss zusätzlich die Sperren auf den Tupeln erwerben – denn nicht alle Zugriffe gehen über den betreffenden Index.

11.10 Zeitstempel-basierende Synchronisation

Wir hatten Zeitstempel schon in Abschnitt 11.7.3 kennengelernt. Dort wurden Sie in Verbindung mit dem Sperrverfahren für die Vermeidung von Verklemmungen eingesetzt. Wir werden jetzt ein Verfahren vorstellen, bei dem die Synchronisation ohne Sperren nur auf der Basis von Zeitstempelvergleichen durchgeführt wird.

Jeder Transaktion wird zu Beginn ein Zeitstempel TS zugewiesen, so dass ältere Transaktionen einen (echt) kleineren Zeitstempel haben als jüngere Transaktionen.

Jedem Datum A in der Datenbasis werden bei diesem Synchronisationsverfahren zwei Marken zugeordnet:

1. *readTS*(A): Diese Marke enthält den Zeitstempelwert der jüngsten Transaktion, die dieses Datum A gelesen hat.
2. *writeTS*(A): In dieser Marke wird der Zeitstempel der jüngsten Transaktion vermerkt, die das Datum A geschrieben hat.

Die Synchronisation einer Menge von Transaktionen wird dann so durchgeführt, dass immer ein Schedule entsteht, der zu einer seriellen Abarbeitung der Transaktionen in Zeitstempel-Reihenfolge äquivalent ist. Um das zu garantieren, muss der Scheduler vor Durchführung einer Lese- oder Schreiboperation durch Transaktion T_i auf dem Datum A – also $r_i(A)$ bzw. $w_i(A)$ – zunächst den Zeitstempel $TS(T_i)$ mit den A zugeordneten Marken vergleichen. Wir unterscheiden zwischen Lesen (read) und Schreiben (write):

- T_i will A lesen, also $r_i(A)$
  - Falls $TS(T_i) < writeTS(A)$ gilt, haben wir ein Problem: Die Transaktion T_i ist älter als eine andere Transaktion, die A schon geschrieben hat. Also muss T_i zurückgesetzt werden.
  - Anderenfalls, wenn also $TS(T_i) \geq writeTS(A)$ gilt, kann T_i ihre Leseoperation durchführen und die Marke $readTS(A)$ wird auf $max(TS(T_i), readTS(A))$ gesetzt.
- T_i will A schreiben, also $w_i(A)$
  - Falls $TS(T_i) < readTS(A)$ gilt, gab es eine jüngere Lesetransaktion, die den neuen Wert von A, den T_i gerade beabsichtigt zu schreiben, hätte lesen müssen. Also muss T_i zurückgesetzt werden.
  - Falls $TS(T_i) < writeTS(A)$ gilt, gab es eine jüngere Schreibtransaktion. D.h. T_i beabsichtigt einen Wert einer jüngeren Transaktion zu überschreiben. Das muss natürlich verhindert werden, so dass T_i auch in diesem Fall zurückgesetzt werden muss.
  - Anderenfalls darf T_i das Datum A schreiben und die Marke $writeTS(A)$ wird auf $TS(T_i)$ gesetzt.

Bei dieser Synchronisationsmethode muss man dennoch darauf achten, dass geänderte, aber noch nicht festgeschriebene Daten, nicht gelesen bzw. überschrieben werden. Warum? Dies kann durch die Zuordnung eines („dirty") Bits geschehen, das so lange gesetzt bleibt, bis das Datenobjekt festgeschrieben (**committed**) ist. Solange das Bit gesetzt ist, werden Zugriffe anderer Transaktionen verzögert.

Die letzte Bedingung für Schreiboperationen kann man noch optimieren, so dass weniger Abbrüche stattfinden – siehe dazu die Übungsaufgabe 11.19. Eine zurückgesetzte Transaktion wird – anders als in dem in Abschnitt 11.7.3 beschriebenen Verfahren! – mit einem neuen (d.h. dem aktuell größten) Zeitstempel neu gestartet.

Die Leser mögen verifizieren, dass diese Methode serialisierbare Schedules garantiert und verklemmungsfrei arbeitet (siehe Übungsaufgabe 11.18).

Diese Zeitstempel-basierende Synchronisation liefert also Schedules, die äquivalent zu einer seriellen Abarbeitung der Transaktionen in Zeitstempel-Reihenfolge sind. Demgegenüber liefert die strenge 2PL-Synchronisation Schedules, die zu der seriellen Abarbeitung der Transaktionen in **commit**-Reihenfolge äquivalent sind. Die Leser mögen dies verifizieren.

11.11 Optimistische Synchronisation

Die bisher betrachteten Synchronisationsmethoden werden als *pessimistische* Verfahren bezeichnet, da sie von der Prämisse ausgehen, dass Mehrbenutzerkonflikte auftreten werden. Deshalb werden Vorkehrungen getroffen, diese potentiellen Konflikte zu verhindern – in manchen Fällen auf Kosten der Parallelität, da auch einige serialisierbare Schedules „abgewiesen" werden.

Bei der *optimistischen* Synchronisation geht man davon aus, dass Konflikte selten auftreten und man Transaktionen einfach mal ausführen sollte und im Nachhinein (à posteriori) entscheidet, ob ein Mehrbenutzerkonflikt aufgetreten ist oder nicht. In diesem Fall kommt dem Scheduler eine Art Beobachterrolle (Protokollant) während der Ausführung zu. Auf der Basis der (protokollierten) Beobachtungen wird dann entschieden, ob die Ausführung der betreffenden Transaktion konfliktfrei war oder nicht. Im Falle eines Konflikts wird die Transaktion zurückgesetzt – also nachdem die komplette Arbeit verrichtet worden ist. Diese optimistische Art der Synchronisation eignet sich besonders für Datenbank-Anwendungen mit einer Mehrzahl von Lesetransaktionen, die sich sowieso nicht gegenseitig „stören" können.

