ANDREAS MEIER
MICHAEL KAUFMANN

SQL- & NoSQL-Datenbanken

8. AUFLAGE

eXamen.press

Springer Vieweg

eXamen.press

Weitere Informationen zu dieser Reihe finden Sie unter
http://www.springer.com/series/5520

eXamen.press ist eine Reihe, die Theorie und Praxis aus allen Bereichen der Informatik für die Hochschulausbildung vermittelt.

Andreas Meier • Michael Kaufmann

SQL- & NoSQL-Datenbanken

8., überarbeitete und erweiterte Auflage

Springer Vieweg

Andreas Meier
Institut für Informatik
Universität Fribourg
Fribourg, Schweiz

Michael Kaufmann
Departement für Informatik
Hochschule Luzern
Rotkreuz, Schweiz

ISSN 1614-5216
eXamen.press
ISBN 978-3-662-47663-5 ISBN 978-3-662-47664-2 (eBook)
DOI 10.1007/978-3-662-47664-2

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

Springer Vieweg

Lektorat: Melody Aimée Reymond

Grafikgestaltung: Thomas Riediker

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist Teil von Springer Nature
Die eingetragene Gesellschaft ist Springer-Verlag GmbH Berlin Heidelberg

Geleitwort

Der Begriff „Datenbank“ gehört längst zum Alltagswortschatz. Manager und Verwaltungsangestellte, aber auch Studierende fast aller Fachrichtungen verwenden ihn häufig. Sie verstehen darunter eine sinnvoll organisierte Sammlung von computergespeicherten Datensätzen, in denen gezielt Daten gesucht und eingesehen werden können. Wie und warum das möglich ist, überlassen sie gerne den Datenbankspezialisten.

Datenbanknutzer scheren sich meist wenig darum, welche immateriellen und wirtschaftlichen Werte in jeder einzelnen Datenbank stecken. Das gilt sowohl für das Ersatzteillagersystem eines Autoimporteurs, für die Computerlösung mit den Kundendepots einer Bank als auch für das Patienteninformationssystem eines Krankenhauses. Allerdings können ein Zusammenbruch solcher Systeme oder nur schon gehäufte Fehler das betroffene Unternehmen oder die involvierte Organisation existenziell bedrohen. In all diesen Betrieben lohnt es sich daher für einen Personenkreis, der weit über die „Datenbankspezialisten“ hinausreicht, genauer hinzusehen. Sie alle sollten verstehen, was Datenbanken effektiv leisten können und welche Rahmenbedingungen dafür geschaffen und unterhalten werden müssen.

Die wohl wichtigste Überlegung im Zusammenhang mit Datenbanken überhaupt betrifft einerseits die Unterscheidung zwischen den darin gespeicherten Daten (Benutzerdaten) und andererseits deren Verwaltung sowie das wirtschaftliche Gewicht dieser zwei Bereiche. Zur Verwaltung gehören vielfältige technische und administrative Einrichtungen und Dienste, namentlich Computer, Datenbanksysteme und zusätzliche Speicher, aber auch Fachleute für die Bereitstellung und den sicheren Betrieb all dieser Komponenten – eben die „Datenbankspezialisten“. Und nun das Wichtigste: Der Gesamtaufwand für die Datenverwaltung ist im betrieblichen Alltag einer Datenbank meist viel kleiner als der Aufwand für die Benutzerdaten selber; die Daumenregel besagt hier etwa ein Viertel.

Der Grossteil des Aufwands bei Datenbanken stammt von Beschaffung, Betreuung und Nutzung der Benutzerdaten. Dazu gehören namentlich die Personalkosten all jener Mitarbeiterinnen und Mitarbeiter, welche Daten in die Datenbank eingeben, bereinigen, und daraus Antworten abrufen und Unterlagen erstellen. In den oben genannten Beispielen

sind dies Lagermitarbeitende, Bankangestellte und Spitalpersonal in verschiedensten Funktionen – und das meist über viele Jahre.

Nur wer diese übergeordnete Aufwandverteilung zwischen Datenpflege und -nutzung einerseits und Datenbankbetreuung anderseits verstanden und verinnerlicht hat, wird die Bedeutung der damit verbundenen Arbeiten angemessen einschätzen können. Die Datenbankbetreuung beginnt mit dem Datenbankentwurf, der bereits eine Vielzahl von Fachthemen mit einschliesst, etwa die Festlegung der Konsistenzprüfungen bei Datenmanipulationen sowie die Regelung von Datenredundanzen, welche auf logischer Ebene unerwünscht, auf Speicherebene absolut unverzichtbar sind. Immer ist die Entwicklung von Datenbanklösungen auf die spätere Nutzung ausgerichtet. Ungeschickte Entwicklungsentscheide belasten den späteren Betrieb auf Dauer. Es braucht daher einige Erfahrung, um bei der Festlegung von Konsistenzbedingungen den guten Mittelweg etwa zwischen allzu strengen und allzu lockeren Regeln zu finden. Allzu strenge Bedingungen behindern den Betrieb, zu lockere tolerieren Datenfehler und führen immer wieder zu teuren nachträglichen Datenreparaturen.

Es lohnt sich somit für alle, welche mit Datenbankentwicklung und -betrieb auf Führungsebene oder als Datenbank-Fachleute zu tun haben, diesen Informatikbereich systematisch kennen zu lernen. Das Inhaltsverzeichnis dieses Buchs zeigt die Breite der Themen. Bereits sein Titel weist darauf hin, dass das Buch nicht nur die klassische Datenbankwelt (Relationenmodell, SQL) in ihrer ganzen Tiefe darstellt, sondern zusätzlich fundierte Ausblicke auf aktuelle Weiterentwicklungen und Nachbargebiete vermittelt. Stichworte dazu sind namentlich „NoSQL/Postrelational“ und „Big Data“. Ich wünsche dem Buch auch in seiner neuesten Auflage eine gute Aufnahme bei Studierenden und Praktikern; dessen Verfasser kennen beide Seiten bestens.

Carl August Zehnder

Vorwort zur achten Auflage

Als ich 2002 an der Vorlesung „Software Engineering III: Datenbanken“ von Prof. Dr. Andreas Meier an der Universität Fribourg teilnahm, kam ich zum ersten Mal in Kontakt mit der vierten Auflage dieses Buchs, damals mit dem Titel „Relationale Datenbanken: Leitfaden für die Praxis“. Ich fand die Lektüre erkenntnisreich und erfrischend anders. Besonders gefiel mir die Art, wie das Buch abstrakte Konzepte auf verständliche Weise vermittelt. Die Inhalte waren mir als Student im dritten Studienjahr im Vergleich zu anderen, eher theoriegeleiteten Abhandlungen sehr zugänglich.

Heute, vierzehn Jahre später, bin ich selbst Dozent für Datenbanken. Ich kenne Andreas Meier nun schon seit zwolf Jahren persönlich. Er hat meine Masterarbeit 2004–2005 und meine Dissertation 2008–2012 betreut, und er unterstützt meine Forschung und Lehre bis heute. Andreas ist mein Lehrer und Mentor, und es freut mich, dass ich dazu beitragen darf, die achte Auflage seines Buchs zu gestalten.

Aktuell verwende ich im Herbstsemester 2015 die siebte Auflage, um meinen Studierenden an der Hochschule Luzern – Technik & Architektur das Thema Datenmanagement zu vermitteln. Ich stütze mich dabei auf meine Erfahrung in der praktischen Arbeit mit Datenbanken bei Finanzinstituten, Versicherungen und Informatikdienstleistern seit 2005, aber auch auf die Theorie der relationalen Datenbanksysteme, welche bereits in den 1970er-Jahren von Codd (Relationenmodell und Normalformen), Chen (Entitäten- und Beziehungsmodell) und Chamberlin & Boyce (SEQUEL) in die Wege geleitet wurde und noch heute die Praxis des Datenmanagements wesentlich prägt.

Es ist bemerkenswert, wie stabil bestehende fachliche Inhalte im Umfeld der Datenbanken sind. Die Informatik ist allgemein dafür bekannt, einer rasanten Entwicklung unterworfen zu sein, welche neue Technologien in unglaublichem Tempo hervorbringt und dadurch die Älteren rasch überflüssig macht. Meiner Erfahrung nach ist das aber nur oberflächlich der Fall. Viele Aspekte der Informatik verändern sich im Kern nicht wesentlich. Dazu gehören nicht nur die Grundlagen wie beispielsweise die Funktionsprinzipen von universellen Rechenmaschinen, Prozessoren, Compilern oder Betriebssystemen, sondern in die Jahre gekommene Technologien wie beispielsweise COBOL, TCP/IP, aber auch die Sprache SQL und die relationalen Datenbanken mit ihren

Methoden und Architekturen. Diese Technologien sind seit Jahrzenten im Einsatz und werden das auch in absehbarer Zeit bleiben.

Die vorliegende achte Auflage dieses Buchs bedeutet ein fünfundzwanzigjähriges Jubiläum, denn das Vorwort zur ersten Auflage wurde im Jahr 1991 verfasst (siehe Nachdruck auf S. XI). Die ursprüngliche Version war eine Broschüre von 175 Seiten, welche aus Vorlesungsnotizen entstanden ist. Die Grundstruktur der ersten Auflage schimmert bis heute durch: Die Grundbegriffe der Datenbanktheorie, die Datenmodellierung, Abfrage- und Manipulationssprachen und die Systemarchitektur bilden noch heute wichtige Grundlagen des Datenmanagements. Gleichwohl ist fachlich einiges hinzugekommen. Im Lauf der Jahre und in sieben weiteren Auflagen wurden die Inhalte ergänzt und auf den neusten Stand gebracht. Gerade die Entwicklungen im Umfeld von Big Data brachten neue Technologien in die Welt der Datenbanken, welchen wir in dieser achten Auflage so viel Beachtung schenken, dass ein neuer Titel des Buchs daraus resultiert.

Die nichtrelationalen Datenbanktechnologien, welche unter dem Sammelbegriff NoSQL immer mehr Einsatzgebiete finden, unterscheiden sich nicht nur oberflächlich von den klassischen relationalen Datenbanken, sondern auch in den zugrunde liegenden Prinzipien. Während die relationalen Datenbanken im zwanzigsten Jahrhundert mit dem Zweck von hierarchischen, straff organisierten, betrieblichen Formen des Datenmanagements entwickelt wurden, welche dadurch zwar Stabilität boten, aber Flexibilität einschränkten, zielen die NoSQL-Datenbanken auf die Freiheit, Struktur rasch verändern zu können. Das hat weitreichende Konsequenzen und führt zu einem neuen Ansatz im Datenmanagement, welcher von den bisherigen Theorien zum Grundbegriff der Datenbank deutlich abweicht: Zur Art und Weise, wie Daten modelliert werden, wie Daten abgefragt und manipuliert werden, bis hin zur Systemarchitektur. Sie stellt diese aber nicht grundsätzlich in Frage, sondern ergänzt sie lediglich. Aus diesem Grund haben wir in allen Kapiteln diese beiden Welten, die SQL- und NoSQL-Datenbanken, aus verschiedenen Perspektiven miteinander verglichen.

Andreas hat mich im Herbst 2014 angefragt, dieses vorliegende Werk für die achte Auflage um den Teil NoSQL-Datenbanken zu erweitern und bei der Weiterentwicklung des bestehenden Inhalts mitzuwirken. Seither haben wir Inhalte hinzugefügt und Bestehendes angepasst. Zudem konnten wir eine Website unter dem Namen *sql-nosql.org* lancieren, auf der wir Lehr- und Übungsmaterialien wie Folien, Tutorien, Fallstudien, ein Repetitorium sowie eine Workbench für MySQL sowie Neo4j freigeben, damit Sprachübungen mit SQL resp. Cypher durchgeführt werden können. An dieser Stelle danken wir den beiden Assistenten Alexander Denzler sowie Marcel Wehrle für die Entwicklung der Workbench für relationale und graphorientierte Datenbanken. Für die Neugestaltung der Grafiken konnten wir im Februar 2015 Thomas Riediker gewinnen. Wir danken ihm für seinen unermüdlichen Einsatz. Es ist ihm gelungen, den Abbildungen einen modernen Stil und eine individuelle Note zu verleihen. Für die Weiterentwicklung der Tutorien und Fallbeispiele, die auf der Website *sql-nosql.org* abrufbar sind, danken wir den Informatikstudierenden Andreas Waldis, Bettina Willi, Markus Ineichen und Simon Studer für ihre Beiträge zum Tutoriat in Cypher respektive zur Fallstudie Travelblitz mit OpenOffice

Base und mit Neo4J. Für das Feedback zum Manuskriptentwurf danke ich dem Doktoranden Alexander Denzler der Universität Fribourg, meinem ehemaligen Kommilitonen Dr. Daniel Fasel, dem Datenbankkollegen beim Institut für Wirtschaftsinformatik der Hochschule Luzern, Prof. Konrad Marfurt, und besonders meinem Vorgänger Prof. Dr. Thomas Olnhoff, für ihre Bereitschaft, zur Qualität unseres Werks mit ihren Hinweisen beizutragen. Ein Dankeschön geht an Dorothea Glaunsinger und Hermann Engesser vom Springer-Verlag, die uns mit Geduld und Sachverstand unterstützt haben. Das Geleitwort verdanken wir Carl August Zehnder, emeritierter Professor für Datenbanken der ETH Zürich. Ganz herzlich danke ich meiner Frau Melody Reymond für das Lektorat unseres Texts. Es liegt ihr persönlich am Herzen, dass unser Buch, und alle Versionen, die folgen mögen, nicht nur fachlich, sondern auch sprachlich überzeugen.

Horw, im Januar 2016 Michael Kaufmann

Vorwort der ersten Auflage

Viele Unternehmen der Wirtschaft und der öffentlichen Hand haben die Zeichen der Zeit erkannt und stellen ihre Informationssysteme auf ein neues Fundament. Sie legen ein unternehmensweites Datenmodell fest, bauen relationale Datenbanken auf und setzen entsprechende Entwicklungs- und Auswertungswerkzeuge ein.

Die vorliegende Fachbroschüre zeigt das gesamte Anwendungsspektrum vom relationalen Datenbankentwurf bis hin zur Entwicklung von postrelationalen Datenbankmanagementsystemen im Überblick und praxisnah auf. Als *fundierte Einführung in das Gebiet der relationalen Datenbankmanagementsysteme* ermöglicht sie den interessierten Datenverarbeitungs-Praktikern,

- bei der Datenmodellierung und in der Datenbankentwurfsmethodik das Relationenmodell anzuwenden,
- relationale Abfrage- und Manipulationssprachen kennenzulernen und einzusetzen,
- die automatischen Abläufe und Techniken innerhalb eines relationalen Datenbanksystems zu verstehen,
- die Stärken und Schwächen relationaler Technologie zu erkennen, in seine Überlegungen miteinzubeziehen und künftige Entwicklungen einigermassen richtig abzuschätzen.

Die Broschüre richtet sich ausser an *DV-Praktiker* auch an *Ausbildungsverantwortliche* von Unternehmen, *Dozierende, Schülerinnen und Schüler sowie Studentinnen und Studenten* an Fachhochschulen und Berufsakademien sowie an alle, die eine *praxisbezogene Einführung* in die relationale Datenbanktechnik suchen. Im Zentrum stehen wichtige Begriffe, die zwar vielerorts gebraucht, jedoch selten richtig verstanden und angewendet werden. Illustriert wird die Broschüre durch eine grosse Anzahl von einfachen, meist selbsterklärenden Abbildungen. Ein gezielt zusammengestelltes Literaturverzeichnis verweist den interessierten Leser auf weiterführende Publikationen, die einen vertieften Einblick in die angeschnittenen Themenkreise bieten.

Diese Einführung geht die relationale Datenbanktechnologie von verschiedenen Blickwinkeln an. Sie erläutert Entwurfsmethoden und Sprachaspekte ebenso wie wichtige Architekturkonzepte relationaler Datenbanksysteme. Auf eine Beschreibung spezifischer Herstellerprodukte wird verzichtet, um den grundlegenden Methoden und Techniken mehr Platz einzuräumen und das Verständnis beim Benützen relationaler Datenbanken zu fördern. Somit schliesst diese Veröffentlichung eine Lücke in der praxisbezogenen Fachliteratur.

Die Fachbroschüre ist im Rahmen eines firmeninternen Ausbildungsprogrammes entstanden, ergänzt durch Diskussionsbeiträge aus der eigenen Vorlesungsrunde „Praxis relationaler Datenbanken“ an der ETH in Zürich. Viele Fachkollegen und -kolleginnen aus Praxis und Hochschule haben dazu beigetragen, den Text verständlicher und die Abbildungen anschaulicher zu gestalten. Mein Dank richtet sich an Eirik Danielsen, Bernardin Denzel, Emmerich Fuchs, Caroline Grässle-Mutter, Michael Hofmann, Günther Jakobitsch, Hans-Peter Joos, Klaus Küspert, Thomas Myrach, Michel Patcas, Ernst-Rudolf Patzke, Thomas Rätz, Werner Schaad, August Scherrer, Walter Schnider, Max Vetter und Gerhard Weikum. Urs Bebler danke ich für die spontane Bereitschaft, die ursprünglichen Kursunterlagen in Buchform herauszugeben. Ein besonderes Kompliment richte ich an Peter Gasche als kritischen Gutachter, der das Skriptum sprachlich sorgfältig bearbeitet hat. Dem Springer-Verlag, vor allem Hans Wössner, danke ich für viele praktische Hinweise.

Liestal, im Oktober 1991 Andreas Meier

Inhaltsverzeichnis

Abbildungsverzeichnis

Datenmanagement

1

1.1 Informationssysteme und Datenbanken

Der Wandel von der Industrie- zur Informations- und Wissensgesellschaft spiegelt sich in der Bewertung der Information als Produktionsfaktor. *Information* (engl. *information*) hat im Gegensatz zu materiellen Wirtschaftsgütern folgende Eigenschaften:

- **Darstellung**: Information wird durch Daten (Zeichen, Signale, Nachrichten oder Sprachelemente) spezifiziert.
- **Verarbeitung**: Information kann mit Hilfe von Algorithmen (Berechnungsvorschriften) und Datenstrukturen übermittelt, gespeichert, klassifiziert, aufgefunden und in andere Darstellungsformen transformiert werden.
- **Kombination**: Information ist beliebig kombinierbar. Die Herkunft einzelner Teile ist nicht nachweisbar. Manipulationen sind jederzeit möglich.
- **Alter**: Information unterliegt keinem physikalischen Alterungsprozess.
- **Original**: Information ist beliebig kopierbar und kennt keine Originale.
- **Vagheit**: Information ist unscharf, d. h. sie ist oft unpräzis und hat unterschiedliche Aussagekraft (Qualität).
- **Träger**: Information benötigt keinen fixierten Träger, d. h. sie ist unabhängig vom Ort.

Diese Eigenschaften belegen, dass sich digitale Güter (Informationen, Software, Multimedia etc.), sprich Daten, in der Handhabung sowie in der ökonomischen und rechtlichen Wertung von materiellen Gütern stark unterscheiden. Beispielsweise verlieren physische Produkte durch die Nutzung meistens an Wert, gegenseitige Nutzung von Informationen hingegen kann einem Wertzuwachs dienen. Ein weiterer Unterschied besteht darin, dass materielle Güter mit mehr oder weniger hohen Kosten hergestellt werden, die Vervielfältigung von Informationen jedoch einfach und kostengünstig ist (Rechen-

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_1

aufwand, Material des Informationsträgers). Dies wiederum führt dazu, dass die Eigentumsrechte und Besitzverhältnisse schwer zu bestimmen sind, obwohl man digitale Wasserzeichen und andere Datenschutz- und Sicherheitsmechanismen zur Verfügung hat.