Es gibt viele unterschiedliche Varianten der optimistischen Synchronisation. Wir werden hier nur eine Methode vorstellen. Eine Transaktion wird dabei in drei Phasen aufgeteilt:

1. *Lesephase*: In dieser Phase werden alle Operationen der Transaktion ausgeführt – also auch die Änderungsoperationen. Gegenüber der Datenbasis tritt die Transaktion in dieser Phase aber nur als Leser in Erscheinung, da alle gelesenen Daten in lokalen Variablen der Transaktion gespeichert werden und alle Schreiboperationen (zunächst) auf diesen lokalen Variablen ausgeführt werden. Es muss sichergestellt werden, dass alle Daten in dem Zustand gelesen werden, der zum Startzeitpunkt der Transaktion gültig war. Gegebenenfalls muss man einen zwischenzeitlich geänderten Datensatz für diese Transaktion mithilfe des Logs wiederherstellen.

2. *Validierungsphase*: In dieser Phase wird entschieden, ob die Transaktion möglicherweise in Konflikt mit anderen Transaktionen geraten ist. Dies wird anhand von Zeitstempeln entschieden, die den Transaktionen in der Reihenfolge zugewiesen werden, in der sie in die Validierungsphase eintreten.

3. *Schreibphase*: Die Änderungen der Transaktionen, bei denen die Validierung positiv verlaufen ist, werden in dieser Phase in die Datenbank eingebracht.

Transaktionen, bei denen die Validierung scheitert, werden zurückgesetzt, und die Schreibphase entfällt. Da sie bis zur Validierungsphase noch keine Änderungen am Datenbestand verursacht haben, können keine anderen Transaktionen von einer gescheiterten Transaktion in Mitleidenschaft gezogen worden sein. Es gibt also kein kaskadierendes Zurücksetzen.

Die Validierung einer Transaktion T_j geht wie folgt vonstatten. Man muss *alle* Transaktionen T_a betrachten, die älter sind als T_j, also $TS(T_a) < TS(T_j)$. Das sind, wie gesagt, nicht unbedingt Transaktionen die früher als T_j gestartet wurden, sondern Transaktionen, die vor T_j die Validierungsphase erreicht haben. Erst zu diesem Zeitpunkt werden die Zeitstempel vergeben. Für jede derartige Transaktion T_a muss – bezogen auf T_j – mindestens eine von zwei Bedingungen erfüllt sein:

1. T_a war zum Beginn der Transaktion T_j schon abgeschlossen – einschließlich der Schreibphase.

2. Die Menge der von T_a geschriebenen Datenelemente, genannt $WriteSet(T_a)$, enthält keine Elemente der Menge der von T_j gelesenen Datenelemente, genannt $ReadSet(T_j)$. Es muss also gelten:

$$WriteSet(T_a) \cap ReadSet(T_j) = \emptyset$$

Nur wenn für alle älteren Transaktionen mindestens eine der beiden Bedingungen erfüllt ist, darf T_j **commit**ten und in die Schreibphase übergehen. Anderenfalls wird T_j zurückgesetzt.

Für die Korrektheit dieser Synchronisationsmethode ist zudem erforderlich, dass die Validierungs- und Schreibphase ununterbrechbar durchgeführt werden, damit sich nicht zwei Schreibvorgänge ins „Gehege kommen". Mit anderen Worten, das System sollte immer nur eine Transaktion gleichzeitig in die Validierungs- und Schreibphase lassen.

11.12 Snapshot Isolation

Die Snapshot Isolation stellt eine „relaxierte" Synchronisationsmethode dar, die **keine** Serialisierbarkeit garantiert. Sie wird dennoch von vielen Systemen verwendet, da sie zu weniger Transaktionsabbrüchen bei der optimistischen Synchronisation führt. Die Validierung überprüft dann nicht mehr den vollständigen *Read Set* der zu validierenden Transaktion, sondern nur deren *Write Set*. In der oben eingeführten Notation wird also nur noch sichergestellt, dass gilt:

$$WriteSet(T_a) \cap WriteSet(T_j) = \emptyset$$

Es sei den Lesern überlassen, sich Beispiele auszudenken, bei denen die relaxierte Validierung zu nicht-serialisierbaren Historien führt – und deshalb möglicherweise auch „schlimme" Folgen haben könnte.

11.13 Klassifizierung der Synchronisations- und Deadlockbehandlungs-Verfahren

In Abbildung 11.23 haben wir versucht, die Leser bei der (mentalen) Einordnung der Vielzahl von Synchronisations- und Deadlockbehandlungs-Verfahren zu unterstützen. Man unterscheidet zwischen *pessimistischen* Verfahren, die „präventiv" mit Sperren oder Zeitstempeln arbeiten, und *optimistischen* Verfahren, die erst zum Schluss kontrollieren ob „alles gut gegangen" ist.

Die Deadlockbehandlung ist nur bei den sperrbasierten Synchronisationsverfahren notwendig. Hier wird zwischen den Vermeidungs- und den Erkennungsverfahren unterschieden.