Fasst man die *Daten als Grundlage von Information als Produktionsfaktor* im Unternehmen auf, so hat das wichtige Konsequenzen:

- **Entscheidungsgrundlage**: Daten bilden Entscheidungsgrundlagen und sind somit in allen Organisationsfunktionen von Bedeutung.
- **Qualitätsanspruch**: Daten können aus unterschiedlichen Quellen zugänglich gemacht werden; die Qualität der Information ist von der Verfügbarkeit, Korrektheit und Vollständigkeit der Daten abhängig.
- **Investitionsbedarf**: Durch das Sammeln, Speichern und Verarbeiten von Daten fallen Aufwand und Kosten an.
- **Integrationsgrad**: Aufgabengebiete und -träger jeder Organisation sind durch Informationsbeziehungen miteinander verknüpft, die Erfüllung ist damit in hohem Maße vom Integrationsgrad der Daten abhängig.

Ist man bereit, die Daten als Produktionsfaktor zu betrachten, muss diese Ressource geplant, gesteuert, überwacht und kontrolliert werden. Damit ergibt sich die Notwendigkeit, das Datenmanagement auch als Führungsaufgabe wahrzunehmen. Dies bedeutet einen grundlegenden Wechsel im Unternehmen: Neben einer technisch orientierten Funktion wie Betrieb der Informations- und Kommunikationsinfrastruktur (Produktion) muss die Planung und Gestaltung der Datenflüsse (Anwendungsportfolio) ebenso wahrgenommen werden.

Ein *Informationssystem* (engl. *information system*) erlaubt den Anwendenden gemäß Abb. 1.1, interaktiv Informationen zu speichern und zu verknüpfen, Fragen zu stellen und Antworten zu erhalten. Je nach Art des Informationssystems sind hier Fragen zu einem begrenzten Anwendungsbereich zulässig. Darüber hinaus existieren offene Informationssysteme und Webplattformen im World Wide Web, die beliebige Anfragen mit der Hilfe einer Suchmaschine bearbeiten.

In Abb. 1.1 ist das rechnergestützte Informationssystem mit einem Kommunikationsnetz resp. mit dem Internet verbunden, um neben unternehmensspezifischen Auswertungen webbasierte Recherchearbeiten sowie Informationsaustausch weltweit zu ermöglichen. Jedes umfangreichere Informationssystem nutzt Datenbanktechnologien, um die Verwaltung und Auswertung der Daten nicht jedes Mal von Grund auf neu entwickeln zu müssen.

Ein *Datenbanksystem* (engl. *database management system*) ist eine Software zur applikationsunabhängigen Beschreibung, Speicherung und Abfrage von Daten. Jedes Datenbanksystem besteht aus einer Speicherungs- und einer Verwaltungskomponente. Die Speicherungskomponente umfasst alle Daten, welche in einer organisierten Form abgespeichert werden sowie deren Beschreibung. Die Verwaltungskomponente enthält eine Abfrage- und Manipulationssprache, um die Daten und Informationen auswerten und

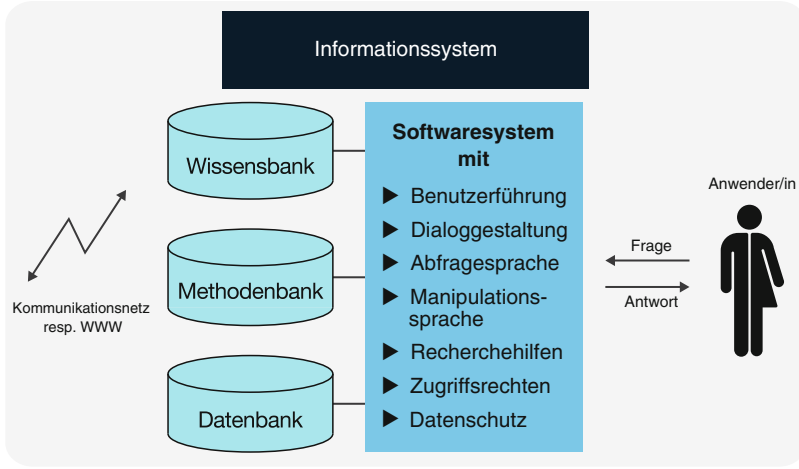

Abb. 1.1 Architektur und Komponenten eines Informationssystems

verändern zu können. Die Verwaltungskomponente bedient nicht nur die Benutzerschnittstelle, sondern verwaltet auch Zugriffs- und Bearbeitungsrechte der Anwendenden.

In der Praxis sind häufig SQL-Datenbanken (SQL steht für Structured Query Language, siehe Abschn. 1.2) im Einsatz. Eine besondere Herausforderung stellt sich, wenn webbasierte Dienstleistungen mit heterogenen Datenbeständen in Echtzeit bewältigt werden müssen (Abschn. 1.3 über Big Data). Aus diesem Grund kommen vermehrt NoSQL-Ansätze zur Anwendung (siehe Abschn. 1.4). Für den Einsatz relationaler wie nicht-relationaler Technologien müssen die Vor- und Nachteile gegeneinander abgewogen werden; ev. drängt sich die Kombination unterschiedlicher Technologien für ein Anwendungsfeld auf (vgl. Betrieb eines Webshops in Abschn. 5.6). Abhängig von der gewählten Datenbankarchitektur muss das Datenmanagement im Unternehmen etabliert und mit geeigneten Fachkräften entwickelt werden (Abschn. 1.5). Weiterführende Literaturangaben sind in Abschn. 1.6 gegeben.

1.2 SQL-Datenbanken

1.2.1 Relationenmodell

Eine einfache und anschauliche Form Daten oder Informationen zu sammeln oder darzustellen, ist die der Tabelle. Von jeher sind wir es gewohnt, tabellarische Datensammlungen ohne Interpretationshilfen zu lesen und zu verstehen.

Möchten wir Informationen über Mitarbeitende sammeln, so können wir ein Tabellengerüst gemäß Abb. 1.2 entwerfen. Der in Großbuchstaben geschriebene Tabellenname MITARBEITER bezeichnet die Tabelle selbst. Die einzelnen Tabellenspalten werden mit

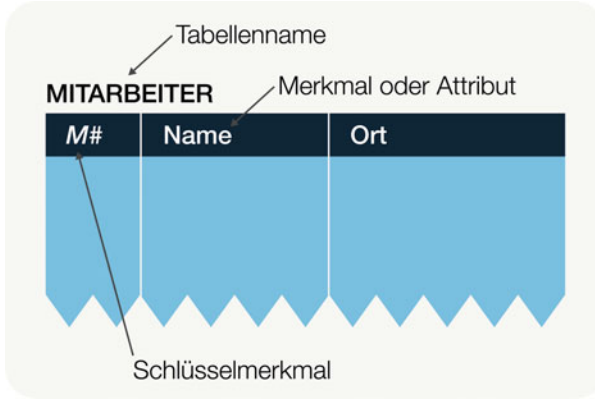

Abb. 1.2 Tabellengerüst für eine Tabelle MITARBEITER

den gewünschten Merkmals- oder Attributsnamen überschrieben; in unserem Beispiel sind dies die Mitarbeiternummer «M#», der Name des Mitarbeitenden «Name» und dessen Wohnort «Ort».

Ein *Merkmal* oder *Attribut* (engl. *attribute*) ordnet jedem Eintrag in der Tabelle einen bestimmten Datenwert aus einem vordefinierten *Wertebereich* (engl. *domain*) als Eigenschaft zu. In der Tabelle MITARBEITER ermöglicht das Merkmal M# das eindeutige Identifizieren der Mitarbeitenden. Aufgrund dieser Eigenschaft erklären wir die Mitarbeiternummer zum Schlüssel. Zur Verdeutlichung der Schlüsseleigenschaft werden die Schlüsselmerkmale im Folgenden kursiv im Tabellenkopf[1] angeschrieben. Mit dem Merkmal Ort werden die dazugehörenden Ortsnamen, mit dem Merkmal Name die entsprechenden Namen der Mitarbeitenden bezeichnet.

Ohne Weiteres lassen sich nun die gewünschten Daten der Mitarbeiter in die Tabelle MITARBEITER zeilenweise eintragen (vgl. Abb. 1.3). Dabei können einzelne Datenwerte mehrfach in den Spalten erscheinen. So bemerken wir in der Tabelle MITARBEITER, dass der Wohnort Liestal zweimal vorkommt. Dieser Sachverhalt ist wesentlich und sagt aus, dass sowohl der Mitarbeiter Becker als auch Mitarbeiterin Meier in Liestal wohnen. In der Tabelle MITARBEITER können nicht nur Ortsbezeichnungen mehrfach vorkommen, sondern auch die Namen der Mitarbeitenden. Aus diesen Gründen ist das bereits erwähnte Schlüsselmerkmal M# notwendig, das jeden Mitarbeitenden in der Tabelle eindeutig bestimmt.

Identifikationsschlüssel

Ein *Identifikationsschlüssel* (engl. *identification key*) oder einfach *Schlüssel* einer Tabelle ist ein Merkmal oder eine minimale Merkmalskombination, wobei innerhalb der Tabelle die Schlüsselwerte die Datensätze (genannt *Zeilen* oder *Tupel*) eindeutig identifizieren.

[1] In einigen Standardwerken der Datenbankliteratur werden die Schlüsselmerkmale durch Unterstreichung kenntlich gemacht.

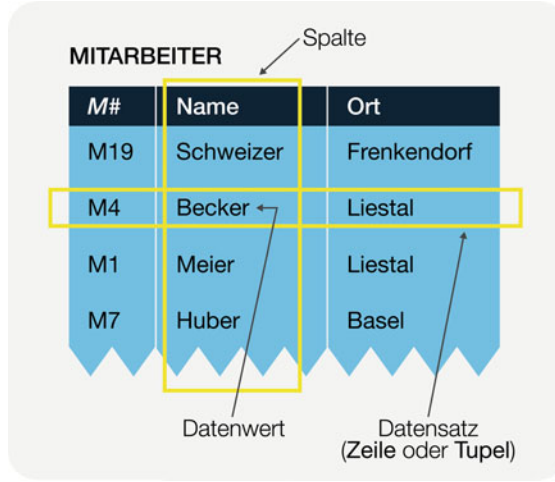

Abb. 1.3 Tabelle MITARBEITER mit Ausprägungen

Aus dieser Kurzdefinition lassen sich zwei wichtige *Schlüsseleigenschaften* herleiten:

- **Eindeutigkeit**: Jeder Schlüsselwert identifiziert eindeutig einen Datensatz innerhalb der Tabelle, d. h. verschiedene Tupel dürfen keine identischen Schlüssel aufweisen.
- **Minimalität**: Falls der Schlüssel eine Kombination von Merkmalen darstellt, muss diese minimal sein. Mit anderen Worten: Kein Merkmal der Kombination kann gestrichen werden, ohne dass die Eindeutigkeit der Identifikation verlorengeht.

Mit den beiden Forderungen nach Eindeutigkeit und Minimalität ist ein Schlüssel vollständig charakterisiert.

Anstelle eines natürlichen Merkmals oder einer natürlichen Merkmalskombination kann ein Schlüssel als künstliches Merkmal eingeführt werden. Die Mitarbeiternummer M# ist künstlich, weil sie keine natürliche Eigenschaft der Mitarbeitenden darstellt.

Aus ideellen Gründen sträuben wir uns dagegen, *künstliche Schlüssel* oder «Nummern» als identifizierende Merkmale vorzusehen, vor allem wenn es sich um personenbezogene Informationen handelt. Auf der anderen Seite führen natürliche Schlüssel nicht selten zu Eindeutigkeits- und Datenschutzproblemen: Falls ein Schlüssel beispielsweise aus Teilen des Namens und des Geburtstags zusammengesetzt wird, so ist die Eindeutigkeit nicht immer gewährleistet. Kommt hinzu, dass „sprechende“ Schlüssel etwas über die betroffene Person aussagen und damit die Privatsphäre tangieren.

Ein künstlicher Schlüssel sollte aufgrund obiger Überlegungen *anwendungsneutral und ohne Semantik* (Aussagekraft, Bedeutung) definiert werden. Sobald aus den Datenwerten eines Schlüssels irgendwelche Sachverhalte abgeleitet werden können, besteht ein Interpretationsspielraum. Zudem kann es vorkommen, dass sich die ursprünglich wohldefinierte Bedeutung eines Schlüsselwertes im Laufe der Zeit ändert oder verloren geht.

Tabellendefinition
Zusammenfassend verstehen wir unter einer *Tabelle* oder *Relation* (engl. *table, relation*) eine Menge von Tupeln, die tabellenförmig dargestellt werden und folgende Anforderungen erfüllen:

- **Tabellenname**: Eine Tabelle besitzt einen eindeutigen Tabellennamen.
- **Merkmalsname**: Innerhalb der Tabelle ist jeder Merkmalsname eindeutig und bezeichnet eine bestimmte Spalte mit der gewünschten Eigenschaft.
- **Keine Spaltenordnung**: Die Anzahl der Merkmale ist beliebig, die Ordnung der Spalten innerhalb der Tabelle ist bedeutungslos.
- **Keine Zeilenordnung**: Die Anzahl der Tupel einer Tabelle ist beliebig, die Ordnung der Tupel innerhalb der Tabelle ist bedeutungslos.
- **Identifikationsschlüssel**: Eines der Merkmale oder eine Merkmalskombination identifiziert eindeutig die Tupel innerhalb der Tabelle und wird als Identifikationsschlüssel deklariert.

Beim *Relationenmodell* (engl. *relational model*) wird gemäß obiger Definition jede Tabelle als Menge ungeordneter Tupel aufgefasst.

Relationenmodell
Das Relationenmodell drückt Daten wie Beziehungen zwischen den Daten tabellarisch aus. Mathematisch ist eine Relation R nichts anderes als eine *Teilmenge aus dem kartesischen Produkt von Wertebereichen*: $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ mit D_i als Wertebereich (engl. *domain*) des i-ten Attributs resp. Merkmals.

Ein Tupel r selber ist demnach eine Menge konkreter Datenwerte resp. Ausprägungen, $r = (d_1, d_2, \ldots, d_n)$. Zu beachten ist, dass aufgrund dieses Mengenbegriffs in einer Tabelle ein und dasselbe Tupel nur einmal vorkommen darf, d. h. $R = \{r_1, r_2, \ldots, r_m\}$.

Das Relationenmodell wurde Anfang der Siebzigerjahre durch die Arbeiten von Edgar Frank Codd begründet. Darauf aufbauend, entstanden in Forschungslabors erste *relationale Datenbanksysteme*, die SQL oder ähnliche Datenbanksprachen unterstützten. Inzwischen haben ausgereiftere Produkte die Praxis erobert.

1.2.2 Strukturierte Abfragesprache SQL

Wie erwähnt, stellt das Relationenmodell Informationen in Form von Tabellen dar. Dabei entspricht jede Tabelle einer Menge von Tupeln oder Datensätzen desselben Typs. Dieses Mengenkonzept erlaubt grundsätzlich, *Abfrage- und Manipulationsmöglichkeiten mengenorientiert* anzubieten.

Zum Beispiel ist das Resultat einer Selektionsoperation eine Menge, d. h. jedes Ergebnis eines Suchvorgangs wird vom Datenbanksystem als Tabelle zurückgegeben. Falls

Abb. 1.4 Formulierung einer Abfrage mit SQL

keine Tupel der durchsuchten Tabelle die geforderten Eigenschaften erfüllen, erhalten die Anwendenden eine leere Resultattabelle. Änderungsoperationen sind ebenfalls mengenorientiert und wirken auf eine Tabelle oder auf einzelne Tabellenbereiche.

Die wichtigste Abfrage- und Manipulationssprache für Tabellen heißt *Structured Query Language* oder abgekürzt SQL (vgl. Abb. 1.4). Diese Sprache wurde durch das ANSI (American National Standards Institute) und durch die ISO (International Organization for Standardization) genormt.[2]

Die Sprache SQL ist deskriptiv, denn die Ausdrücke beschreiben das gewünschte Resultat, und nicht die dafür erforderlichen Rechenschritte. Ein SQL-Ausdruck genügt einem allgemeinen Grundmuster, das wir anhand der in Abb. 1.4 aufgeführten Abfrage illustrieren:

«Selektiere (SELECT) das Merkmal Name aus (FROM) der Tabelle MITARBEITER, wobei (WHERE) der Wohnort Liestal ist!»

Der Ausdruck SELECT-FROM-WHERE wirkt auf eine oder mehrere Tabellen und erzeugt als Resultat immer eine Tabelle. Auf unser Beispiel bezogen erhalten wir für obige Abfrage eine Resultattabelle mit den gewünschten Namen Becker und Meier.

[2] Das ANSI ist der nationale Normenausschuss der USA und entspricht dem DIN (Deutsches Institut für Normierung) in Deutschland. Der ISO gehören die nationalen Normenausschüsse an.

In der mengenorientierten Arbeitsweise liegt ein wesentlicher Vorteil für die Anwendenden, da eine einzige SQL-Abfrage eine ganze Reihe von Aktionen im Datenbanksystem auslösen kann. So ist es nicht notwendig, dass die Anwendenden die Suchvorgänge selbst „ausprogrammieren“.

Relationale Abfrage- und Manipulationssprachen sind deskriptiv, also beschreibend. Allein durch das Festlegen der gesuchten Eigenschaften im Selektionsprädikat erhalten die Anwendenden die gewünschten Informationen. Eine Anleitung zur Berechnung der resultierenden Datensätze muss von ihnen nicht vorgenommen werden. Das Datenbanksystem übernimmt diese Aufgabe, bearbeitet die Abfrage oder die Manipulation mit eigenen Such- und Zugriffsmethoden und erstellt die gewünschte Resultattabelle.

Im Gegensatz zu den deskriptiven Abfrage- und Manipulationssprachen müssen die Abläufe zur Bereitstellung der gesuchten Information bei den herkömmlichen *prozeduralen Datenbanksprachen* durch die Anwendenden selbst ausprogrammiert werden. Dabei ist das Ergebnis jeder Abfrageoperation ein einzelner Datensatz und nicht eine Menge von Tupeln.

Bei der deskriptiven Formulierung einer Abfrage beschränkt sich SQL auf die Angabe der gewünschten Selektionsbedingung in der WHERE-Klausel, bei den prozeduralen Sprachen hingegen muss ein Algorithmus zum Auffinden der einzelnen Datensätze von den Anwendenden spezifiziert werden. Betrachten wir als Beispiel eine Abfragesprache für hierarchische Datenbanken, so suchen wir gemäß Abb. 1.5 zuerst mit GET_FIRST einen ersten Datensatz, der das gewünschte Suchkriterium erfüllt. Anschließend lesen wir sämtliche Datensätze durch GET_NEXT-Befehle, bis wir das Ende der Datei oder eine nächste Hierarchiestufe innerhalb der Datenbank erreichen.

Bei den prozeduralen Datenbanksprachen stellen wir zusammenfassend fest, dass sie satzorientierte oder navigierende Befehle für das Bearbeiten von Datensammlungen verlangen. Dieser Sachverhalt setzt von den Anwendungsentwicklern einigen Sachverstand und Kenntnis der inneren Struktur der Datenbank voraus. Zudem kann ein gelegentlicher Benutzer eine Datenbank nicht selbstständig auswerten. Im Gegensatz zu den prozeduralen Sprachen müssen bei relationalen Abfrage- und Manipulationssprachen keine Zugriffspfade, Verarbeitungsabläufe oder Navigationswege spezifiziert werden. Dadurch wird der Entwicklungsaufwand für Datenbankauswertungen wesentlich reduziert.

Möchte man Datenbankabfragen und -auswertungen von den Fachabteilungen oder von den Endbenutzern selbst durchführen lassen, so kommt dem deskriptiven Ansatz eine große Bedeutung zu. Untersuchungen deskriptiver Datenbankschnittstellen haben offengelegt, dass ein *gelegentlicher Benutzer eine echte Chance* hat, mit Hilfe deskriptiver Sprachelemente seine gewünschten Auswertungen selbstständig durchführen zu können. Aus Abb. 1.5 ist zudem ersichtlich, dass die Sprache SQL der natürlichen Sprache nahesteht. So existieren heute relationale Datenbanksysteme, die über einen natürlichsprachlichen Zugang verfügen.

natürliche Sprache:

«Selektiere den Namen der Mitarbeiter, welche in Liestal wohnen»

deskriptive Sprache:

```
SELECT   Name
FROM     MITARBEITER
WHERE    Ort = 'Liestal'
```

prozedurale Sprache:

```
get first  MITARBEITER
    search argument (Ort = 'Liestal')
while status = 0 do
begin
   print (Name)
   get next  MITARBEITER
       search argument (Ort = 'Liestal')
end
```

Abb. 1.5 Unterschied zwischen deskriptiven und prozeduralen Sprachen

1.2.3 Relationales Datenbanksystem

Für die Entwicklung und den Betrieb von Informationssystemen werden Datenbanken eingesetzt, um die Daten zentral, strukturiert und persistent (dauerhaft) zu speichern.

Ein relationales Datenbanksystem ist gemäß Abb. 1.6 ein integriertes System zur einheitlichen Verwaltung von Tabellen. Neben Dienstfunktionen stellt es die deskriptive Sprache SQL für Datenbeschreibungen, Datenmanipulationen und -selektionen zur Verfügung.

Jedes relationale Datenbanksystem besteht aus einer Speicherungs- und einer Verwaltungskomponente: Die Speicherungskomponente dient dazu, sowohl Daten als auch Beziehungen zwischen ihnen lückenlos in Tabellen abzulegen. Neben Tabellen mit Benutzerdaten aus unterschiedlichen Anwendungen existieren vordefinierte Systemtabellen, die beim Betrieb der Datenbanken benötigt werden. Diese enthalten Beschreibungsinformationen und lassen sich durch die Anwendenden jederzeit abfragen, nicht aber verändern.

Die Verwaltungskomponente enthält als wichtigsten Bestandteil die relationale Datendefinitions-, Datenselektions- und Datenmanipulationssprache SQL. Daneben umfasst diese Sprache auch Dienstfunktionen für die Wiederherstellung von Datenbeständen nach einem Fehlerfall, zum Datenschutz und zur Datensicherung.

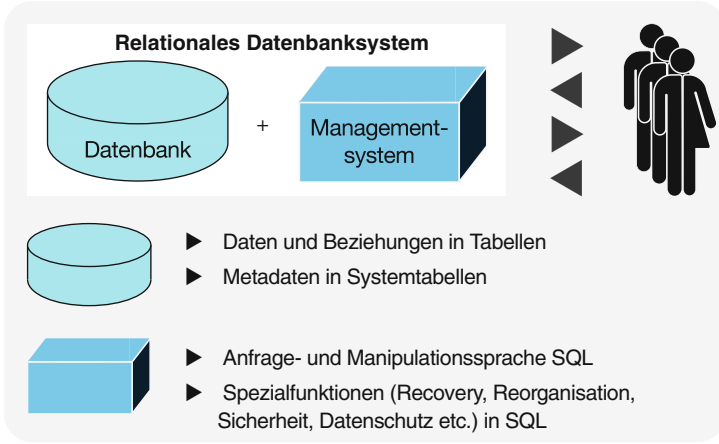

Abb. 1.6 Grundstruktur eines relationalen Datenbanksystems

Relationale Datenbanksysteme bilden oft die Grundlage betrieblicher Informationssysteme und lassen sich wie folgt definieren:

Relationales Datenbanksystem
Ein *relationales Datenbanksystem* (engl. *relational database system*) ist durch folgende Eigenschaften charakterisiert:

- **Modell**: Das Datenbankmodell unterliegt dem Relationenmodell, d. h. alle Daten und Datenbeziehungen werden in Form von Tabellen ausgedrückt. Abhängigkeiten zwischen den Merkmalswerten von Tupeln oder mehrfach vorkommende Sachverhalte können aufgedeckt werden (vgl. Normalformen in Abschn. 2.3.1).
- **Schema:** Die Definition der Tabellen und der Merkmale werden im relationalen Datenbankschema abgelegt. Dieses enthält zudem die Definition der Identifikationsschlüssel sowie Regeln zur Gewährung der Integrität.
- **Sprache**: Das Datenbanksystem umfasst SQL für Datendefinition, -selektion und -manipulation. Die Sprachkomponente ist deskriptiv und entlastet die Anwendenden bei Auswertungen oder bei Programmiertätigkeiten.
- **Architektur**: Das System gewährleistet eine große *Datenunabhängigkeit*, d. h. Daten und Anwendungsprogramme bleiben weitgehend voneinander getrennt. Diese Unabhängigkeit ergibt sich aus der Tatsache, dass die eigentliche Speicherungskomponente aus Sicht der Anwendenden durch eine Verwaltungskomponente entkoppelt ist. Im Idealfall können physische Änderungen in den relationalen Datenbanken vorgenommen werden, ohne dass die entsprechenden Anwendungsprogramme anzupassen sind.

- **Mehrbenutzerbetrieb**: Das System unterstützt den Mehrbenutzerbetrieb (vgl. Abschn. 4.1), d. h. es können mehrere Benutzer gleichzeitig ein und dieselbe Datenbank abfragen oder bearbeiten. Das relationale Datenbanksystem sorgt dafür, dass parallel ablaufende Transaktionen auf einer Datenbank sich nicht gegenseitig behindern oder gar die Korrektheit der Daten beeinträchtigen (Abschn. 4.2).
- **Konsistenzgewährung**: Das Datenbanksystem stellt Hilfsmittel zur Gewährleistung der Datenintegrität bereit. Unter Datenintegrität versteht man die fehlerfreie und korrekte Speicherung der Daten.
- **Datensicherheit und Datenschutz**: Das Datenbanksystem bietet Mechanismen für den Schutz der Daten vor Zerstörung, vor Verlust und vor unbefugtem Zugriff (Abschn. 3.8).

NoSQL-Datenbanksysteme erfüllen obige Eigenschaften nur teilweise (siehe Abschn. 1.4.3 resp. Kap. 4 und 7). Aus diesem Grund sind die relationalen Datenbanksysteme in den meisten Unternehmen, Organisationen und vor allem in KMU's (kleinere und mittlere Unternehmen) nicht mehr wegzudenken. Bei massiv verteilten Anwendungen im Web hingegen oder bei sogenannten Big Data Anwendungen muss die relationale Datenbanktechnologie mit NoSQL-Technologien ergänzt werden, um Webdienste rund um die Uhr und weltweit anbieten zu können.

1.3 Big Data

Mit dem Schlagwort «Big Data» werden umfangreiche Datenbestände bezeichnet, die mit herkömmlichen Softwarewerkzeugen kaum mehr zu bewältigen sind. Die Daten sind meistens unstrukturiert (siehe Abschn. 5.1) und stammen aus den unterschiedlichsten Quellen: Mitteilungen (Postings) aus sozialen Netzwerken, Emails, elektronischen Archiven mit Multimedia-Inhalten, Anfragen aus Suchmaschinen, Dokumentsammlungen von Content Management Systemen, Sensordaten beliebiger Art, Kursentwicklungen von Börsenplätzen, Daten aus Verkehrsströmen und Satellitenbildern, Messdaten von Geräten des Haushalts (Smart Meter); Bestell-, Kauf- und Bezahlvorgängen elektronischer Shops, Daten aus eHealth-Anwendungen, Aufzeichnungen von Monitoring-Systemen etc.

Für den Begriff Big Data gibt es noch keine verbindliche Definition, doch die meisten Datenspezialisten berufen sich auf mindestens drei V's: *Volume* (umfangreicher Datenbestand), *Variety* (Vielfalt von Datenformaten; strukturierte, semi-strukturierte und unstrukturierte Daten; siehe Abb. 1.7) und *Velocity* (hohe Verarbeitungsgeschwindigkeit und Echtzeitverarbeitung).

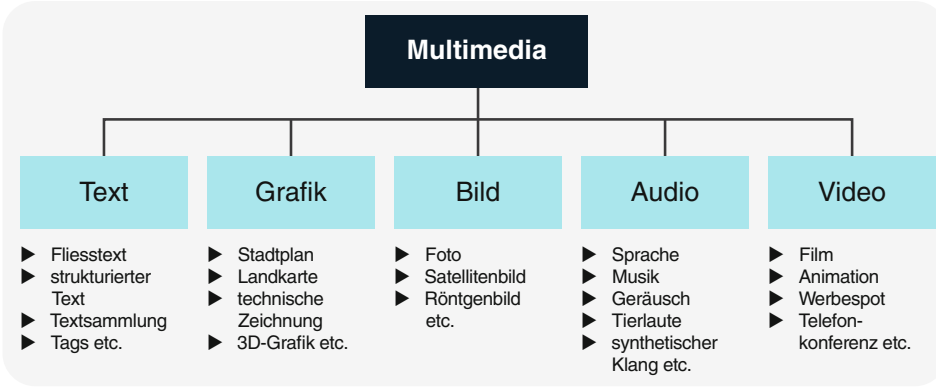

Abb. 1.7 Vielfalt der Quellen bei Big Data

Im IT-Glossar der Gartner Group findet sich folgende Definition:

Big Data
‚*Big data is high-volume, high-velocity and high-variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making*‘.[3]

Also: Big Data ist ein umfangreiches, schnell zu verarbeitendes und vielfältiges Informationskapital, das nach kosteneffektiven und innovativen Formen der Informationsverarbeitung verlangt, um einen besseren Einblick und eine bessere Grundlage für Entscheidungsfindungen zu ermöglichen.

Die Gartner Group geht soweit, Big Data als *Informationskapital* oder *Vermögenswert* (engl. *information asset*) des Unternehmens zu betrachten. Tatsächlich müssen Unternehmen und Organisationen entscheidungsrelevantes Wissen generieren, um überleben zu können. Dabei setzen sie neben den eigenen Informationssystemen vermehrt auf die Vielfalt der Ressourcen im Web, um ökonomische, ökologische und gesellschaftliche Entwicklungen im Markt besser antizipieren zu können.

Big Data ist nicht nur eine Herausforderung für profitorientierte Unternehmen im elektronischen Geschäft, sondern auch für das Aufgabenspektrum von Regierungen, öffentlichen Verwaltungen, NGO’s (Non Governmental Organizations) und NPO’s (Non Profit Organizations).

Als Beispiel seien die Programme für Smart City oder Ubiquitous City erwähnt, d. h. die Nutzung von Big Data Technologien in Städten und Agglomerationen. Ziel dabei ist,

[3] Gartner Group, IT Glossary – Big Data. http://www.gartner.com/it-glossary/big-data/. Zugegriffen am 11. Februar 2015.

den sozialen und ökologischen Lebensraum nachhaltig zu entwickeln. Dazu zählen beispielsweise Projekte zur Verbesserung der Mobilität, Nutzung intelligenter Systeme für Wasser- und Energieversorgung, Förderung sozialer Netzwerke, Erweiterung politischer Partizipation, Ausbau von Entrepreneurship, Schutz der Umwelt oder Erhöhung von Sicherheit und Lebensqualität.

Zusammengefasst geht es bei Big Data Anwendungen um die Beherrschung der folgenden drei V's:

- **Volume**: Der Datenbestand ist umfangreich und liegt im Tera- bis Zettabytebereich (Megabyte $= 10^6$ Byte, Gigabyte $= 10^9$ Byte, Terabyte $= 10^{12}$ Byte, Petabyte $= 10^{15}$ Byte, Exabyte $= 10^{18}$ Byte, Zettabyte $= 10^{21}$ Byte).
- **Variety**: Unter Vielfalt versteht man bei Big Data die Speicherung von strukturierten, semi-strukturierten und unstrukturierten Multimedia-Daten (Text, Grafik, Bilder, Audio und Video; vgl. Abb. 1.7).
- **Velocity**: Der Begriff bedeutet Geschwindigkeit und verlangt, dass Datenströme (engl. *data streams*, vgl. Abschn. 5.1) in Echtzeit ausgewertet und analysiert werden können.

In der Definition der Gartner Group wird von Informationskapital oder Vermögenswert gesprochen. Aus diesem Grund fügen einige Experten ein weiteres V hinzu:

- **Value**: Big Data Anwendungen sollen den Unternehmenswert steigern. Investitionen in Personal und technischer Infrastruktur werden dort gemacht, wo eine Hebelwirkung besteht resp. ein Mehrwert generiert werden kann.

Viele Angebote für NoSQL-Datenbanken sind Open Source. Zudem zeichnen sich diese Technologien dadurch aus, dass sie mit kostengünstiger Hardware auskommen und leicht skalierbar sind. Auf der anderen Seite besteht ein Engpass bei gut ausgebildetem Personal, denn das Berufsbild des *Data Scientist* (siehe Abschn. 1.5) ist erst im Entstehen. Entsprechende Ausbildungsangebote sind in Diskussion oder in der Pilotphase.

Ein letztes V soll die Diskussion zur Begriffsbildung von Big Data abrunden:

- **Veracity**: Da viele Daten vage oder ungenau sind, müssen spezifische Algorithmen zur Bewertung der Aussagekraft resp. zur Qualitätseinschätzung der Resultate verwendet werden. Umfangreiche Datenbestände garantieren nicht per se bessere Auswertungsqualität.

Veracity bedeutet in der deutschen Übersetzung Aufrichtigkeit oder Wahrhaftigkeit. Im Zusammenhang mit Big Data wird damit ausgedrückt, dass Datenbestände in unterschiedlicher Datenqualität vorliegen und dass dies bei Auswertungen berücksichtigt werden muss. Neben statistischen Verfahren existieren unscharfe Methoden des Soft Computing, die einem Resultat oder einer Aussage einen Wahrheitswert zwischen 0 (falsch) und 1 (wahr) zuordnen (vgl. Fuzzy-Datenbanken in Abschn. 6.8).

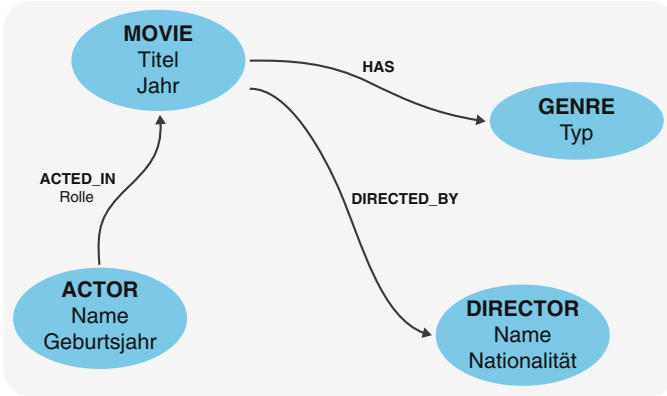

Abb. 1.8 Ausschnitt Graphenmodell für Filme

1.4 NoSQL-Datenbanken

1.4.1 Graphenmodell

NoSQL-Datenbanken unterstützen unterschiedliche Datenbankmodelle (siehe Abschn. 1.4.3 resp. Abb. 1.11). Hier greifen wir das Graphenmodell heraus und diskutieren dessen Eigenschaften:

Property Graph
Ein *Eigenschaftsgraph* (engl. *property graph*) besteht aus *Knoten* (Konzepte, Objekte) und *gerichteten Kanten* (Beziehungen), die Knoten verbinden. Sowohl die Knoten wie die Kanten tragen einen Namen (engl. *label*) und können *Eigenschaften* (engl. *properties*) aufweisen. Die Eigenschaften werden als Attribut-Wert-Paare der Form (*attribute*: value) mit Attributnamen und zugehörigen Werten charakterisiert.

Im Graphenmodell werden abstrakt die Knoten und Kanten mit ihren Eigenschaften dargestellt. Als Beispiel wird in Abb. 1.8 ein Ausschnitt einer Filmsammlung aufgezeigt. Dieser besteht aus den Knoten MOVIE mit den Attributen *Titel* und *Jahr* (Erscheinungsjahr), GENRE mit dem jeweiligen *Typ* (z. B. Unterhaltung, Krimi, Komödie, Drama, Thriller, Western, Science-Fiction, Dokumentarfilm etc.), dem ACTOR (Schauspieler) mit *Name* und *Geburtsjahr* sowie dem DIRECTOR (Regisseur) mit *Name* und *Nationalität*.

Das Beispiel aus Abb. 1.8 verwendet drei gerichtete Kanten. Die Kante ACTED-IN drückt aus, welcher Schauspieler aus dem Knoten ACTOR bei welchem Film aus dem Knoten MOVIE mitgewirkt hat. Die Kante ACTED_IN hat zudem eine Eigenschaft, und

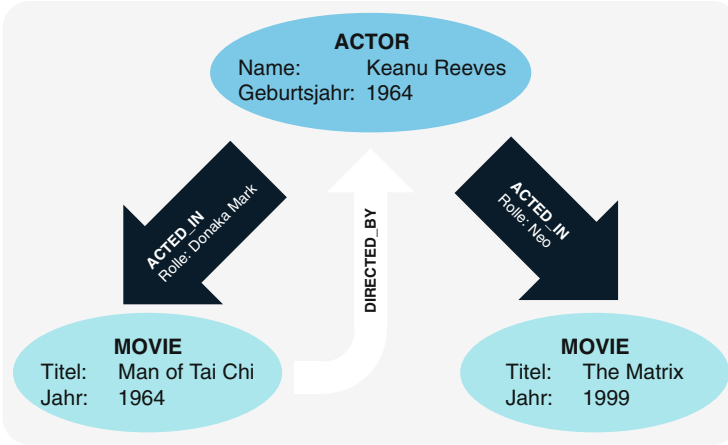

Abb. 1.9 Ausschnitt einer Graphdatenbank über Filme

zwar die *Rolle* des Schauspielers. Die beiden anderen Kanten, HAS und DIRECTED_BY, führen vom Knoten MOVIE in den Knoten GENRE resp. DIRECTOR.

Auf der Ausprägungsebene resp. in der Graphdatenbank enthält der Property Graph dann die konkreten Vorkommen (vgl. Abb. 1.9 in Abschn. 1.4.2).

Das Graphenmodell für Datenbanken ist formal abgestützt durch die Graphentheorie. Je nach Reifegrad enthalten die entsprechenden Softwareprodukte Algorithmen zur Berechnung folgender Eigenschaften:

- **Zusammenhang**: Ein Graph ist zusammenhängend, falls es zu jedem Knoten im Graphen einen Pfad zu jedem anderen Knoten des Graphen gibt.
- **Kürzester Pfad**: Der kürzeste Pfad oder Weg zwischen zwei Knoten eines Graphen ist derjenige, der die geringste Anzahl Kanten zwischen den Knoten aufweist.
- **Nächster Nachbar**: Sind die Kanten eines Graphen gewichtet (z. B. durch Längen- oder Zeitangaben bei einem Transportnetz), so lassen sich die nächsten Nachbarn eines Knoten berechnen, indem nach den minimalsten Abständen (kürzester Weg in Meter resp. Kilometer, kürzester Weg in Stunden resp. Minuten) gesucht wird.
- **Matching**: In der Graphentheorie versteht man unter dem Matching resp. dem Zuordnungsproblem die Berechnung einer Menge von Kanten, die keine gemeinsamen Knoten enthalten.