11.14 Synchronisation von Indexstrukturen

Es wäre theoretisch möglich, Indexstrukturen genauso wie „normale" Daten zu behandeln. Dann würden die Datensätze eines Indexes – also z.B. Knoten eines B^+-Baums – denselben Synchronisations- und Recoverytechniken unterliegen, wie die anderen Datensätze eines DBMS. Diese Vorgehensweise ist aber i.A. zu aufwendig für Indexstrukturen:

- Indices enthalten redundante, d.h. aus dem „normalen" Datenbestand abgeleitete Informationen. Deshalb kann man abgeschwächte – und daher weniger aufwendige – Recoverytechniken einsetzen.
- Für die Mehrbenutzersynchronisation ist das Zweiphasen-Sperrprotokoll – das am häufigsten eingesetzte Synchronisationsverfahren für normale Datenbestände – zu aufwendig. Aus der speziellen Bedeutung der Indexeinträge lassen sich abgeschwächte Synchronisationstechniken konzipieren, die mehr Parallelität gewähren.

Wir wollen hier exemplarisch die sperrbasierte Mehrbenutzersynchronisation für B^+-Bäume diskutieren. Unter Verwendung des strengen Zweiphasen-Sperrprotokolls würde ein Lesezugriff auf einen B^+-Baum alle Knoten auf dem Weg von der Wurzel bis zum Blatt für die Dauer der Transaktion mit einer Lesesperre versehen. Dann wären zwar noch beliebig viele andere Lesevorgänge auf dem B^+-Baum möglich, aber Einfügeoperationen wären auf diesem Teil des B^+-Baums für die Dauer der Lesetransaktion nicht möglich. Besonders gravierend wäre das bei Bereichsanfragen, die über eine große Anzahl von Blattknoten eines B^+-Baums hinweg ausgewertet würden – alle diese Knoten wären dann mit einer Lesesperre versehen.

Eine analoge Konstellation ergibt sich bei einem Einfüge- oder Löschvorgang. Dann wäre mindestens ein Knoten – u.U. auch mehrere Knoten – exklusiv gesperrt, so dass auf diesen Knoten des Baums für die Dauer der Änderungstransaktion gar keine Lesevorgänge ausgeführt werden könnten.

Unter Berücksichtigung der speziellen Semantik von B^+-Bäumen kann man jedoch ein Sperrprotokoll einsetzen, das Sperren früher – also vor EOT der jeweiligen Transaktion – wieder freigibt. Dies erfordert jedoch eine kleine Modifikation der Knotenstruktur: Die Knoten einer Stufe des Baumes werden über sogenannte

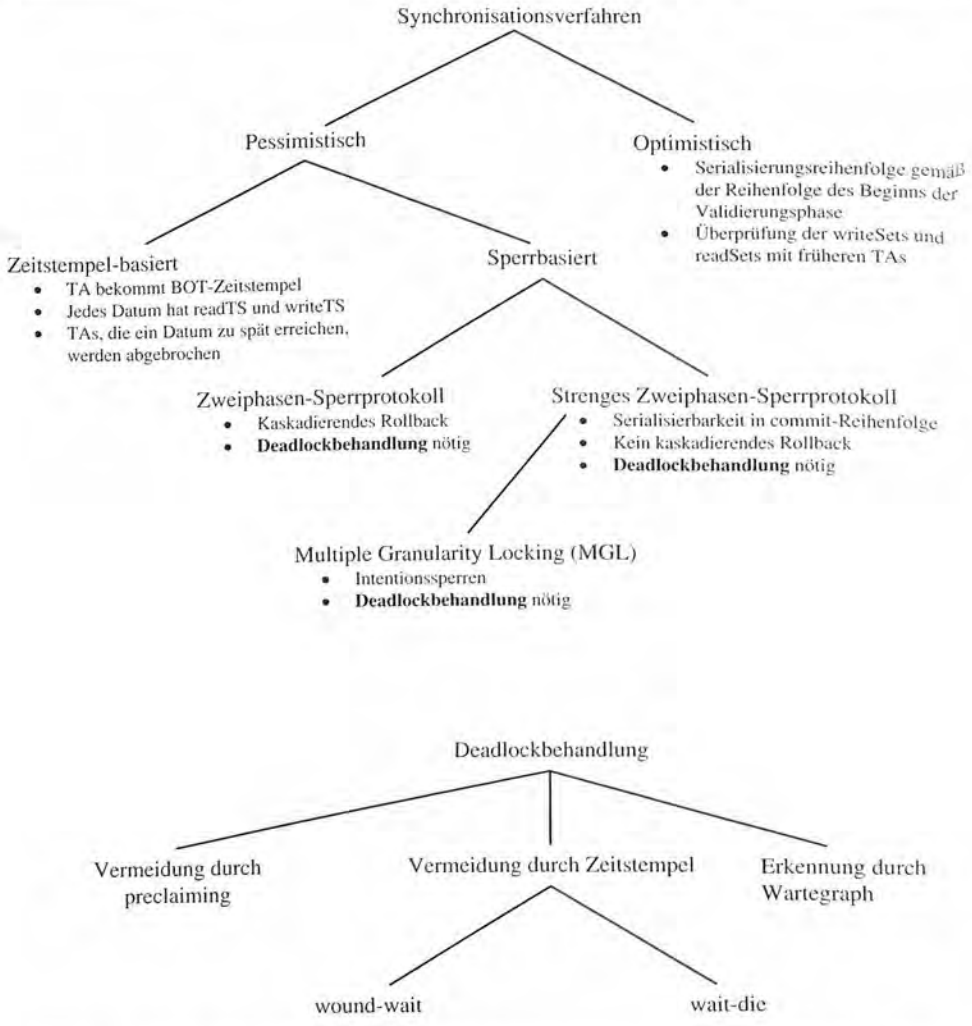


Abbildung 11.23: Klassifizierung der Synchronisations- und Deadlockbehandlungs-Verfahren