Diese Grapheigenschaften sind bedeutend für viele Anwendungen. Beispielsweise ist die Berechnung des kürzesten Pfades oder des nächsten Nachbarn wichtig, wenn es um die Berechnung von Reise- oder Transportrouten geht. Darüber hinaus können obige Algorithmen die Freundschaftsbeziehungen in einem sozialen Netz nach Pfadlängen ordnen und auswerten (Abschn. 2.4).

1.4.2 Graphenorientierte Abfragesprache Cypher

Cypher ist eine *deklarative Abfragesprache*, um Muster in Graphdatenbanken extrahieren zu können. Die Anwendenden spezifizieren ihre Suchfrage durch die Angabe von Knoten und Kanten. Daraufhin berechnet das Datenbanksystem alle gewünschten Muster, indem es die möglichen Pfade (Verbindungen zwischen Knoten via Kanten) auswertet. Mit anderen Worten deklarieren die Anwendenden die Eigenschaften des gesuchten Musters, und die Algorithmen des Datenbanksystems traversieren alle notwendigen Verbindungen (Pfade) und stellen das Resultat zusammen.

Gemäß Abschn. 1.4.1 besteht das Datenmodell einer Graphdatenbank aus *Knoten* (Konzepte, Objekte) und *gerichteten Kanten* (Beziehungen zwischen den Knoten). Sowohl Knoten wie Kanten können neben ihrem Namen eine Menge von Eigenschaften haben (vgl. Property Graph in Abschn. 1.4.1). Die Eigenschaften werden durch Attribut-Wert-Paare ausgedrückt.

In Abb. 1.9 ist ein Ausschnitt einer Graphdatenbank über Filme und Schauspieler aufgezeigt. Der Einfachheit halber werden nur zwei Knotentypen aufgeführt: ACTOR und MOVIE. Die Knoten für die Schauspieler enthalten zwei Attribut-Wert-Paare, nämlich (*Name*: Vorname Nachname) und (*Geburtsjahr*: Jahr).

Der Ausschnitt aus Abb. 1.9 zeigt unterschiedliche Kantentypen: Die Beziehung ACTED_IN drückt aus, welche Schauspieler in welchen Filmen mitwirkten. Kanten können Eigenschaften besitzen, falls man Attribut-Wert-Paare anfügt. Bei der Beziehung ACTECD_IN werden jeweils die Rollen aufgeführt, welche die Schauspieler gespielt hatten. So war beispielsweise Keanu Reeves im Film ‚The Matrix' als Hacker unter dem Namen Neo engagiert.

Knoten können durch unterschiedliche Beziehungskanten miteinander verbunden sein. Zwischen dem Film ‚Man of Tai Chi' und dem Schauspieler Keanu Reeves gibt es nicht nur die Darstellerrolle (ACTED_IN), sondern auch die Regieführung (DIRECTED_BY). Daraus geht hervor, dass Keanu Reeves im Film ‚Man of Tai Chi' einerseits Regie führte und andrerseits Donaka Mark darstellte.

Möchte man die Graphdatenbank über Filme auswerten, so kann Cypher verwendet werden. Die Grundelemente des Abfrageteils von Cypher sind die Folgenden:

- MATCH: Identifikation von Knoten und Kanten sowie Deklaration von Suchmustern.
- WHERE: Bedingungen zur Filterung von Ergebnissen.
- RETURN: Bereitstellung des Resultats, bei Bedarf aggregiert.

Möchte man das Erscheinungsjahr des Films The Matrix berechnen, so lautet die Anfrage in Cypher:

```
MATCH   (m :  Movie {Titel :  „The Matrix})
RETURN  m.Jahr
```

In dieser Abfrage wird die Variable m für den Film ‚The Matrix' losgeschickt, um das Erscheinungsjahr dieses Filmes durch m.Jahr zurückgeben zu lassen. Die runden Klammern drücken bei Cypher immer Knoten aus, d. h. der Knoten (m: Movie) deklariert die Laufvariable m für den Knoten MOVIE. Neben Laufvariablen können konkrete Attribut-Wert-Paare in geschweiften Klammern mitgegeben werden. Da wir uns für den Film ‚The Matrix' interessieren, wird der Knoten (m: Movie) um die Angabe {Titel: „The Matrix"} ergänzt.

Interessant sind nun Abfragen, die die Beziehungen der Graphdatenbank betreffen. Beziehungen zwischen zwei beliebigen Knoten (a) und (b) werden in Cypher durch das Pfeilsymbol „- - > " ausgedrückt, d. h. der Pfad von (a) nach (b) wird durch „(a) - - > (b)" deklariert. Falls die Beziehung zwischen (a) und (b) von Bedeutung ist, wird die Pfeilmitte mit der Kante [r] ergänzt. Die eckigen Klammern drücken Kanten aus, und r soll hier als Variable für Beziehungen (engl. *relationships*) dienen.

Möchten wir herausfinden, wer im Film ‚The Matrix' den Hacker Neo gespielt hat, so werten wir den entsprechenden Pfad ACTED_IN zwischen ACTOR und MOVIE wie folgt aus:

```
MATCH   (a : Actor) − [: Acted_In {Rolle : „Neo}] − >
                                  (: Movie {Titel : „The Matrix})
RETURN  a.Name
```

Cypher gibt uns als Resultat Keanu Reeves zurück.

Möchten wir eine Liste von Filmtiteln (m), Schauspielernamen (a) und zugehörigen Rollen (r), so lautet die Anfrage:

```
MATCH   (a : Actor) − [r : Acted_In] − > (m : Movie)
RETURN  m.Titel, a.Name, r.Rolle
```

Da die Graphdatenbank nur einen Schauspieler und zwei Filme listet, würde als Resultat der Film ‚Man of Tai Chi' mit Schauspieler Keanu Reeves in der Rolle Donaka Mark sowie der Film ‚The Matrix' mit Keanu Reeves und der Rolle Neo ausgegeben.

Im Normalfall hat die Graphdatenbank mit Schauspielern, Filmen und Rollen- wie Regiebeziehungen unzählige Einträge. Aus diesem Grund müsste die obige Abfrage auf den Schauspieler Keanu Reeves beschränkt bleiben und die Anfrage würde lauten:

```
MATCH   (a : Actor) − [r : Acted_In] − > (m : Movie)
WHERE   (a.name = „Keanu Reeves)
RETURN  m.Titel, a.Name, r.Rolle
```

Bei Cypher werden ähnlich wie bei SQL deklarative Abfragen formuliert. Die Anwendenden spezifizieren das Resultatsmuster (Cypher) oder die Resultatstabelle (SQL) mit den gewünschten Eigenschaften. Das jeweilige Datenbanksystem berechnet daraufhin die Resultate. Allerdings ist das Auswerten von Beziehungsgeflechten, das Verwenden rekursiver Suchstrategien oder die Analyse von Eigenschaften von Graphen mit SQL kaum zu bewältigen.

1.4.3 NoSQL-Datenbanksystem

Nicht-relationale Datenbanken gab es vor der Entdeckung des Relationenmodells durch Ted Codd in der Form von hierarchischen oder netzwerkartigen Datenbanken. Nach dem Aufkommen von relationalen Datenbanksystemen wurden nicht-relationale Ansätze weiterhin für technische oder wissenschaftliche Anwendungen genutzt. Beispielsweise war es schwierig, ein CAD-System (CAD = Computer Aided Design) für Bau- oder Maschinenteile mit relationaler Technologie zu betreiben. Das Aufteilen technischer Objekte in eine Vielzahl von Tabellen war für CAD-Systeme problematisch, da geometrische, topologische und grafische Manipulationen in Echtzeit durchgeführt werden mussten.

Mit dem Aufkommen des Internets und einer Vielzahl von webbasierten Anwendungen haben nicht-relationale Datenkonzepte gegenüber relationalen an Gewicht gewonnen. Es ist schwierig oder teilweise unmöglich, Big Data Anwendungen mit relationaler Datenbanktechnologie zu bewältigen.

Die Bezeichnung ‚nicht-relational' wäre besser geeignet als NoSQL, doch hat sich der Begriff in den letzten Jahren bei Datenbankforschenden wie bei Anbietern im Markt etabliert.

NoSQL
Der Begriff NoSQL wird heute für *nicht-relationale Ansätze im Datenmanagement* verwendet, mit folgenden zwei Bedingungen:

- Erstens: Die Speicherung der Daten erfolgt nicht in Tabellen.
- Zweitens: Die Datenbanksprache ist nicht SQL.

Manchmal wird der Ausdruck NoSQL durch ‚Not only SQL' übersetzt. Damit soll ausgedrückt werden, dass bei einer massiv verteilten Webanwendung nicht nur relationale Datentechnologien zum Einsatz gelangen. Vor allem dort, wo die Verfügbarkeit des Webdienstes im Vordergrund steht, sind NoSQL-Technologien gefragt. Im Abschn. 5.6 wird beispielsweise ein Webshop diskutiert, der unterschiedliche NoSQL-Technologien neben einer relationaler Datenbank einsetzt (vgl. Abb. 5.13).

Die Grundstruktur eines NoSQL-Datenbanksystems ist in Abb. 1.10 skizziert. Meistens unterliegt ein NoSQL-Datenbanksystem einer massiv verteilten Datenhaltungsarchitektur. Die Daten selber werden je nach Typ der NoSQL-Datenbank entweder als Schlüssel-Wert-Paare (engl. *key/value Store*), in Spalten oder Spaltenfamilien (engl. *column*

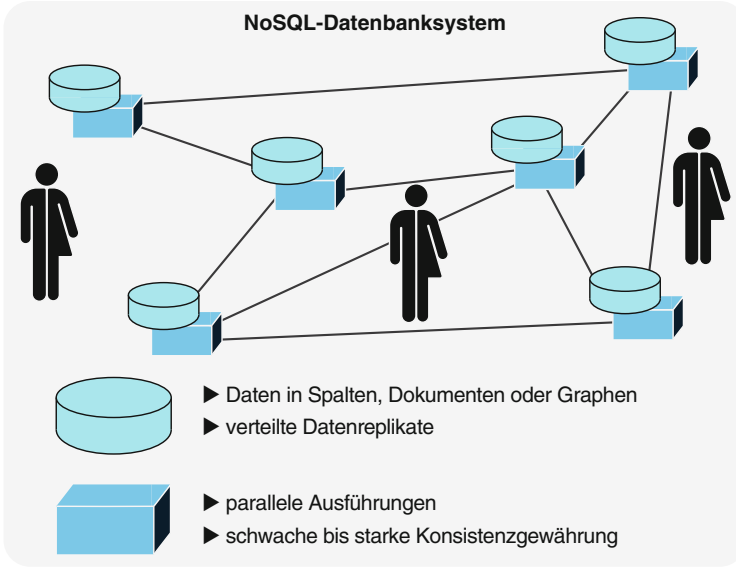

Abb. 1.10 Grundstruktur eines NoSQL-Datenbanksystems

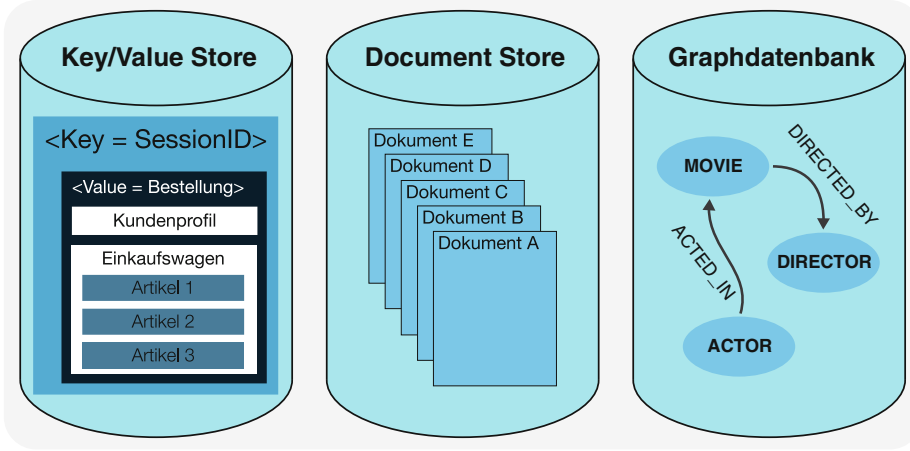

Abb. 1.11 Drei unterschiedliche NoSQL-Datenbanken

store), in Dokumentspeichern (engl. *document store*) oder in Graphen (engl. *graph database*) gehalten (siehe Abb. 1.11 resp. Kap. 7). Um hohe Verfügbarkeit zu gewähren und das NoSQL-Datenbanksystem gegen Ausfälle zu schützen, werden unterschiedliche Replikationskonzepte unterstützt (vgl. das Konzept „Consistent Hashing“ in Abschn. 5.2.3).

Bei einer massiv verteilten und replizierten Rechnerarchitektur können parallele Auswertungsverfahren genutzt werden (vgl. „Map/Reduce“ in Abschn. 5.4). Die Analyse

umfangreicher Datenvolumen oder das Suchen nach bestimmten Sachverhalten kann mit verteilten Berechnungsvorgängen beschleunigt werden. Beim Map/Reduce-Verfahren werden Teilaufgaben an diverse Rechnerknoten verteilt und einfache Schlüssel-Wert-Paare extrahiert (Map), bevor die Teilresultate zusammengefasst und ausgegeben werden (Reduce).

In massiv verteilten Rechnernetzen werden zudem differenzierte Konsistenzkonzepte angeboten (vgl. Abschn. 4.3). Unter starker Konsistenz (engl. *strong consistency*) wird verstanden, dass das NoSQL-Datenbanksystem die Konsistenz jederzeit gewährleistet. Bei der schwachen Konsistenzforderung (engl. *weak consistency*) wird toleriert, dass Änderungen auf replizierten Knoten verzögert durchgeführt und zu kurzfristigen Inkonsistenzen führen können. Daneben existieren weitere Differenzierungsoptionen, wie beispielsweise „Consistency by Quorum“ (vgl. Abschn. 4.3.2).

Die folgende Definition für NoSQL-Datenbanken ist angelehnt an das webbasierte NoSQL-Archiv[4]:

NoSQL-Datenbanksystem
Webbasierte Speichersysteme werden als *NoSQL-Datenbanksysteme* (engl. *NoSQL database systems*) bezeichnet, falls sie folgende Bedingungen erfüllen:

- **Modell**: Das zugrundeliegende Datenbankmodell ist nicht relational.
- **Mindestens 3 V**: Das Datenbanksystem erfüllt die Anforderungen für umfangreiche Datenbestände (Volume), flexible Datenstrukturen (Variety) und Echtzeitverarbeitung (Velocity).
- **Schema**: Das Datenbanksystem unterliegt keinem fixen Datenbankschema.
- **Architektur**: Die Datenbankarchitektur unterstützt massiv verteilte Webanwendungen und horizontale Skalierung.
- **Replikation**: Das Datenbanksystem unterstützt die Datenreplikation.
- **Konsistenzgewährung**: Aufgrund des CAP-Theorems (vgl. Abschn. 4.3.1) ist die Konsistenz lediglich verzögert gewährleistet (weak consistency), falls hohe Verfügbarkeit und Ausfalltoleranz angestrebt werden.

Die Forscher und Betreiber des NoSQL-Archivs listen auf ihrer Webplattform 150 NoSQL-Datenbankprodukte. Der Großteil dieser Systeme ist Open Source. Allerdings zeigt die Vielfalt der Angebote auf, dass der Markt der NoSQL-Lösungen noch unsicher ist. Zudem müssen für den Einsatz von geeigneten NoSQL-Technologien Spezialisten gefunden werden, die nicht nur die Konzepte beherrschen, sondern auch die vielfältigen Architekturansätze und Werkzeuge.

In Abb. 1.11 sind drei unterschiedliche NoSQL-Datenbanksysteme aufgezeigt.

Sogenannte *Schlüssel-Wert-Datenbanken* (vgl. Abschn. 7.2) sind die einfachsten Systeme. Sie speichern die Daten in der Form eines Identifikationsschlüssels <Key = „Schlüssel“>

[4] NoSQL-Archiv. http://nosql-database.org/. Zugegriffen am 17. Februar 2015.

und einer Liste von Wertvorkommen <Value = „Wert 1“, „Wert 2“, . . .>. Als Beispiel einer solchen Datenbank dient ein Webshop mit Session Management und Einkaufskorb. Die SessionID dient als Identifikationsschlüssel; als Werte werden neben dem Kundenprofil die einzelnen Artikel im Einkaufswagen abgelegt.

Die *Dokumentdatenbank* (siehe Abschn. 7.4) ist eine NoSQL-Datenbank, die Datensätze als Dokumente verwaltet. Diese sind strukturierte Text-Daten, beispielsweise im JSON- oder XML-Format, welche über einen eindeutigen Schlüssel oder über Merkmale im Dokument auffindbar sind. Im Unterschied zu den Schlüssel-Wert-Datenbanken haben somit die Dokumente bereits eine gewisse Struktur, diese ist aber schemafrei, d. h. die Struktur jedes einzelnen Datensatzes (Dokuments) kann beliebig sein.

Als drittes Beispiel einer NoSQL-Datenbank dient die *Graphdatenbank* über Filme und Schauspieler, die in den vorigen Abschnitten diskutiert wurde (siehe zu Graphdatenbanken vertieft auch Abschn. 7.6).

1.5 Organisation des Datenmanagements

Viele Firmen und Institutionen betrachten ihre Datenbestände als *unentbehrliche Ressource*. Sie pflegen und unterhalten zu Geschäftszwecken nicht nur ihre eigenen Daten, sondern schließen sich mehr und mehr an öffentlich zugängliche Datensammlungen an. Die weltweite Zunahme und das stetige Wachstum der Informationsanbieter mit ihren Dienstleistungen rund um die Uhr untermauern den Stellenwert webbasierter Datenbestände.

Die Bedeutung aktueller und realitätsbezogener Information hat einen direkten Einfluss auf die Ausgestaltung des Informatikbereiches. So sind vielerorts Stellen des Datenmanagements entstanden, um die datenbezogenen Aufgaben und Pflichten bewusster angehen zu können. Ein zukunftgerichtetes Datenmanagement befasst sich sowohl strategisch mit der Informationsbeschaffung und -bewirtschaftung als auch operativ mit der effizienten Bereitstellung und Auswertung von aktuellen und konsistenten Daten.

Aufbau und Betrieb eines Datenmanagements verursachen beträchtliche Kosten mit anfänglich nur schwer messbarem Nutzen. Es ist nicht einfach, eine flexible Datenarchitektur, widerspruchsfreie und für jedermann verständliche Datenbeschreibungen, saubere und konsistente Datenbestände, griffige Sicherheitskonzepte, aktuelle Auskunftsbereitschaft und weiteres mehr eindeutig zu bewerten und aussagekräftig in Wirtschaftlichkeitsüberlegungen mit einzubeziehen. Erst ein allmähliches Bewusstwerden von Bedeutung und Langlebigkeit der Daten relativiert für das Unternehmen die notwendigen Investitionen.

Um den Begriff *Datenmanagement* (engl. *data management*) besser fassen zu können, sollte das Datenmanagement zunächst in seine Aufgabenbereiche Datenarchitektur, Datenadministration, Datentechnik und Datennutzung aufgegliedert werden. Die Abb. 1.12 charakterisiert diese vier Teilgebiete des Datenmanagements mit ihren Zielen und Instrumenten.

	Ziele	Werkzeuge
Datenarchitektur	▶ Formulieren und Pflegen der unternehmensweiten Datenarchitektur ▶ Festlegen von Datenschutzkonzepten	▶ Datenanalyse und Entwurfsmethodik ▶ Werkzeuge der rechnergestützten Informationsmodellierung
Datenadministration	▶ Verwalten von Daten und Methoden anhand von Standardisierungsrichtlinien und internationalen Normen ▶ Beraten von Entwicklern und Endbenutzern	▶ Data Dictionary Systeme ▶ Werkzeuge für den Verwendungsnachweis
Datentechnik	▶ Installieren, reorganisieren und sicherstellen von Datenbeständen ▶ Festlegen des Verteilungskonzeptes inkl. Replikation ▶ Katastrophenvorsorge	▶ Diverse Dienste der Datenbanksysteme ▶ Werkzeuge der Leistungsoptimierung ▶ Monitoring Systeme ▶ Werkzeuge für Recovery/Restart
Datennutzung	▶ Datenanalyse und -interpretation ▶ Wissensgenerierung ▶ Erstellen von Prognosen ▶ Mustererkennung	▶ Auswertungswerkzeuge ▶ Reportgeneratoren ▶ Werkzeuge des Data Mining ▶ Visualisierungstechniken für mehrdimensionale Daten

Abb. 1.12 Die vier Eckpfeiler des Datenmanagements

Mitarbeitende der *Datenarchitektur* analysieren, klassifizieren und strukturieren mit ausgefeilter Methodik die Unternehmensdaten. Neben der eigentlichen Analyse der Daten- und Informationsbedürfnisse müssen die wichtigsten Datenklassen und ihre gegenseitigen Beziehungen untereinander in unterschiedlichster Detaillierung in Datenmodellen festgehalten werden. Diese aus der Abstraktion der realen Gegebenheiten entstandenen und aufeinander abgestimmten Datenmodelle bilden die Basis der Datenarchitektur.

Die *Datenadministration* bezweckt, die Datenbeschreibungen und die Datenformate sowie deren Verantwortlichkeiten einheitlich zu erfassen, um eine anwendungsübergreifende Nutzung der langlebigen Unternehmensdaten zu gewährleisten. Beim heutigen Trend zu einer dezentralen Datenhaltung auf intelligenten Arbeitsplatzrechnern oder auf verteilten Abteilungsrechnern kommt der Datenadministration eine immer größere Verantwortung bei der Pflege der Daten und bei der Vergabe von Berechtigungen zu.

Die Spezialisten der *Datentechnik* installieren, überwachen und reorganisieren Datenbanken und stellen diese in einem mehrstufigen Verfahren sicher. Dieser Fachbereich, oft Datenbanktechnik oder Datenbankadministration genannt, ist zudem für das Technologiemanagement verantwortlich, da Erweiterungen in der Datenbanktechnologie immer wieder berücksichtigt und bestehende Methoden und Werkzeuge laufend verbessert werden müssen.

Der vierte Eckpfeiler des Datenmanagements, die *Datennutzung*, ermöglicht die eigentliche Bewirtschaftung der Unternehmensdaten. Mit einem besonderen Team von Datenspezialisten (Berufsbild Data Scientist, siehe unten) wird das *Business Analytics* vorangetrieben, das der Geschäftsleitung und dem Management periodisch Datenanalysen erarbeitet und rapportiert. Zudem unterstützen diese Spezialisten diverse Fachabteilungen wie Marketing, Verkauf, Kundendienst etc., um spezifische Erkenntnisse aus Big Data zu generieren.

Somit ergibt sich von der Charakterisierung der datenbezogenen Aufgaben und Pflichten her gesehen für das Datenmanagement folgende Definition:

Datenmanagement
Unter dem *Datenmanagement* (engl. *data management*) fasst man alle *betrieblichen, organisatorischen und technischen Funktionen* der Datenarchitektur, der Datenadministration und der Datentechnik zusammen, die der unternehmensweiten *Datenhaltung, Datenpflege, Datennutzung sowie dem Business Analytics* dienen.

Für das Datenmanagement sind im Laufe der Jahre unterschiedliche Berufsbilder entstanden. Die wichtigsten lauten:

- **Datenarchitektinnen und -architekten**: Datenarchitekte sind für die unternehmensweite Datenarchitektur verantwortlich. Aufgrund der Geschäftsmodelle entscheiden sie, wo und in welcher Form Datenbestände bereitgestellt werden müssen. Für die Fragen der Verteilung, Replikation oder Fragmentierung der Daten arbeiten sie mit den Datenbankspezialisten zusammen.
- **Datenbankspezialistinnen und -spezialisten**: Die Datenbankspezialisten beherrschen die Datenbank- und Systemtechnik und sind für die physische Auslegung der Datenarchitektur verantwortlich. Sie entscheiden, welche Datenbanksysteme (SQL- und/oder NoSQL-Technologien) für welche Komponenten der Anwendungsarchitektur eingesetzt werden. Zudem legen sie das Verteilungskonzept fest und sind zuständig für die Archivierung, Reorganisation und Restaurierung der Datenbestände.
- **Data Scientists**: Die Data Scientists sind die Spezialisten des Business Analytics. Sie beschäftigen sich mit der Datenanalyse und -interpretation, extrahieren noch nicht bekannte Fakten aus den Daten (Wissensgenerierung) und erstellen bei Bedarf Zukunftsprognosen über die Geschäftsentwicklung. Sie beherrschen die Methoden und Werkzeuge des Data Mining (Mustererkennung), der Statistik und der Visualisierung von mehrdimensionalen Zusammenhängen unter den Daten.

Die hier vorgeschlagene Begriffsbildung zum Datenmanagement sowie zu den Berufsbildern umfasst technische, organisatorische wie betriebliche Funktionen. Dies bedeutet allerdings nicht zwangsläufig, dass in der Aufbauorganisation eines Unternehmens oder Organisation die Funktionen der Datenarchitektur, der Datenadministration, der Datentechnik und der Datennutzung in einer einzigen Organisationseinheit zusammengezogen werden müssen.

1.6 Literatur

Es gibt eine große Anzahl von Standardwerken zum Gebiet der Datenbanken, woran sich die Bedeutung dieses Informatikbereiches ablesen lässt. Einige Lehrbücher behandeln nicht nur relationale, sondern auch verteilte, objektorientierte oder wissensbasierte Datenbanksysteme. Bekannt sind die Werke von Connolly und Begg (2014); Hoffer et al. (2012) sowie von Date (2004), eher theoretisch ist das Textbuch von Ullman (1982), aufschlussreich dasjenige von Silberschatz et al. (2010) und umfassend die Standardwerke von Elmasri und Navathe (2015) sowie von Garcia-Molina et al. (2014). Gardarin und Valduriez (1989) geben eine Einführung in die relationale Datenbanktechnologie sowie in das Gebiet der Wissensbanken.

Als deutschsprachige Werke im Datenbankbereich sind Lang und Lockemann (1995), Schlageter und Stucky (1983); Wedekind (1981) und Zehnder (2005) zu erwähnen. Die bekannten Lehrbücher von Saake et al. (2013); Kemper und Eickler (2013) sowie von Vossen (2008) gehen auf Grundlagen und Erweiterungen von Datenbanksystemen ein. Die verwendete Definition des Datenbankbegriffs lehnt sich an Claus und Schwill (2001).

Was Big Data betrifft, so ist der Markt in den letzten Jahren mit Büchern überschwemmt worden. Allerdings beschreiben die meisten Werke den Trend nur oberflächlich. Zwei englische Kurzeinführungen, das Buch von Celko (2014) sowie dasjenige von Sadalage und Fowler (2013), erläutern die Begriffe und stellen die wichtigsten NoSQL-Datenbankansätze vor. Für technisch Interessierte gibt es das Werk von Redmond und Wilson (2012), die sieben Datenbanksysteme konkret erläutern.

Erste deutschsprachige Veröffentlichungen zum Themengebiet Big Data gibt es ebenfalls. Das Buch von Edlich et al. (2011) gibt eine Einführung in NoSQL-Datenbanktechnologien, bevor unterschiedliche Datenbankprodukte für Key/Value Store, Document Store, Column Store und Graphdatenbanken vorgestellt werden. Das Werk von Freiknecht (2014) beschreibt das bekannte System Hadoop (Framework für skalierbare und verteilte Systeme) inkl. der Komponenten für die Datenhaltung (HBase) und für das Data Warehousing (Hive). Das Herausgeberwerk von Fasel und Meier (2016) gibt einen Überblick über die Big Data Entwicklung im betrieblichen Umfeld. Die wichtigsten NoSQL-Datenbanken werden vorgestellt, Fallbeispiele diskutiert, rechtliche Aspekte erläutert und Umsetzungshinweise gegeben.

Ein Fachbuch über betriebliche Aspekte des Datenmanagements stammt von Dippold et al. (2005). Biethahn et al. (2000) widmen in ihrem Band über das Daten- und Entwicklungsmanagement der Datenarchitektur und der Datenadministration mehrere Kapitel. Heinrich und Lehner (2005) sowie Österle et al. (1991) streifen in ihren Werken über das Informationsmanagement Themen des Datenmanagements. Meier (1994) charakterisiert in seinem Beitrag die Ziele und Aufgaben des Datenmanagements aus der Sicht des Praktikers. Fragen der Datenadministration werden von Meier und Johner (1991) sowie von Ortner et al. (1990) aufgegriffen.

Datenmodellierung

2

2.1 Von der Datenanalyse zur Datenbank

Ein *Datenmodell* (engl. *data model*) beschreibt auf strukturierte und formale Weise die für ein Informationssystem notwendigen Daten und Datenbeziehungen. Benötigt man für die Bearbeitung von Informatikprojekten gemäß Abb. 2.1 Informationen über Mitarbeitende, Detailangaben über Projektvorhaben und Auskunft über einzelne Firmenabteilungen, so können in einem entsprechenden Datenmodell die dazu notwendigen Datenklassen (Datenkategorien) bestimmt und in Beziehung zueinander gebracht werden. Das Festlegen von Datenklassen, im Fachjargon Entitätsmengen genannt, und das Bestimmen von Beziehungsmengen geschieht vorläufig noch unabhängig davon, mit welchem Datenbanksystem (SQL oder NoSQL) die Informationen später erfasst, gespeichert und nachgeführt werden. Damit möchte man erreichen, dass Daten und Datenbeziehungen bei der Entwicklung oder Erweiterung von Informationssystemen *von Seite der Anwendenden aus gesehen stabil* bleiben.

Zur Beschreibung eines Ausschnittes aus der realen Welt bis hin zur Festlegung der eigentlichen Datenbank sind drei wesentliche Schritte notwendig, und zwar die Datenanalyse, der Entwurf eines konzeptionellen Datenmodells (hier Entitäten-Beziehungsmodells) und dessen Überführung in ein relationales oder nicht-relationales Datenbankschema.

Bei der Datenanalyse (siehe unter 1 in Abb. 2.1) geht es darum, zusammen mit dem Benutzer die für das Informationssystem notwendigen Daten und deren Beziehungen samt Mengengerüst zu ermitteln. Nur so lassen sich überhaupt die Systemgrenzen frühzeitig festlegen. Mit Hilfe von Interviews, Bedarfsanalysen, Fragebogenaktionen, Formularsammlungen etc. muss durch ein iteratives Vorgehen eine aussagekräftige *Dokumentation zu den Anforderungen* (engl. *requirement analysis*) zusammengestellt werden. Diese umfasst mindestens sowohl eine verbale Beschreibung des Auftrages mit einer klaren

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_2

Zielsetzung als auch eine *Liste der relevanten Informationssachverhalte* (vgl. Beispiel in Abb. 2.1). Die verbale Beschreibung der Datenzusammenhänge kann bei der Datenanalyse ergänzt werden durch grafische Darstellungen oder durch ein zusammenfassendes Beispiel. Wichtig ist, dass die Datenanalyse die für den späteren Aufbau einer Datenbank notwendigen Fakten in der Sprache der Anwendenden formuliert.

Im nächsten Abstraktionsschritt 2 in Abb. 2.1 wird das sogenannte *Entitäten-Beziehungsmodell* (engl. *entity relationship model*) entworfen, das neben den Entitätsmengen auch die dazugehörigen Beziehungsmengen angibt. Dieses Modell stellt die Entitätsmengen grafisch durch Rechtecke und die Beziehungsmengen durch Rhomben dar. Aufgrund der Datenanalyse aus Abb. 2.1 erhält man ABTEILUNG, MITARBEITER und PROJEKT[1] als wesentliche Entitätsmengen. Um festzuhalten, in welchen Abteilungen die Mitarbeitenden tätig sind und an welchen Projekten sie arbeiten, werden die beiden Beziehungsmengen UNTERSTELLUNG und ZUGEHÖRIGKEIT definiert und grafisch mit den entsprechenden Entitätsmengen verknüpft. Das Entitäten-Beziehungsmodell erlaubt somit, die in der Datenanalyse zusammengestellten Fakten zu strukturieren und anschaulich darzustellen. Dabei darf nicht verschwiegen werden, dass das Erkennen von Entitäts- und Beziehungsmengen sowie das Festlegen der zugehörigen Merkmale nicht immer einfach und eindeutig erfolgt. Vielmehr verlangt dieser Entwurfsschritt von den Datenarchitekten einige Übung und praktische Erfahrung.

Das Entitäten-Beziehungsmodell wird nun unter 3a in Abb. 2.1 in ein *relationales Datenbankschema* (engl. *relational database schema*) oder unter 3b in ein *graphenorientiertes Datenbankschema* (engl. *graph-oriented database schema*) überführt. Unter einem Datenbankschema versteht man die formale Beschreibung der Datenbankobjekte; entweder mit der Hilfe von Tabellen oder Knoten und Kanten.

Da ein relationales Datenbanksystem als Objekte nur Tabellen zulässt, müssen *sowohl die Entitätsmengen als auch die Beziehungsmengen in Tabellenform* ausgedrückt werden. In Abb. 2.1 ergibt sich deshalb für die Entitätsmengen ABTEILUNG, MITARBEITER und PROJEKT je eine Entitätsmengentabelle in 3a. Um die Beziehungen ebenfalls tabellarisch darstellen zu können, definiert man für jede Beziehungsmenge eine eigenständige Tabelle. In unserem Beispiel 3a führt dies zu den beiden Tabellen UNTERSTELLUNG und ZUGEHÖRIGKEIT. Solche Beziehungsmengentabellen enthalten immer die Schlüssel der in die Beziehung eingehenden Entitätsmengen als Fremdschlüssel und allenfalls weitere Beziehungsmerkmale.

Die Beschreibung einer Graphdatenbank erfolgt in Abb. 2.1 unter 3b. Jede Entitätsmenge entspricht einem Knoten im Graphen, d. h. wir erhalten die Knoten ABTEILUNG, MITARBEITER und PROJEKT. Die Beziehungsmengen UNTERSTELLUNG und ZUGEHÖRIGKEIT werden aus dem Entitäten-Beziehungsmodell in Kanten im Graphenmodell überführt. Die Beziehungsmenge UNTERSTELLUNG wird zur gerichteten Kante

[1] In Analogie zu den Tabellennamen resp. Knoten- und Kantennamen verwenden wir für die Namen von Entitäts- und Beziehungsmengen ebenfalls Großbuchstaben.

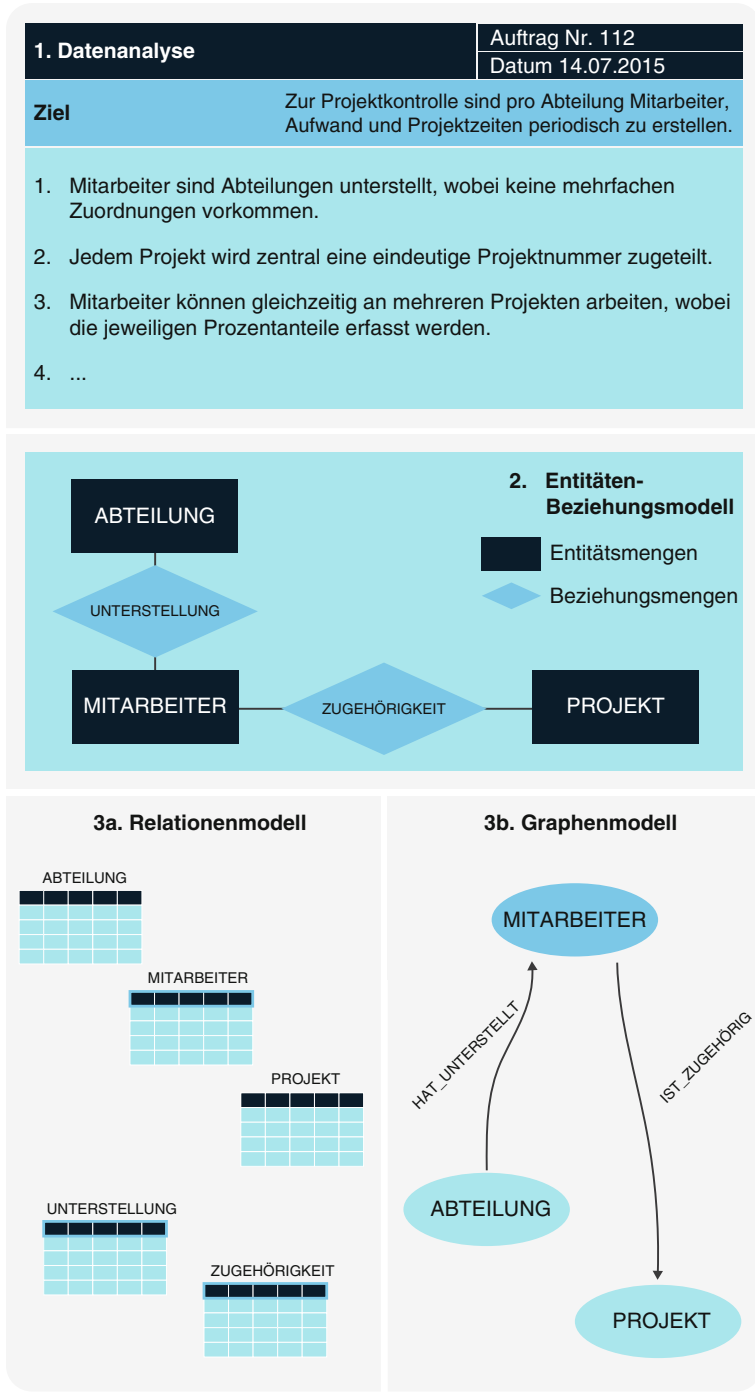

Abb. 2.1 Die drei notwendigen Schritte bei der Datenmodellierung

vom Knoten ABTEILUNG zum Knoten MITARBEITER und bekommt den Kantennamen HAT_UNTERSTELLT. Entsprechend wird eine gerichtete Kante vom Mitarbeiterknoten zum Projektknoten mit der Benennung IST_ZUGEHÖRIG gezogen.

Das Durchführen einer Datenanalyse, das Entwickeln eines Entitäten-Beziehungsmodells und das Definieren eines relationalen Datenbankschemas resp. eines graphenorientierten Datenmodells sind hier nur grob umrissen. Wesentlich ist die Erkenntnis, dass ein Datenbankentwurf sinnvollerweise anhand eines Entitäten-Beziehungsmodells entwickelt werden sollte. Dies erlaubt, losgelöst von einem bestimmten Datenbanksystem, mit den Anwendenden Datenmodellierungsaspekte festzuhalten und zu diskutieren. Erst in einem weiteren Entwurfsschritt wird das zweckmäßige Datenbankschema definiert, wobei für relationale wie graphenorientierte Datenbanken klare Abbildungsregeln existieren (vgl. Abschn. 2.3.2 resp. 2.4.2).

Im Folgenden wird in Abschn. 2.2 das Entitäten-Beziehungsmodell vorgestellt inkl. der Diskussion von Generalisation und Aggregation. Abschn. 2.3 widmet sich Modellierungsaspekten für relationale, Abschn. 2.4 für graphenorientierte Datenbanken. Hier werden jeweils die Abbildungsregeln für Entitäts- und Beziehungsmengen resp. Generalisation und Aggregation sowohl für das Relationenmodell wie auch für das Graphenmodell erläutert. Abschnitt 2.5 zeigt die Notwendigkeit auf, eine unternehmensweite Datenarchitektur für eine Organisation zu entwickeln. Abschnitt 2.6 liefert ein Rezept für die Analyse-, Modellierungs- und Datenbankschritte. Bibliographische Angaben finden sich in Abschn. 2.7.