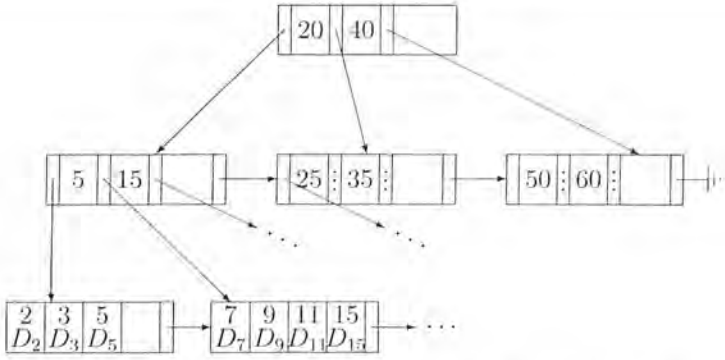


Abbildung 11.24: B^+-Baum mit *rechts*-Verweisen zur Synchronisation

rechts-Verweise miteinander verkettet. D.h., ein Knoten ist über den *rechts*-Verweis mit seinem nächsten Geschwisterknoten verkettet. Dies ist exemplarisch in Abbildung 11.24 gezeigt.

Die Operationen auf dem B^+-Baum werden dann wie folgt durchgeführt.

Suche Man startet eine Suche, indem für die Wurzel des B^+-Baums eine (kurze) Lesesperre angefordert wird. Dann ermittelt man den Weg (Zeiger) zum Knoten der nächst-niedrigeren Ebene des Baums und gibt die Lesesperre wieder frei. Für den nächsten zu lesenden Knoten wird wiederum eine (kurze) Lesesperre angefordert, um das Intervall, in das der Suchschlüssel fällt, aufzusuchen. Falls dieses Intervall gefunden wird, kann die Lesesperre wieder freigegeben werden, und die Suche geht zur nächsten Ebene über. Es kann aber vorkommen, dass in der Zwischenzeit eine (oder auch mehrere) Einfügeoperationen zum Überlauf – und damit zur Spaltung – des aktuellen Knotens geführt haben. In diesem Fall wird das Intervall u.U. nicht in dem gerade aktuellen Knoten gefunden, sondern befindet sich in einem der rechten Geschwisterknoten. Dann muss zunächst für den direkten rechten Geschwisterknoten eine Lesesperre angefordert werden, und die Lesesperre des anderen Knotens kann freigegeben werden. Die Suchoperation geht dann vom rechten Geschwisterknoten aus in gleicher Form weiter. Entweder wird das Suchintervall hier gefunden und man kann eine Stufe im Baum weitergehen oder man muss nochmals den *rechts*-Verweis verfolgen – dieser Fall kann natürlich nur auftreten, wenn in der Zwischenzeit sehr viele Einfügungen stattgefunden haben, die eine wiederholte Spaltung von Knoten bewirkt haben.

Auch auf der Blattebene muss eine Lesesperre gesetzt werden, und auch hier kann es aufgrund eines verzahnten Einfügevorgangs mit einhergehender Seitenspaltung notwendig sein, die rechten Geschwisterknoten mit in die Suche einzubeziehen.

Einfügen Zum Zweck des Einfügens wird zunächst die Seite (d.h. der Blattknoten) gesucht, in die das neue Datum einzufügen ist. Die während der Suche erworbene Lesesperre auf diesem Blattknoten muss zu einer exklusiven Schreibsperre konvertiert werden – dazu muss man natürlich die Beendigung möglicher anderer paralleler Lesezugriffe abwarten. Falls ausreichend Platz auf der Seite existiert, kann das neue

Datum eingetragen und die Schreibsperre aufgegeben werden. Im Falle eines Überlaufs muss die Operation *Seitenspaltung* durchgeführt werden.

Seitenspaltung Diese Operation wird aufgerufen, falls eine Einfüge-Operation auf eine vollständig belegte Blattseite stößt. Diese Blattseite ist demnach schon mit einer exklusiven Schreibsperre versehen. Es wird ein neuer Knoten angelegt, und (etwa) die Hälfte der Einträge der vollen Seite werden auf diesen neuen Knoten transferiert. Die *rechts*-Verweise der beiden Knoten werden gesetzt – d.h. der *rechts*-Verweis der ursprünglich vollen Seite wird auf die neue Seite gesetzt und der *rechts*-Verweis der neuen Seite auf den alten *rechts*-Verweis der ursprünglich vollen Seite. Danach kann die Sperre auf dem ursprüglich vollen Knoten aufgegeben und eine Schreib-Sperre im Vaterknoten angefordert werden, damit ein Verweis auf die neu eingefügte Seite eingebaut werden kann.[3]

Es kann natürlich vorkommen, dass auch dieser Vaterknoten schon voll belegt ist. Dann wird auf dieser Stufe nochmals eine *Seitenspaltung* durchgeführt.

Löschen Diese Operation ist analog zum *Einfügen* realisierbar. Es wird also zunächst die Blattseite gesucht und dann – nachdem eine Exklusivsperre erworben wurde – der Eintrag von dieser Seite entfernt.