2.2 Das Entitäten-Beziehungsmodell

2.2.1 Entitäten und Beziehungen

Unter *Entität* (engl. *entity*) versteht man ein bestimmtes, d. h. von anderen unterscheidbares Objekt der realen Welt oder unserer Vorstellung. Dabei kann es sich um ein Individuum, um einen Gegenstand, um einen abstrakten Begriff oder um ein Ereignis handeln. Entitäten des gleichen Typs werden zu *Entitätsmengen* zusammengefasst und durch Merkmale weiter charakterisiert. Solche sind Eigenschaftskategorien der Entität bzw. der Entitätsmenge wie z. B. Größe, Bezeichnung, Gewicht.

Für jede Entitätsmenge ist ein Identifikationsschlüssel, d. h. ein Merkmal oder eine Merkmalskombination zu bestimmen, der die Entitäten innerhalb der Entitätsmenge eindeutig festlegt. Neben der Forderung der Eindeutigkeit gilt auch die Forderung nach minimaler Merkmalskombination, wie wir sie in Abschn. 1.2.1 für Identifikationsschlüssel diskutiert haben.

Die Abb. 2.2 charakterisiert einen bestimmten Mitarbeitenden durch seine konkreten Merkmale als Entität. Möchte man für die betriebsinterne Projektkontrolle sämtliche Mitarbeitende mit ihren Namensangaben und Adressdaten erfassen, so legt man eine Entitätsmenge MITARBEITER fest. Neben den Merkmalen Name, Straße und Ort erlaubt

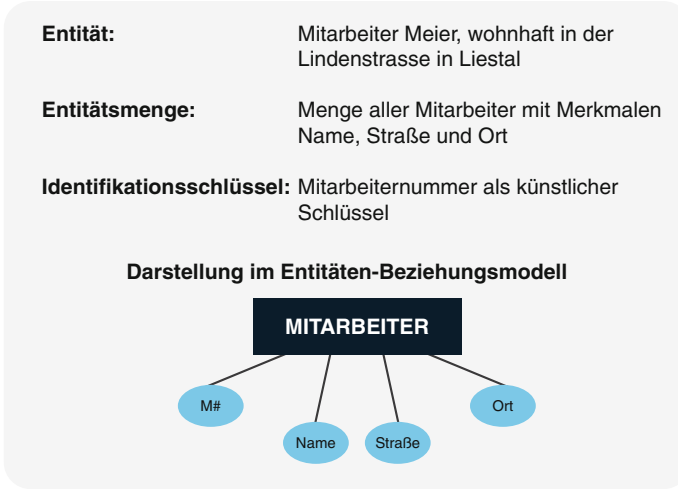

Abb. 2.2 Entitätsmenge MITARBEITER

eine künstliche Mitarbeiternummer, die einzelnen Mitarbeitenden (Entitäten) innerhalb der Belegschaft (Entitätsmenge) eindeutig zu identifizieren.

Neben den Entitätsmengen selbst sind *Beziehungen* (engl. *relationships*) zwischen ihnen von Bedeutung. Diese bilden wiederum eine Menge. Beziehungsmengen können, ebenso wie Entitätsmengen, durch eigene Merkmale näher charakterisiert werden.

In Abb. 2.3 wird die Aussage «Mitarbeiterin Meier arbeitet zu 70 % am Projekt P17» als konkretes Beispiel einer Mitarbeiter-Projektbeziehung verstanden. Die entsprechende Beziehungsmenge ZUGEHÖRIGKEIT soll sämtliche Projektzugehörigkeiten unter den Mitarbeitenden aufzählen. Sie enthält als zusammengesetzten Schlüssel die Fremdschlüssel Mitarbeiter- und Projektnummer. Mit dieser Merkmalskombination lässt sich jede Mitarbeiter-Projektzugehörigkeit eindeutig festhalten. Neben diesem zusammengesetzten Schlüssel wird ein eigenständiges Beziehungsmerkmal mit der Bezeichnung «%-Anteil» beigefügt. Dieses nennt den prozentualen Anteil der Arbeitszeit, die Projektmitarbeitende ihren zugeteilten Projekten widmen.

Im Allgemeinen lassen sich Beziehungen immer in zwei Richtungen als sogenannte Assoziationen deuten. Die Beziehungsmenge ZUGEHÖRIGKEIT kann aus der Sicht der Entitätsmenge MITARBEITER wie folgt interpretiert werden: Ein Mitarbeiter kann an mehreren Projekten mitwirken. Von der Entitätsmenge PROJEKT aus betrachtet lässt sich die Beziehung so verstehen, dass ein Projekt von mehreren Mitarbeitenden bearbeitet wird.

2.2.2 Assoziationstypen

Unter *Assoziation* (engl. *association*) einer Entitätsmenge EM_1 nach einer zweiten Entitätsmenge EM_2 versteht man die Bedeutung der Beziehung in dieser Richtung.

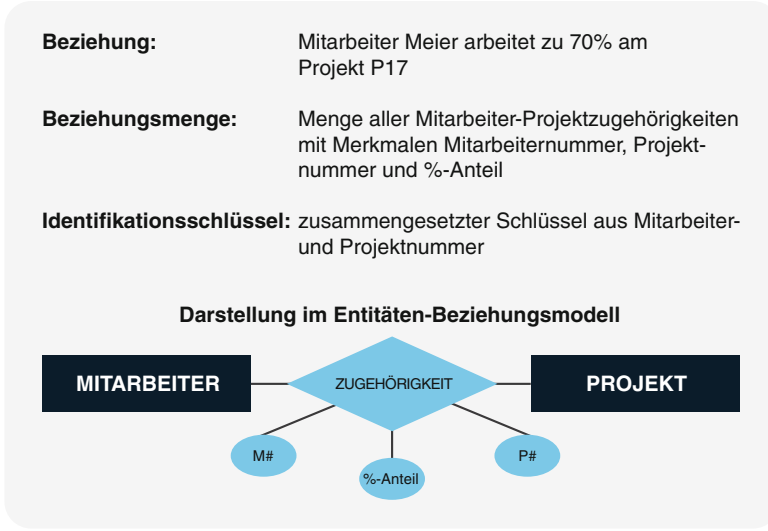

Abb. 2.3 Beziehung ZUGEHÖRIGKEIT zwischen Mitarbeitenden und Projekten

Betrachten wir dazu die Beziehung ABTEILUNGSLEITER in Abb. 2.4, so erkennen wir für diese Beziehungsmenge zwei Assoziationen. Einerseits gehört zu jeder Abteilung ein Mitarbeitender in der Funktion der Abteilungsleiterin oder des Abteilungsleiters, andererseits können gewisse Mitarbeitende die Funktion des Abteilungsleiters für eine bestimmte Abteilung ausüben.

Jede Assoziation einer Entitätsmenge EM_1 nach einer Entitätsmenge EM_2 kann mit einem Assoziationstyp gewichtet werden. Der Assoziationstyp von EM_1 nach EM_2 gibt an, wie viele Entitäten aus der assoziierten Entitätsmenge EM_2 einer bestimmten Entität aus EM_1 zugeordnet werden können.[2] Im Wesentlichen werden einfache, konditionelle, mehrfache und mehrfach-konditionelle Assoziationstypen unterschieden.

Einfache Assoziation (Typ 1)
Bei einer einfachen Assoziation oder einer Assoziation vom Typ 1 ist jeder Entität aus der Entitätsmenge EM_1 *«genau eine»* (engl. *unique*) Entität aus der Entitätsmenge EM_2 zugeordnet. Beispielsweise ist aufgrund unserer Datenanalyse jeder Mitarbeitende genau einer Abteilung unterstellt; es wird also keine «Matrixorganisation» zugelassen. Die Assoziation UNTERSTELLUNG aus Abb. 2.4 von Mitarbeitenden zu Abteilungen ist somit einfach oder vom Typ 1.

[2] Es entspricht der Konvention in der Datenbankliteratur, dass der Assoziationstyp von EM_1 nach EM_2 in der Nähe der assoziierten Entitätsmenge, hier also bei EM_2, annotiert wird.

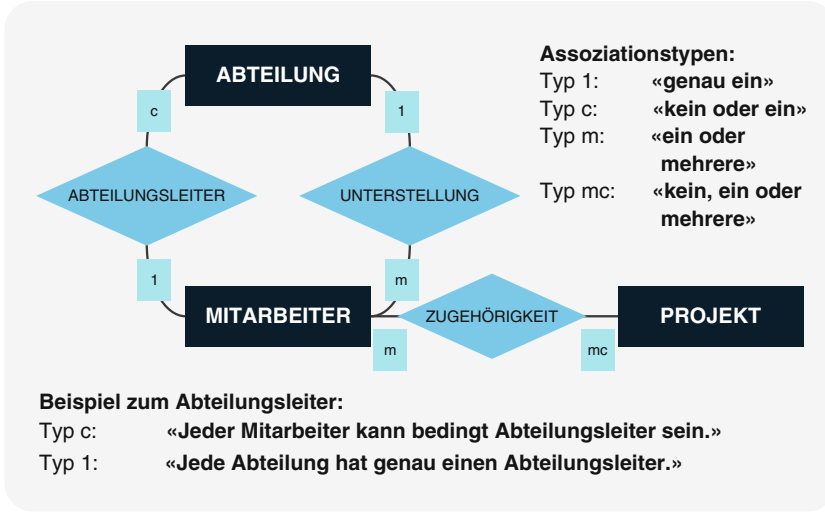

Abb. 2.4 Entitäten-Beziehungsmodell mit Assoziationstypen

Konditionelle Assoziation (Typ c)
Beim Typ c wird jeder Entität aus der Entitätsmenge EM_1 *«keine oder eine»*, also höchstens eine Entität aus der Entitätsmenge EM_2 zugeordnet. Der Assoziationstyp ist *bedingt* oder *konditionell* (engl. *conditional*). Eine bedingte Beziehung tritt beispielsweise beim ABTEILUNGSLEITER auf (vgl. Abb. 2.4), da nicht jeder Mitarbeitende die Funktion eines Abteilungsleiters ausüben kann.

Mehrfache Assoziation (Typ m)
Bei einer mehrfachen Assoziation oder einer Assoziation vom Typ m (engl. *multiple*) sind jeder Entität aus der Entitätsmenge EM_1 *«eine oder mehrere»* Entitäten in der Entitätsmenge EM_2 zugeordnet. Dieser und der nächste Assoziationstyp werden oft als *komplex* bezeichnet, da eine Entität aus EM_1 mit beliebig vielen in EM_2 in Beziehung stehen kann. Ein Beispiel einer mehrfachen Assoziation ist in Abb. 2.4 die Beziehung ZUGEHÖRIGKEIT von Projekten zu Mitarbeitenden: Jedes Projekt kann von mehreren, muss jedoch von mindestens einer Mitarbeiterin oder einem Mitarbeiter bearbeitet werden.

Mehrfach-konditionelle Assoziation (Typ mc)
Jeder Entität aus der Entitätsmenge EM_1 sind *«keine, eine oder mehrere»* Entitäten aus der Entitätsmenge EM_2 zugeordnet. Der *mehrfach-konditionelle Assoziationstyp* (engl. *multiple conditional*) hebt sich von dem mehrfachen dadurch ab, dass hier nicht zu jeder Entität aus EM_1 eine Beziehung zu denen von EM_2 bestehen muss. Als Beispiel können wir nochmals die ZUGEHÖRIGKEIT aus Abb. 2.4 betrachten, diesmal aus Sicht der Mitarbeitenden: Nicht jeder Mitarbeitende braucht Projektarbeit zu leisten; es gibt andererseits Mitarbeitende, die an mehreren Projekten mitwirken.

Bj := (A1, A2) Grad der Beziehungen mit Assoziationstypen A1 und A2

A1 \ A2	1	c	m	mc
1	(1,1)	(1,c)	(1,m)	(1,mc)
c	(c,1)	(c,c)	(c,m)	(c,mc)
m	(m,1)	(m,c)	(m,m)	(m,mc)
mc	(mc,1)	(mc,c)	(mc,m)	(mc,mc)

B1 umfasst die Felder (1,1), (1,c), (c,1), (c,c)
B2 umfasst die Felder (1,m), (c,m), (m,1), (m,c), (mc,1), (mc,c)
B3 umfasst die Felder (m,m), (m,mc), (mc,m), (mc,mc)

B1: einfach-einfache Beziehungen
B2: einfach-komplexe Beziehungen
B3: komplex-komplexe Beziehungen

Abb. 2.5 Übersicht über die Mächtigkeiten von Beziehungen

Die Assoziationstypen machen eine Aussage über die Mächtigkeit der Beziehung. Wie wir gesehen haben, umfasst jede Beziehung zwei Assoziationstypen. Die *Mächtigkeit einer Beziehung* zwischen den Entitätsmengen EM_1 und EM_2 ergibt sich somit durch ein *Paar von Assoziationstypen* der Form:

Mächtigkeit := (Typ von EM_1 nach EM_2, Typ von EM_2 nach EM_1).[3]

Beispielsweise sagt das Paar (mc,m) von Assoziationstypen zwischen MITARBEITER und PROJEKT aus, dass die Beziehung ZUGEHÖRIGKEIT (bedingt-komplex, komplex) ist.

In Abb. 2.5 sind alle sechzehn Kombinationsmöglichkeiten von Beziehungen aufgezeigt. Als Erstes gibt es vier Möglichkeiten von einfach-einfachen Beziehungen (Fall B1 in Abb. 2.5). Diese sind durch die Mächtigkeiten (1,1), (1,c), (c,1) und (c,c) charakterisiert. Bei den einfach-komplexen Beziehungen, oft *hierarchische Beziehungen* genannt, gibt es acht Optionen (Fall B2). Bei den komplex-komplexen oder *netzwerkartigen Beziehungen* treten die vier Fälle (m,m), (m,mc), (mc,m) und (mc,mc) auf (Fall B3).

Anstelle der Assoziationstypen können auch *Minimal- und Maximalschranken* angegeben werden, falls dies sinnvoll erscheint. So könnte man für den mehrfachen Assoziationstyp von Projekten zu Mitarbeitenden anstelle von «m» eine Angabe (MIN,MAX) := (3,8) festlegen. Die untere Schranke sagt aus, dass an einem Projekt definitionsgemäß mindestens drei

[3] Die Zeichenkombination «:=» bedeutet «ist definiert durch . . .».

Mitarbeitende beteiligt sein müssen. Umgekehrt verlangt die obere Schranke die Begrenzung eines Projektes auf höchstens acht Mitarbeitende.

2.2.3 Generalisation und Aggregation

Unter *Generalisation* (engl. *generalization*) versteht man das Verallgemeinern von Entitäten und somit auch von Entitätsmengen. Die Generalisation ist ein Abstraktionsvorgang, bei dem einzelne Entitätsmengen zu einer übergeordneten Entitätsmenge verallgemeinert werden. Umgekehrt lassen sich die in einer Generalisationshierarchie abhängigen Entitätsmengen oder Subentitätsmengen als *Spezialisierung* interpretieren. Bei der Generalisation von Entitätsmengen treten die folgenden Fälle auf:

- **Überlappende Subentitätsmengen**: Die Entitätsmengen der Spezialisierung *überlappen sich gegenseitig*. Betrachten wir als Beispiel die Entitätsmenge MITARBEITER mit zwei Subentitätsmengen FOTOCLUB und SPORTCLUB, so können wir die Clubmitglieder als Mitarbeitende auffassen. Umgekehrt können Mitarbeitende sowohl im firmeninternen Fotoclub wie auch im Sportclub aktiv mitwirken, d. h. die Subentitätsmengen FOTOCLUB und SPORTCLUB überlappen sich.
- **Überlappend-vollständige Subentitätsmengen**: Die Entitätsmengen der Spezialisierung *überlappen sich gegenseitig und überdecken vollständig* die verallgemeinerte Entitätsmenge der Generalisation. Ergänzen wir die beiden obigen Subentitätsmengen SPORTCLUB und FOTOCLUB durch einen SCHACHCLUB und nehmen wir an, dass jede Mitarbeiterin und jeder Mitarbeiter beim Eintritt in die Firma mindestens einem dieser drei Clubs beitritt, so liegt eine «überlappend-vollständige» Konstellation vor. Jeder Mitarbeitende ist also mindestens in einem der drei Clubs, kann aber auch in zwei oder in allen drei Clubs mitwirken.
- **Disjunkte Subentitätsmengen**: Die Entitätsmengen der Spezialisierung sind *gegenseitig disjunkt*, d. h. sie schließen sich gegenseitig aus. Als Beispiel betrachten wir die Entitätsmenge MITARBEITER mit den Spezialisierungen FÜHRUNGSKRAFT und FACHSPEZIALIST. Da Mitarbeitende nicht gleichzeitig eine Kaderposition bekleiden und eine Fachkarriere verfolgen können, fassen wir die beiden Spezialisierungen FÜHRUNGSKRAFT und FACHSPEZIALIST als gegenseitig disjunkt auf.
- **Disjunkt-vollständige Subentitätsmengen**: Die Entitätsmengen der Spezialisierung sind gegenseitig disjunkt und überdecken vollständig die verallgemeinerte Entitätsmenge der Generalisation. In diesem Fall muss also zu jeder Entität aus der übergeordneten Entitätsmenge eine Subentität in der Spezialisierung bestehen und umgekehrt. Als Beispiel nehmen wir wiederum die Entitätsmenge MITARBEITER, wobei wir neben den Spezialisierungen FÜHRUNGSKRAFT und FACHSPEZIALIST eine weitere Spezialisierung LEHRLING betrachten. Dabei soll gelten, dass jeder Mitarbeitende entweder als Führungskraft, Fachspezialistin oder Fachspezialist oder in Ausbildung tätig ist.

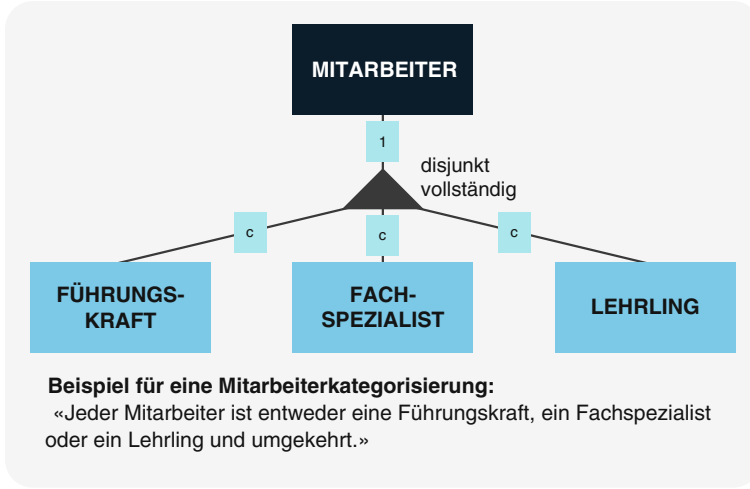

Abb. 2.6 Generalisation am Beispiel MITARBEITER

Jede Generalisationshierarchie wird durch ein spezielles grafisches Gabelungssymbol dargestellt, ergänzt durch die Charakterisierung «überlappend-unvollständig», «überlappend-vollständig», «disjunkt- unvollständig» oder «disjunkt-vollständig».