Nach der Löschung des Eintrags kann es zu einem Unterlauf kommen, wodurch eigentlich das Verschmelzen zweier benachbarter Seiten erforderlich würde. Die Verschmelzung, oder präziser gesagt, das Löschen eines Knotens aus dem Baum führt bei dieser Synchronisationsmethode zu Problemen, da dann eine Suchoperation möglicherweise auf einen nicht mehr vorhandenen Knoten gelenkt werden könnte. Dies passiert, wenn ein Suchvorgang von einer Löschoperation überholt wird und diese Löschoperation wegen eines Unterlaufs genau den Knoten entfernt, auf den die Suche als nächstes zugreifen will. Aus diesem Grunde wird in der Literatur vorgeschlagen, auf die Unterlaufbehandlung ganz zu verzichten – in der Regel wird ein Unterlauf ohnehin durch nachfolgende Einfügeoperationen wieder revidiert.

Wir wollen das verzahnte Zusammenspiel zweier Operationen auf dem B^+-Baum aus Abbildung 11.24 illustrieren:

- Suchen(15)
- Einfügen(14)

Wir nehmen an, dass die Suchoperation als erstes startet und die Wurzel und den linken Knoten der zweiten Stufe inspiziert. Als nächstes würde die *Suche* den zweiten Blattknoten von rechts besuchen. Jetzt nehmen wir aber an, dass zu diesem Zeitpunkt ein Kontextwechsel stattfindet, so dass *Einfügen*(14) ausgeführt wird. Im Zuge des Einfügevorgangs wird der zweite Blattknoten von rechts aufgespalten und der Zustand aus Abbildung 11.25 erzeugt. Wenn jetzt die Ausführung der Operation *Suchen*(15) wieder aufgenommen wird, befindet sich der Eintrag 15 nicht mehr auf der ursprünglich ermittelten Seite (2. von rechts). Deshalb muss die Suche auf den rechten Geschwisterknoten ausgedehnt werden.

[3] Man beachte, dass der Vaterknoten u.U. über die *rechts*-Verweise aufzufinden ist, falls der beim „Herunternavigieren" besuchte Vaterknoten selbst in der Zwischenzeit aufgespalten wurde.

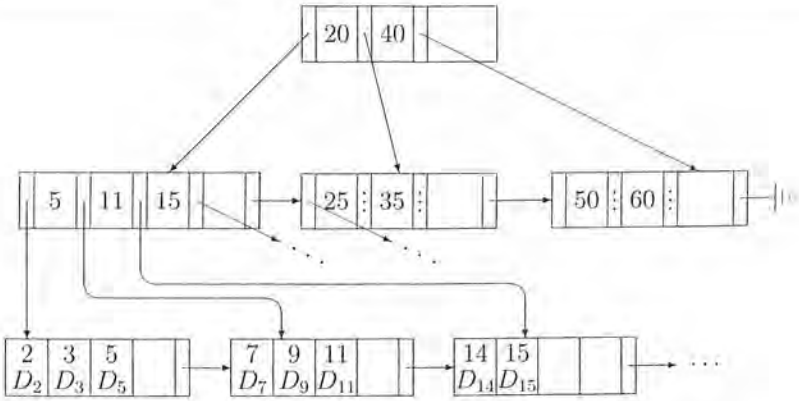


Abbildung 11.25: B$^+$-Baum mit *rechts*-Verweisen nach Einfügen von 14

11.15 Mehrbenutzersynchronisation in SQL-92

Bislang haben wir immer die *Serialisierbarkeit* als Korrektheitskriterium für die Parallelausführung von Transaktionen zugrunde gelegt. Die Erzwingung der Serialisierbarkeit schränkt natürlich die Parallelität ein. Deshalb haben die SQL-92-Entwerfer einige weitere weniger restriktive (aber teilweise obskure und potentiell die Datenbankkonsistenz gefährdende) Konsistenzstufen eingeführt. Diese Konsistenzstufen werden als „isolation level" bezeichnet, da sie die Isolationsstufe von parallel ausgeführten Transaktionen zueinander beschreiben. Der Transaktionsmodus wird in folgender Syntax beschrieben:

set transaction
[read only, | read write,]
[**isolation level**
read uncommitted, |
read committed, |
repeatable read, |
serializable,]
[**diagnostics size** ...,]

Der senkrechte Strich „|" gibt Alternativen an, die durch „[" und „]" eingegrenzten Teile sind optional. Bei Verwendung des **set transaction**-Befehls muss mindestens einer der optionalen Teile vorhanden sein; weiterhin muss bei einer Anweisung, die aus obiger Vorschrift hergeleitet wurde, das letzte Komma wegfallen. Unterstrichene Schlüsselwörter werden als default-Einstellungen bei fehlendem **set transaction** Befehl verwendet.

Zum Beispiel kann eine Transaktion auf Lesezugriffe durch **read only** beschränkt werden oder allgemein lesend und schreibend auf die Datenbasis zugreifen (**read write**). Die vier Konsistenzstufen sind wie folgt (auch im Standard sehr vage) definiert:

read uncommitted Dies ist die schwächste Konsistenzstufe. Sie darf auch nur für **read only**-Transaktionen spezifiziert werden. Eine derartige Transaktion hat

Zugriff auf noch nicht festgeschriebene Daten. Zum Beispiel ist folgender Schedule möglich:

T_1	T_2
	read(A)
	...
	write(A)
read(A)	
...	
	rollback

Hierbei liest die Transaktion T_1 einen Wert des Datums A, der von T_2 nie festgeschrieben wurde. Man kann sich leicht vorstellen, dass eine solche „**read uncommitted**"-Transaktion beliebig inkonsistente Datenbasis-Zustände zu sehen bekommt. Deshalb ist die Einschränkung solcher Transktionen auf reine Lesezugriffe (**read only**) äußerst notwendig.