In Abb. 2.6 ist die Entitätsmenge MITARBEITER als disjunkte und vollständige Verallgemeinerung von FÜHRUNGSKRAFT, FACHSPEZIALIST und LEHRLING aufgeführt. Jede abhängige Entität wie Gruppenchefin oder Abteilungsleiter der Entitätsmenge FÜHRUNGSKRAFT ist vom Typ MITARBEITER, da der entsprechende Assoziationstyp 1 ist. Aus diesem Grund wird die Generalisation oft in Anlehnung ans Englische als *IS-A-Struktur* bezeichnet: Eine Gruppenchefin «ist eine» (engl. *is a*) Mitarbeitendende, und ein Abteilungsleiter ist ebenfalls als Mitarbeiter angestellt. Die umgekehrte Assoziation ist bei einer disjunkten und vollständigen Generalisationshierarchie vom Typ c; so gehört jeder Mitarbeitende einer der Subentitätsmengen an.

Neben der Generalisation spielt eine zweite Beziehungsstruktur eine wichtige Rolle. Unter dem Begriff *Aggregation* (engl. *aggregation*) versteht man das Zusammenfügen von Entitäten zu einem übergeordneten Ganzen. Dabei werden die Struktureigenschaften der Aggregation in einer Beziehungsmenge erfasst.

Soll für einen Konzern die Beteiligungsstruktur modelliert werden, wie in Abb. 2.7 dargestellt, so geschieht das mit einer Beziehungsmenge KONZERNSTRUKTUR. Diese beschreibt ein Beziehungsnetz der Entitätsmenge FIRMA auf sich selbst. Die Firmennummer aus der Entitätsmenge FIRMA wird zweimal in der KONZERNSTRUKTUR als Fremdschlüssel verwendet, einmal als Nummer der übergeordneten und einmal als Nummer der untergeordneten Firmengesellschaften (vgl. Abb. 2.21 resp. 2.36). Daneben können in der KONZERNSTRUKTUR weitere Beziehungsmerkmale wie beispielsweise das der Beteiligung geführt werden.

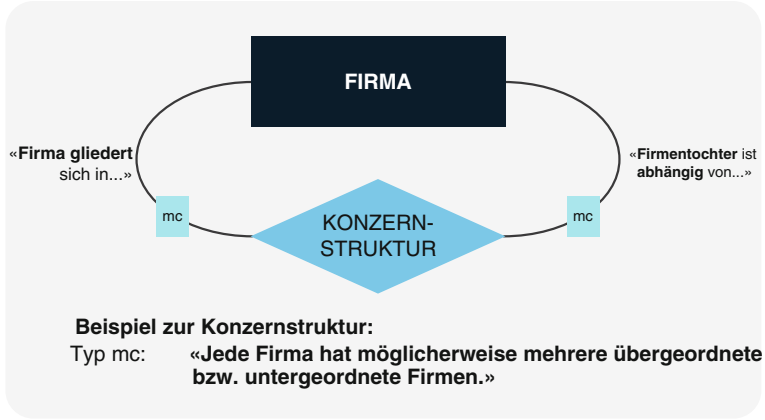

Abb. 2.7 Netzwerkartige Aggregation am Beispiel KONZERNSTRUKTUR

Allgemein beschreibt eine Aggregation das strukturierte Zusammenfügen von Entitäten, das wir in Anlehnung ans Englische als *PART-OF-Struktur* bezeichnen. So kann bei der KONZERNSTRUKTUR jede Firma eine «Tochter von» (engl. *part of*) einem bestimmten Konzern sein. Da in unserem Beispiel die KONZERNSTRUKTUR netzwerkartig definiert ist, müssen beide Assoziationstypen der übergeordneten wie der untergeordneten Teile bedingt-komplex sein.

Die beiden Abstraktionskonzepte Generalisation und Aggregation sind wichtige Strukturierungselemente[4] bei der Datenmodellierung. Im Entitäten-Beziehungsmodell können sie durch eigene grafische Symbole charakterisiert oder als Spezialkästchen ausgedrückt werden. So könnte man obige Aggregation aus Abb. 2.7 durch eine verallgemeinerte Entitätsmenge KONZERN darstellen, die implizit die Entitätsmenge FIRMA und die Beziehungsmenge KONZERNSTRUKTUR einschließen würde.

Neben netzwerkartigen PART-OF-Strukturen gibt es auch hierarchische. In Abb. 2.8 ist dazu eine STÜCKLISTE illustriert: Jeder Artikel kann aus mehreren Teilartikeln zusammengesetzt sein. Umgekehrt zeigt jeder Teilartikel mit Ausnahme des obersten Artikels genau auf einen übergeordneten Artikel (vgl. Abb. 2.22 resp. 2.37).

Beim Einsatz von rechnergestützten Werkzeugen zur Datenmodellierung spielt das Entitäten-Beziehungsmodell eine besondere Rolle, das von vielen CASE-Werkzeugen (CASE = Computer Aided Software Engineering) mehr oder weniger unterstützt wird. Je nach Güte dieser Werkzeuge können neben Entitätsmengen und Beziehungsmengen auch Generalisation und Aggregation in separaten Konstruktionsschritten beschrieben werden. Danach erst lässt sich das *Entitäten-Beziehungsmodell in ein Datenbankschema teilweise automatisch überführen*. Da dieser Abbildungsprozess nicht immer eindeutig ist, liegt es

[4] Objektorientierte oder objektrelationale Datenbanksysteme unterstützen Generalisation und Aggregation als Strukturierungskonzepte (vgl. Kap. 6).

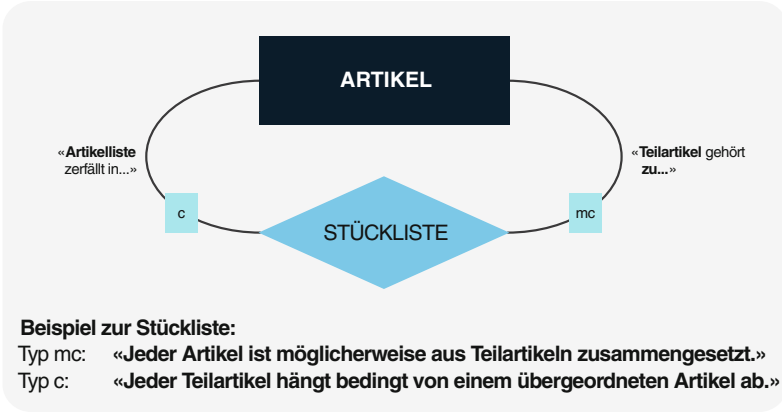

Abb. 2.8 Hierarchische Aggregation am Beispiel STÜCKLISTE

am Datenarchitekten, die richtigen Entscheidungen zu treffen. Aus diesem Grund werden in den Abschn. 2.3.2 resp. 2.4.2 einfache Abbildungsregeln erläutert, die ein Entitäten-Beziehungsmodell konfliktfrei auf eine relationale Datenbank resp. Graphdatenbank abbilden.

2.3 Umsetzung im Relationenmodell

2.3.1 Abhängigkeiten und Normalformen

Die Untersuchung des Relationenmodells hat eine eigentliche Datenbanktheorie hervorgebracht, bei der die formalen Aspekte präzise beschrieben werden. Ein bedeutendes Teilgebiet dieser Datenbanktheorie bilden die sogenannten *Normalformen* (engl. *normal forms*). Mit diesen werden innerhalb von Tabellen Abhängigkeiten studiert und aufgezeigt, oft zur Vermeidung redundanter Information und damit zusammenhängender Anomalien.

Zur Redundanz eines Merkmals
Ein Merkmal einer Tabelle ist redundant, wenn einzelne Werte dieses Merkmals innerhalb der Tabelle *ohne Informationsverlust weggelassen* werden können.

Betrachten wir dazu als Beispiel eine Tabelle ABTEILUNGSMITARBEITER, die neben Mitarbeiternummer, Name, Straße und Ort für jeden Mitarbeitenden noch seine oder ihre Abteilungsnummer samt Abteilungsbezeichnung enthält.

In Abb. 2.9 ist für jeden Mitarbeitenden aus der Abteilung A6 der Abteilungsname Finanz aufgelistet. Entsprechendes gilt für die anderen Abteilungen, da unserer Annahme gemäß in einer Abteilung mehrere Mitarbeitende tätig sind. Das Merkmal Bezeichnung ist also redundant, da mehrmals ein und dieselbe Abteilungsbezeichnung in der Tabelle

ABTEILUNGSMITARBEITER

M#	Name	Strasse	Ort	A#	Bezeichnung
M19	Schweizer	Hauptstrasse	Frenkendorf	A6	Finanz
M1	Meier	Lindenstrasse	Liestal	A3	Informatik
M7	Huber	Mattenweg	Basel	A5	Personal
M4	Becker	Wasserweg	Liestal	A6	Finanz

Abb. 2.9 Redundante und anomalienträchtige Tabelle

vorkommt. Es würde genügen, sich die Bezeichnung der Abteilungen in einer separaten Tabelle ein für allemal zu merken, anstatt diesen Sachverhalt für jeden Mitarbeitenden redundant mitzuführen.

Bei Tabellen mit redundanter Information können sogenannte *Mutationsanomalien* auftreten. Möchten wir aus organisatorischen Überlegungen in der Tabelle ABTEILUNGSMITARBEITER von Abb. 2.9 eine neue Abteilung A9 mit der Bezeichnung Marketing definieren, so können wir diesen Sachverhalt nicht ohne Weiteres in unserer Tabelle festhalten. Es liegt eine *Einfügeanomalie* vor, weil wir keine neue Tabellenzeile ohne eine eindeutige Mitarbeiternummer einbringen können.

Von einer Löschanomalie spricht man, wenn ein Sachverhalt ungewollt verloren geht. Falls wir in unserer Tabelle ABTEILUNGSMITARBEITER sämtliche Mitarbeitenden löschen, verlieren wir gleichzeitig die Abteilungsnummern mit ihren Bezeichnungen.

Schließlich gibt es auch *Änderungsanomalien*: Soll die Bezeichnung der Abteilung A3 von Informatik auf DV-Abteilung abgeändert werden, so muss bei sämtlichen Mitarbeitenden der Informatikabteilung dieser Namenswechsel vollzogen werden. Anders ausgedrückt: Obwohl nur ein einziger Sachverhalt eine Veränderung erfährt, muss die Tabelle ABTEILUNGSMITARBEITER an verschiedenen Stellen angepasst werden. Dieser Nachteil wird als Änderungsanomalie bezeichnet.

In den folgenden Ausführungen werden Normalformen diskutiert, die Redundanzen und Anomalien vermeiden helfen. Die Abb. 2.10 gibt eine Übersicht über die Normalformen und ihre Bedeutungen. Sie illustriert insbesondere, dass mit Hilfe der Normalformen verschiedene Abhängigkeiten zu Konfliktsituationen führen können. Diese Abhängigkeiten werden im Folgenden mit Beispielen eingehender studiert.

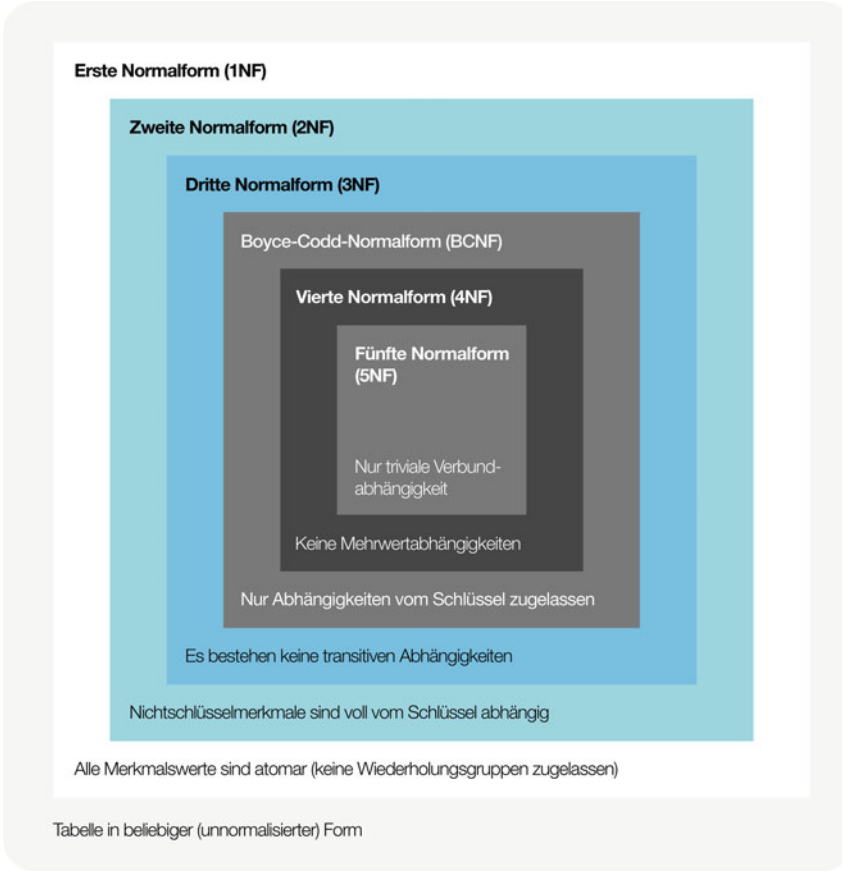

Abb. 2.10 Übersicht über die Normalformen und ihre Charakterisierung

Wie Abb. 2.10 zeigt, schränken die Normalformen die Menge der zugelassenen Tabellen mehr und mehr ein. So muss beispielsweise eine Tabelle oder ein ganzes Datenbankschema in dritter Normalform die erste und die zweite Normalform erfüllen, und zusätzlich dürfen keine sogenannten transitiven Abhängigkeiten unter den Nichtschlüsselmerkmalen auftreten.

Beim Studium der Normalformen ist wesentlich, dass nicht alle Normalformen dieselbe Bedeutung haben. *Meistens beschränkt man sich in der Praxis auf die ersten drei Normalformen*, da Mehrwert- oder Verbundabhängigkeiten kaum in Erscheinung treten; entsprechende Anwendungsbeispiele sind selten. Die vierte und die fünfte Normalform werden deshalb nur kurz diskutiert.

Das Verständnis für die Normalformen hilft, die Abbildungsregeln von einem Entitäten-Beziehungsmodell zu einem Relationenmodell zu untermauern (siehe Abschn. 2.3.2). Tatsächlich bleiben, wie wir sehen werden, bei einer sauberen Definition eines Entitä-

ten-Beziehungsmodells und einer konsequenten Anwendung der entsprechenden Abbildungsregeln die Normalformen jederzeit erfüllt. Mit anderen Worten, wir können *auf das Prüfen der Normalformen bei jedem einzelnen Entwurfsschritt weitgehend verzichten*, falls ein Entitäten-Beziehungsmodell erstellt und anhand von Abbildungsregeln auf das relationale Datenbankschema abgebildet wird.

Funktionale Abhängigkeiten

Die erste Normalform bildet die Ausgangsbasis für die übrigen Normalformen und lautet wie folgt:

Erste Normalform (1NF)
Eine Tabelle ist in erster Normalform, falls die *Wertebereiche der Merkmale* atomar sind. Die erste Normalform verlangt, dass jedes Merkmal Werte aus einem unstrukturierten Wertebereich bezieht. Somit dürfen keine Mengen, Aufzählungen oder Wiederholungsgruppen in den einzelnen Merkmalen vorkommen.

Die Tabelle PROJEKTMITARBEITER aus Abb. 2.11 ist vorerst nicht normalisiert, enthält sie doch pro Mitarbeitertupel mehrere Nummern von Projekten, welche vom jeweiligen Mitarbeitenden zu bearbeiten sind. Die unnormalisierte Tabelle lässt sich auf einfache Art in die erste Normalform bringen, indem für jedes Projektengagement ein eigenes Tupel erzeugt wird. Dabei fällt auf, dass der Schlüssel der Tabelle PROJEKTMITARBEITER beim Überführen in die erste Normalform erweitert werden muss, da wir für das eindeutige Identifizieren der Tupel sowohl die Mitarbeiter- als auch die Projektnummer benötigen. Es ist üblich (aber nicht zwingend), bei zusammengesetzten Schlüsseln beide Schlüsselteile am Tabellenanfang direkt hintereinander zu zeigen.

Paradoxerweise erhalten wir bei der Anwendung der ersten Normalform eine Tabelle mit Redundanz. In Abb. 2.11 sind sowohl Namen wie Adressangaben der Mitarbeitenden redundant, da sie bei jedem Projektengagement wiederholt werden. Die zweite Normalform schafft hier Abhilfe:

Zweite Normalform (2NF)
Eine Tabelle ist in zweiter Normalform, wenn zusätzlich zu den Bedingungen der ersten Normalform jedes Nichtschlüsselmerkmal von jedem Schlüssel *voll funktional abhängig* bleibt.

Ein Merkmal B ist *funktional abhängig* vom Merkmal A, falls zu jedem Wert von A genau ein Wert aus B existiert (abgekürzt durch die Schreibweise A->B). Die *funktionale Abhängigkeit* (engl. *functional dependency*) B von A verlangt somit, dass jeder Wert von A auf eindeutige Art einen Wert von B bestimmt. Bekanntlich haben alle Identifikationsschlüssel die Eigenschaft, dass die Nichtschlüsselmerkmale eindeutig vom Schlüssel abhängig sind. Es gilt also allgemein für einen Identifikationsschlüssel S und für ein beliebiges Merkmal B einer bestimmten Tabelle die funktionale Abhängigkeit S->B.