Transaktionen dieser Art sind i.A. nur sinnvoll, um sich einen globalen Überblick über die Datenbasis zu verschaffen (*browsing*). Die „**read uncommitted**"-Transaktionen behindern die parallele Ausführung anderer Transaktionen nicht, da sie selbst keine Sperren benötigen.

read committed Diese Transaktionen lesen nur festgeschriebene Werte. Allerdings können sie unterschiedliche Zustände der Datenbasis-Objekte zu sehen bekommen:

T_1	T_2
read(A)	
	write(A)
	write(B)
	commit
read(B)	
read(A)	
...	

In diesem Fall liest die „**read committed**"-Transaktion T_1 zunächst den Wert von A, bevor T_2 A und B verändert. Danach liest T_1 den Wert von B – das reicht schon aus, um die Serialisierbarkeit zu verletzen. Warum? Noch schwerwiegender ist, dass T_1 jetzt nochmals A mit einem anderen Wert als zuvor liest. Man bezeichnet dieses Problem als „*non repeatable read*".

repeatable read Das oben aufgeführte Problem des *non repeatable read* wird durch diese Konsistenzstufe ausgeschlossen. Allerdings kann es hierbei noch zum Phantomproblem kommen. Dies kann z.B. dann passieren, wenn eine parallele Änderungstransaktion dazu führt, dass Tupel ein Selektionsprädikat erfüllen, das sie zuvor nicht erfüllten.

serializable Diese Konsistenzstufe fordert die Serialisierbarkeit. Es dürfte klar sein, dass die schwächeren (als **serializable**) Isolationsstufen u.U. zu sehr schwerwiegenden Konsistenzverletzungen führen können. Ein DBMS muss – gemäß dem

SQL92-Standard – nicht alle aufgeführten Isolationsstufen auch tatsächlich implementieren. Wenn eine benötigte Isolationsstufe nicht verfügbar ist, muss die jeweils „schärfere" vorhandene Form angewendet werden. Daraus folgt, dass mindestens die Serialisierbarkeit realisiert sein muss. Datenbankanwender sollten darauf achten, dass einige kommerzielle DBMS-Produkte eine andere, also schwächere, Konsistenzstufe als **serializable** als Default eingestellt haben.

11.16 Übungen

11.1 Entwerfen Sie Historien, die – bezogen auf die Abbildung 11.12 – in folgende Klassen fallen:

- $RC \cap SR$
- $ACA \cap SR$
- $ST \cap SR$

11.2 Diskutieren Sie die Vorteile *strikter* Historien hinsichtlich der Recovery anhand von Beispiel-Transaktionen. Warum sind nicht-strikte Historien – also z.B. solche aus $(SR \cap ACA) - ST$ – problematisch? Denken Sie an das lokale Rücksetzen von Transaktionen bei der Recovery-Behandlung.

11.3 Zeigen Sie, dass es serialisierbare Historien gibt, die ein Scheduler basierend auf dem Zwei-Phasen-Sperrprotokoll nicht zulassen würde. Anders ausgedrückt: Zeigen Sie, dass die Klasse SR größer ist als die Klasse $2PL$ (wobei $2PL$ die Klasse aller nach dem Zwei-Phasen-Sperrprotokoll generierbaren Historien darstellt).

11.4 Zeigen Sie, dass das (normale) Zwei-Phasen-Sperrprotokoll Historien aus $SR - RC$ zulässt. Mit anderen Worten, das 2PL-Verfahren würde nicht-rücksetzbare Historien zulassen.

11.5 Wäre es beim strengen 2PL-Protokoll ausreichend, alle Schreibsperren bis zum EOT zu halten, aber Lesesperren schon früher wieder abzutreten? Begründen Sie Ihre Antwort.

11.6 Wann genau können die Sperren gemäß dem strengen 2PL-Protokolls freigegeben werden? Denken Sie an die Recovery-Komponente.

11.7 Skizzieren Sie die Implementierung eines *Lock-Managers*, d.h. des Moduls, das Sperren verwaltet, Sperranforderungen entgegennimmt und diese ggf. gewährt oder die entsprechende Transaktion blockiert. Wie würden Sie die aktuell gewährten Sperren verwalten?

11.8 Weisen Sie (halbwegs) formal nach, dass das strenge 2PL-Protokoll nur strikte serialisierbare Historien zulässt.

11.9 Zur Erkennung von Verklemmungen wurde der Wartegraph eingeführt. Dabei wird eine Kante $T_i \rightarrow T_j$ eingefügt, wenn T_i auf die Freigabe einer Sperre durch T_j wartet. Kann es vorkommen, dass dieselbe Kante mehrmals eingefügt wird? Kann es vorkommen, dass gleichzeitig zwei Kanten $T_i \rightarrow T_j$ im Wartegraph existieren? Diskutieren Sie diese Aufgabe unter Annahme sowohl des normalen 2PL als auch des strengen 2PL-Protokolls.

11.10 Erläutern Sie den Zusammenhang zwischen dem Wartegraphen (zur Erkennung von Verklemmungen) und dem Serialisierbarkeitsgraphen (zur Feststellung, ob eine Historie serialisierbar ist).

11.11 Wie würden Sie die Zeitstempelmethode zur Vermeidung von Deadlocks anwenden, wenn eine Transaktion T_1 eine X-Sperre auf A anfordert, aber mehrere Transaktionen eine S-Sperre auf A besitzen? Diskutieren Sie die möglichen Fälle für **wound-wait** und **wait-die**.

11.12 Beweisen Sie, dass bei der Zeitstempelmethode garantiert keine Verklemmungen auftreten können. Hinweis: Verwenden Sie für Ihre Argumentation den Wartegraphen (der natürlich im System nicht aufgebaut wird, da er ja nicht benötigt wird).

11.13 Zeigen Sie Schedules, bei denen unnötigerweise Transaktionen nach der Zeitstempelmethode abgebrochen werden, obwohl eine Verklemmung nie aufgetreten wäre. Demonstrieren Sie dies für *wound-wait* und auch für *wait-die*.

11.14 Warum heißt die Strategie *wound-wait* und nicht *kill-wait*? Denken Sie daran, dass die „verwundete" Transaktion schon „so gut wie fertig" sein könnte. Beachten Sie das strenge 2PL-Protokoll und die Recovery-Komponente.

11.15 Beim „multiple-granularity locking" (MGL) werden Sperren von oben nach unten (top-down) in der Datenhierarchie erworben. Zeigen Sie mögliche Fehlerzustände, die eintreten könnten, wenn man die Sperren in umgekehrter Reihenfolge (also bottom-up) setzen würde.

11.16 Zeigen Sie an der Beispielhierarchie aus den Abbildungen 11.20 – 11.22 eine mögliche Verklemmungssituation des MGL-Sperrverfahrens.

11.17 Erweitern Sie das MGL-Sperrverfahren um einen weiteren Sperrmodus *SIX*. Dieser Sperrmodus sperrt den betreffenden Knoten im S-Modus (und damit implizit alle Unterknoten) und kennzeichnet die beabsichtigte Sperrung von einem (oder mehreren) Unterknoten im X-Modus.

- Erweitern Sie die Kompatibilitätsmatrix von Seite 353 um diesen Sperrmodus.
- Zeigen Sie an Beispielen das Zusammenspiel dieses Sperrmodus mit den anderen Modi.
- Skizzieren Sie mögliche Transaktionen, für die dieser Modus vorteilhaft ist.
- Wie verhält sich *SIX* mit den anderen Sperrmodi hinsichtlich eines (höchsten) Gruppenmodus?

11.18 Verifizieren Sie für das Zeitstempel-basierende Synchronisationsverfahren aus Abschnitt 11.10, dass

1. nur serialisierbare Schedules generierbar sind und
2. keine Verklemmungen auftreten können.

11.19 Thomas (1979) hat erkannt, dass man die Bedingung für Schreiboperationen bei der Zeitstempel-basierenden Synchronisation aus Abschnitt 11.10 abschwächen kann, um dadurch u.U. unnötiges Rücksetzen von Transaktionen zu verhindern. Die zweite Bedingung wird wie folgt modifiziert:

- T_i will A schreiben, also $w_i(A)$
  - Falls $TS(T_i) < readTS(A)$ gilt, setze T_i zurück (wie gehabt).
  - Falls $TS(T_i) < writeTS(A)$ gilt, ignoriere diese Operation von T_i einfach; aber fahre mit T_i fort.
  - Anderenfalls führe die Schreiboperation aus und setze $writeTS(A)$ auf $TS(T_i)$.

Verifizieren Sie, dass die generierbaren Schedules immer noch serialisierbar sind.

Zeigen Sie einen Beispiel-Schedule, der mit dieser Modifikation möglich ist, aber beim Originalverfahren abgewiesen würde.

Hinweis: Betrachten Sie sogenannte „blind writes", das sind Schreiboperationen auf ein Datum, denen in derselben Transaktion kein Lesen des betreffenden Datums vorausgegangen ist. Wenn man mehrere „blind writes" $w_1(A), w_2(A), \ldots, w_i(A)$ hat, die in der angegebenen Reihenfolge hätten ausgeführt werden müssen, so ist dies äquivalent zu der alleinigen Ausführung des letzten „blind writes", nämlich $w_i(A)$ und der Ignorierung der anderen Schreibvorgänge auf A. Tatsächlich muss nur der letzte Schreibvorgang – hier $w_i(A)$ – ein „blind write" sein; die anderen Schreibvorgänge könnten auch „normale" Schreiboperationen mit vorausgegangenem Lesen das Datums sein.

11.20 Finden Sie jeweils Anwendungsbeispiele für Anfragen, die ohne Gefährdung der Integrität der Datenbasis die Konsistenzstufen **read committed** und **repeatable read** benutzen können.

11.21 Beschreiben Sie jeweils für die Konsistenzstufen **read committed**, **repeatable read** und **serializable** die Sperrenvergabe beim MGL-Sperrverfahren für exact-match-queries und range-queries, wobei die Parallelität so wenig wie möglich eingeschränkt werden soll.

11.17 Literatur

Die Grundkonzepte der Serialisierbarkeit wurden im Rahmen der System R-Entwicklung am IBM Forschungslabor San Jose entwickelt. In dem dazugehörigen Artikel von Eswaran et al. (1976) wurde auch das Zweiphasen-Sperrprotokoll eingeführt.

Schlageter (1978) hat schon sehr früh einen Aufsatz zur Prozesssychronisation in Datenbanksystemen verfasst.

Ein sehr gutes (und umfassendes) Buch zur Mehrbenutzersynchronisation wurde von Bernstein, Hadzilacos und Goodman (1987) geschrieben. Papadimitriou (1986) beschreibt die Transaktionskonzepte noch etwas formaler.

Die verschiedenene Synchronisationslevel des SQL-92-Standards sind Gegenstand vieler Diskussionen. Berenson et al. (1995) versuchen die verschiedenen Level präziser zu charakterisieren, als dies im Standard (und natürlich auch in unserer sehr kurzen Abhandlung in Abschnitt 11.15) geschehen ist.

Die Zeitstempel-basierende Synchronisation ist von Reed (1983) entwickelt worden.

Das MGL-Sperrverfahren für unterschiedliche Sperrgranulate wurde erstmals von Gray, Lorie und Putzolu (1975) vorgestellt.

Es gibt viele Untersuchungen zur optimistischen Synchronisation: Härder (1984) analysierte verschiedene Varianten, Lausen (1983) hat eine Formalisierung entwickelt und Prädel, Schlageter und Unland (1986) haben Abwandlungen der Verfahren zur Steigerung der Leistungsfähigkeit vorgestellt. Trotzdem haben sich diese Synchronisationsverfahren in der Praxis (noch?) nicht durchsetzen können.

Peinl und Reuter (1983) haben versucht, die Leistungsfähigkeit der unterschiedlichen Synchronisationsverfahren empirisch zu bewerten.

Korth (1983) hat untersucht, inwieweit die Semantik der Operationen – also solcher Operationen, die über die primitiven Datenbankoperationen *read* und *write* hinausgehen – für die Synchronisation ausgenutzt werden können. Diese Arbeiten wurden im Zusammenhang mit abstrakten Datentypen von Schwarz und Spector (1984) und von Weihl und Liskov (1985) erweitert.

Weikum (1988) untersuchte mehrschichtige Transaktionskonzepte. Diese Arbeiten sind auch in Weikum (1991) zusammengefasst.

Alonso et al. (1994) stellten eine konzeptuelle Vereinheitlichung der Mehrbenutzer-Synchronisation und der Recovery vor.

Klahold et al. (1985) haben Transaktionskonzepte für Design-Anwendungen vorgeschlagen.

Synchronisationskonzepte für Suchbäume wurden u.a. von Bayer und Schkolnick (1977) vorgeschlagen. Das in diesem Kapitel behandelte Verfahren für die B^+-Bäume beruht auf Arbeiten von Lehman und Yao (1981) – die wiederum die Arbeiten zur Synchronisation von binären Suchbäumen von Kung und Lehman (1980) als Grundlage haben.

12. Sicherheitsaspekte

Bisher haben wir uns nur mit dem Schutz von Daten vor unabsichtlicher Beschädigung befasst (Kapitel 5 und Kapitel 9 bis 11). Im folgenden soll der Schutz gegen absichtliche Beschädigung und Enthüllung von sensiblen oder persönlichen Daten betrachtet werden. Die Schutzmechanismen sollen dazu in drei Kategorien unterteilt werden:

- **Identifikation und Authentisierung.** Bevor Benutzer Zugang zu einem Datenbanksystem erhalten, müssen sie sich in der Regel identifizieren. Eine Identifikation kann zum Beispiel durch Eingabe des Benutzernamens erfolgen. Die Authentisierung überprüft, ob es sich bei den Benutzern auch wirklich um diejenigen handelt, für die sie sich ausgeben. Üblicherweise werden hierzu Passwörter verwendet.

- **Autorisierung und Zugriffskontrolle.** Eine Autorisierung besteht aus einer Menge von Regeln, die die erlaubten Arten des Zugriffs auf *Sicherheitsobjekte* durch *Sicherheitssubjekte* festlegen. Ein Sicherheitsobjekt ist eine passive Entität, die Informationen beinhaltet, wie z.B. ein Tupel oder ein Attribut. Ein Sicherheitssubjekt ist eine aktive Entität, die einen Informationsfluss bewirkt. Sicherheitssubjekte können Benutzer oder Benutzergruppen sein, aber auch Datenbankprozesse bzw. Anwendungsprogramme.

- **Auditing.** Um die Richtigkeit und Vollständigkeit der Autorisierungsregeln zu verifizieren und Schäden rechtzeitig zu erkennen, kann über jede sicherheitsrelevante Datenbankoperation Buch geführt werden.

Wie die Autorisierungsregeln formuliert und durchgesetzt werden, hängt stark vom Schutzbedürfnis der Datenbankbetreiber ab. Gerade im Sicherheitsbereich gibt es eine Vielzahl von einsetzbaren *Strategien* (engl. policies). Als Veranschaulichung unterschiedlicher Schutzbedürfnisse sollen folgende Beispielfälle dienen:

- **Datenbank an einer Hochschule.** In der Datenbank werden Versuchsergebnisse gespeichert. Austausch von Informationen spielt eine wichtige Rolle, das Sicherheitsbedürfnis ist gering. Daher sind standardmäßig alle Daten lesbar, nur in Ausnahmesituationen ist ein größerer Schutz notwendig.

- **Datenbank in einem Betrieb.** Das Datenbanksystem ist eine zentrale Ressource und muss vor Ausfall geschützt werden. Die Leistung ist ein wichtiger Aspekt. Es existieren einige vertrauliche Daten. Schäden am Datenbestand bzw. Enthüllung von vertraulichen Daten lassen sich vielfach als finanzieller Verlust ausdrücken. In den meisten Fällen wird ein einfacher Schutzmechanismus verwendet, der Benutzergruppen Zugang zu Daten gewährt bzw. entzieht.