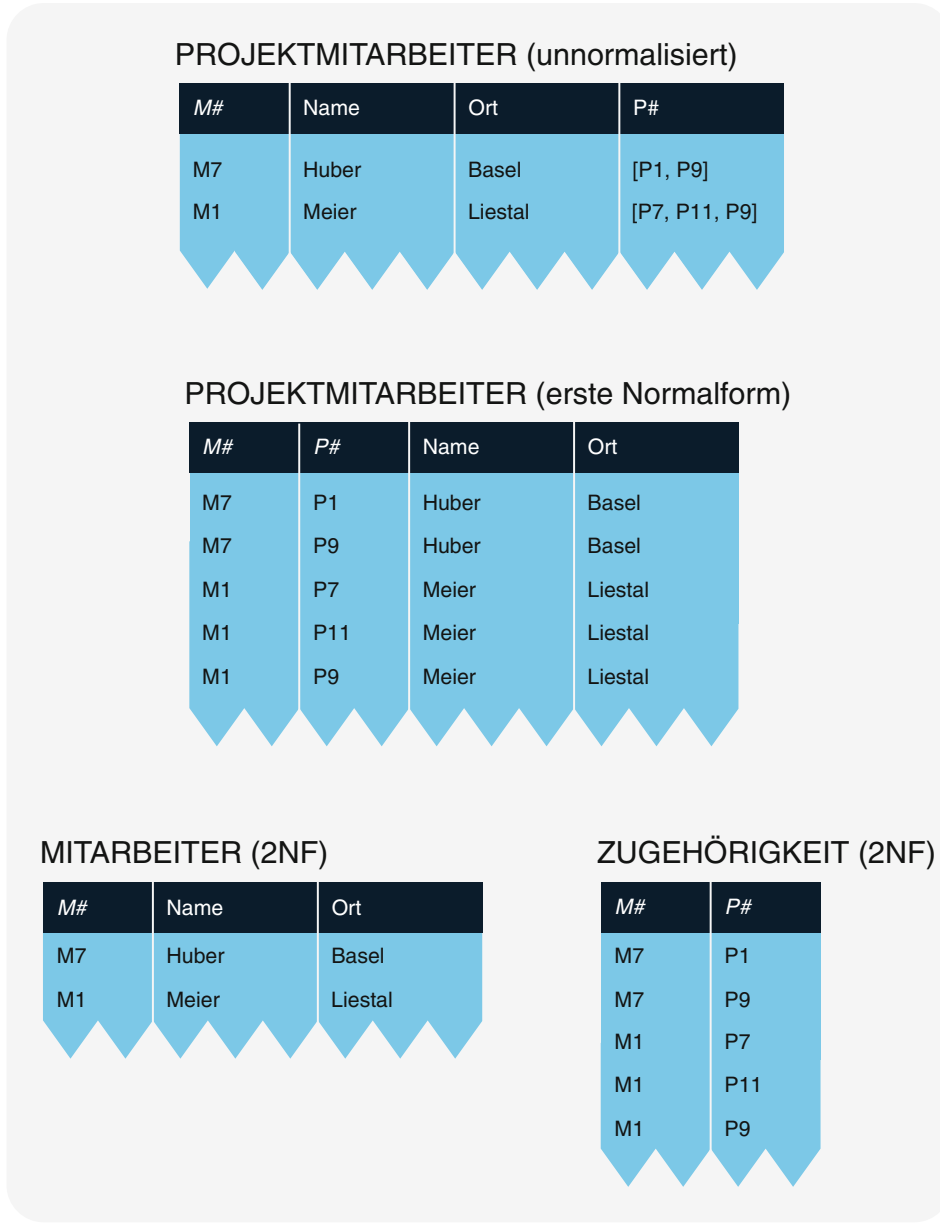

Abb. 2.11 Tabellen in erster und zweiter Normalform

Bei zusammengesetzten Schlüsseln müssen wir den Begriff der funktionalen Abhängigkeit zum Begriff der vollen funktionalen Abhängigkeit wie folgt erweitern: Ein Merkmal B ist *voll funktional abhängig* von einem aus S1 und S2 zusammengesetzten Schlüssel (abgekürzt durch die Schreibweise (S1,S2)=>B), falls B funktional abhängig vom Gesamtschlüssel, nicht jedoch von seinen Teilschlüsseln ist. Volle funktionale Abhängigkeit besteht also dann, wenn der zusammengesetzte Schlüssel allein in seiner Gesamtheit die übrigen Nichtschlüsselattribute eindeutig bestimmt. Es muss die funktionale Abhängigkeit (S1,S2)->B gelten, hingegen dürfen weder S1->B noch S2->B auftreten. Die *volle funktionale Abhängigkeit* (engl. *full functional dependency*) eines Merkmals vom Schlüssel verbietet also, dass es von seinen Teilen funktional abhängig ist.

Wir betrachten die in die erste Normalform überführte Tabelle PROJEKTMITARBEITER in Abb. 2.11. Sie enthält den zusammengesetzten Schlüssel (M#,P#) und muss somit auf ihre volle funktionale Abhängigkeit hin überprüft werden. Für den Namen sowie den Wohnort der Projektmitarbeitenden gelten die funktionalen Abhängigkeiten (M#,P#)-> Name sowie (M#,P#)->Ort. Jede Kombination der Mitarbeiternummer mit der Projektnummer bestimmt eindeutig einen Mitarbeiternamen oder einen Ort. Hingegen wissen wir, dass sowohl der Name als auch der Wohnort der Mitarbeitenden nichts mit den Projektnummern zu tun haben. Also sind die beiden Merkmale Name und Ort von einem Teil des Schlüssels funktional abhängig, d. h. es gilt M#->Name und M#->Ort. Dies ist ein Widerspruch zur Definition der vollen funktionalen Abhängigkeit. Somit ist die Tabelle PROJEKTMITARBEITER nicht in zweiter Normalform.

Ist eine Tabelle mit einem zusammengesetzten Schlüssel nicht in zweiter Normalform, so muss sie in Teiltabellen zerlegt werden. Dabei fasst man die Merkmale, die von einem Teilschlüssel abhängig sind, und diesen Teilschlüssel in einer eigenständigen Tabelle zusammen. Die Resttabelle mit dem zusammengesetzten Schlüssel und eventuell weiteren Beziehungsattributen belässt man als übrigbleibende Tabelle.

Im Beispiel aus Abb. 2.11 erhält man die beiden Tabellen MITARBEITER und ZUGEHÖRIGKEIT. Beide Tabellen sind in erster und zweiter Normalform. In der Tabelle MITARBEITER tritt kein zusammengesetzter Schlüssel mehr auf, und die zweite Normalform ist offensichtlich erfüllt. Die Tabelle ZUGEHÖRIGKEIT hat keine Nichtschlüsselattribute, so dass sich eine Überprüfung der zweiten Normalform auch hier erübrigt.

Transitive Abhängigkeiten

In Abb. 2.12 zeigen wir unsere anfangs diskutierte Tabelle ABTEILUNGSMITARBEITER, die neben Angaben zu den Mitarbeitenden auch Abteilungsinformationen enthält. Wir erkennen sofort, dass sich die Tabelle in erster und zweiter Normalform befindet. Da kein zusammengesetzter Schlüssel auftritt, müssen wir die Eigenschaft der vollen funktionalen Abhängigkeit nicht überprüfen. Doch offensichtlich tritt das Merkmal Bezeichnung redundant auf. Dieser Missstand ist mit der dritten Normalform zu beheben:

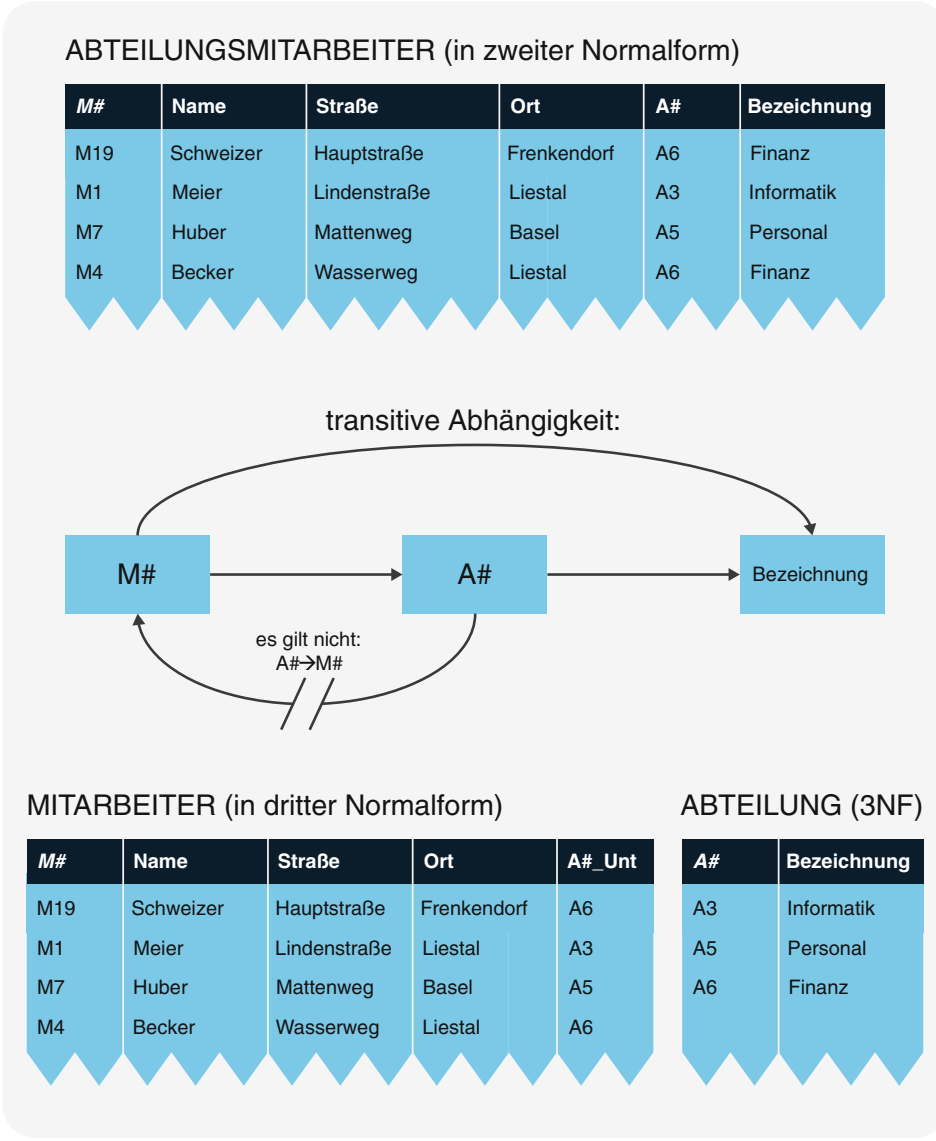

Abb. 2.12 Transitive Abhängigkeit und dritte Normalform

Dritte Normalform (3NF)
Eine Tabelle ist in dritter Normalform, wenn zusätzlich zur zweiten Normalform *kein Nichtschlüsselmerkmal von irgendeinem Schlüssel transitiv abhängig* ist.

Einmal mehr definieren wir eine Normalform durch ein Abhängigkeitskriterium: Transitiv abhängig bedeutet, *über Umwege funktional abhängig* zu sein. Beispielsweise ist unser

Merkmal Bezeichnung über den Umweg Abteilungsnummer von der Mitarbeiternummer funktional abhängig. Wir erkennen zwischen der Mitarbeiternummer und der Abteilungsnummer sowie zwischen Abteilungsnummer und Bezeichnung eine funktionale Abhängigkeit. Die beiden funktionalen Abhängigkeiten, M#->A# und A#->Bezeichnung, lassen sich über die Abteilungsnummer zur transitiven Abhängigkeit M#->Bezeichnung zusammenfügen.

Allgemein ist bei zwei funktionalen Abhängigkeiten A->B und B->C mit einem gemeinsamen Merkmal B die zusammengesetzte Abhängigkeit A->C ebenfalls funktional. Falls A die Werte in B eindeutig bestimmt und B diejenigen in C, so vererbt sich diese Eigenschaft von A auf C. Die Abhängigkeit A->C ist somit sicher funktional. Zusätzlich wird sie transitiv genannt, wenn neben den beiden funktionalen Abhängigkeiten A->B und B->C nicht gleichzeitig A funktional von B abhängig ist. Somit gilt für die *transitive Abhängigkeit* (engl. *transitive dependency*) die Definition: Das Merkmal C ist transitiv abhängig von A, falls B funktional abhängig von A, C funktional abhängig von B und nicht gleichzeitig A funktional abhängig von B ist.

Aus dem Beispiel ABTEILUNGSMITARBEITER in Abb. 2.12 geht hervor, dass das Merkmal Bezeichnung transitiv vom Merkmal M# abhängig ist. Deshalb ist definitionsgemäß die Tabelle ABTEILUNGSMITARBEITER nicht in dritter Normalform. Durch Zerlegung befreien wir sie von der transitiven Abhängigkeit, indem wir das redundante Merkmal Bezeichnung zusammen mit der Abteilungsnummer als eigenständige Tabelle ABTEILUNG führen. Die Abteilungsnummer bleibt als Fremdschlüssel mit dem Rollennamen Unterstellung (siehe Merkmal «A#_Unterstellung») in der Resttabelle MITARBEITER. Die Beziehung zwischen Mitarbeitenden und Abteilung ist dadurch nach wie vor gewährleistet.

Mehrwertige Abhängigkeiten

Mit der zweiten und dritten Normalform gelingt es, Redundanzen bei den Nichtschlüsselattributen zu eliminieren. Das Aufspüren redundanter Informationen darf aber nicht bei den Nichtschlüsselmerkmalen stehen bleiben, da auch zusammengesetzte Schlüssel redundant auftreten können.

Eine allenfalls erforderliche Erweiterung der dritten Normalform wird aufgrund der Arbeiten von Boyce und Codd als «Boyce-Codd-Normalform» oder BCNF bezeichnet. Eine solche kommt zum Tragen, wenn mehrere Schlüsselkandidaten in ein und derselben Tabelle auftreten. So existieren Tabellen mit sich gegenseitig überlappenden Schlüsseln, die zwar in dritter Normalform sind, aber die Boyce-Codd-Normalform verletzen. In einem solchen Fall muss die Tabelle aufgrund der Schlüsselkandidaten zerlegt werden. Ein Hinweis auf die weiterführende Literatur in Abschn. 2.7 soll genügen.

Eine weitere Normalform ergibt sich beim Studium von *mehrwertigen Abhängigkeiten* (engl. *multi-valued dependency*) zwischen einzelnen Merkmalen des Schlüssels. Obwohl in der Praxis mehrwertige Abhängigkeiten eine untergeordnete Rolle spielen, zeigt Abb. 2.13 diese Normalform kurz an einem einfachen Beispiel. Die Ausgangstabelle METHODE liegt als unnormalisierte Tabelle vor, da neben der Angabe der Methode eventuell mehrere Autoren und mehrere Begriffe auftreten. So enthält die Methode Struktogramm die Autoren

Nassi und Shneiderman. Gleichzeitig ist sie durch die drei Sprachelemente Sequenz, Iteration und Verzweigung ebenfalls mit mehreren Wertangaben charakterisiert.

Wir überführen die unnormalisierte Tabelle in die erste Normalform, indem die Mengen {Nassi, Shneiderman} und {Sequenz, Iteration, Verzweigung} aufgelöst werden. Dabei fällt auf, dass die neue Tabelle aus lauter Schlüsselmerkmalen besteht. Diese Tabelle ist nicht nur in erster Normalform, sondern auch in zweiter und in dritter. Hingegen zeigt die Tabelle redundante Informationen, obwohl sie in dritter Normalform ist. Beispielsweise merken wir uns pro Autor des Struktogramms, dass die drei Sprachelemente oder Begriffe Sequenz, Iteration und Verzweigung Anwendung finden. Umgekehrt merken wir uns pro Begriff des Struktogramms die beiden Autoren Nassi und Shneiderman. Mit anderen Worten haben wir es hier mit paarweise mehrwertigen Abhängigkeiten zu tun, die eliminiert werden müssen.

Für eine Tabelle mit den Merkmalen A, B und C gilt die folgende Definition für mehrwertige Abhängigkeiten: Ein Merkmal C ist *mehrwertig abhängig* vom Merkmal A (ausgedrückt durch die Schreibweise A->->C), falls zu jeder Kombination eines bestimmten Wertes aus A mit einem beliebigen Wert aus B eine identische Menge von Werten aus C erscheint.

Auf unser Beispiel von Abb. 2.13 bezogen können wir folgern, dass das Merkmal «Begriff» mehrwertig abhängig vom Merkmal «Methode» ist; es gilt somit «Methode->->Begriff». Kombinieren wir das Struktogramm mit dem Autor Nassi, so erhalten wir dieselbe Menge {Sequenz, Iteration, Verzweigung} wie bei der Kombination des Struktogramms mit dem Autor Shneiderman. Zusätzlich gilt die mehrwertige Abhängigkeit zwischen den Merkmalen Autor und Methode in der Form «Methode->->Autor». Kombinieren wir hier eine bestimmte Methode wie Struktogramm mit einem beliebigen Begriff wie Sequenz, so erhalten wir die Autoren Nassi und Shneiderman. Dieselbe Autorenschaft tritt zu Tage durch die Kombination von Struktogramm mit den Begriffen Iteration oder Verzweigung.

Treten mehrwertige Abhängigkeiten in einer Tabelle auf, so können Redundanzen und Anomalien entstehen. Um diese zu tilgen, muss eine weitere Normalform berücksichtigt werden:

Vierte Normalform (4NF)
Die vierte Normalform lässt es nicht zu, dass in ein und derselben Tabelle *zwei echte und voneinander verschiedene mehrwertige Abhängigkeiten* vorliegen.

Auf unser Beispiel bezogen muss die Tabelle METHODE also in die zwei Teiltabellen METHODIKER und METHODIK aufgeteilt werden. Erstere verbindet Methoden mit Autoren, letztere Methoden mit Begriffen. Offensichtlich sind beide Tabellen redundanzfrei und in vierter Normalform.

Verbundabhängigkeit

Bei einer Zerlegung einer Tabelle in Teiltabellen kann es durchaus vorkommen, dass bestimmte Sachverhalte verloren gehen. Aus diesem Grund hat man Kriterien gesucht, die

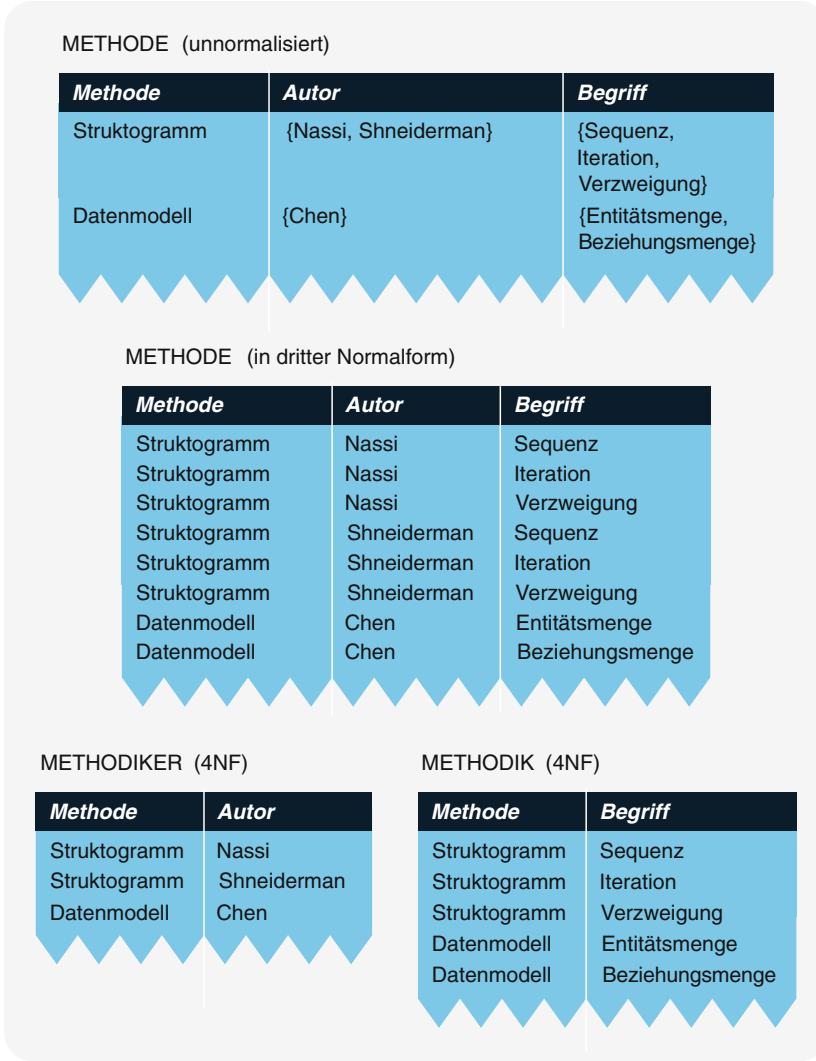

Abb. 2.13 Tabelle mit mehrwertigen Abhängigkeiten

eine *Aufteilung von Tabellen ohne Informationsverlust* garantieren, d. h. so dass die ursprünglichen Sachverhalte über die Teiltabellen erhalten bleiben.

In Abb. 2.14 ist eine unerlaubte Zerlegung der Tabelle EINKAUF in die beiden Teiltabellen KÄUFER und WEIN dargestellt: Die Tabelle EINKAUF enthält die getätigten Weinkäufe von Schweizer, Meier, Huber und Becker. Kombiniert man die davon abgeleiteten Teiltabellen KÄUFER und WEIN über das gemeinsame Merkmal Weinsorte, so erhält man die Tabelle FALSCHEINKAUF. Diese Tabelle zeigt gegenüber der Ursprungstabelle EINKAUF einen Informationskonflikt auf, da hier suggeriert wird, dass

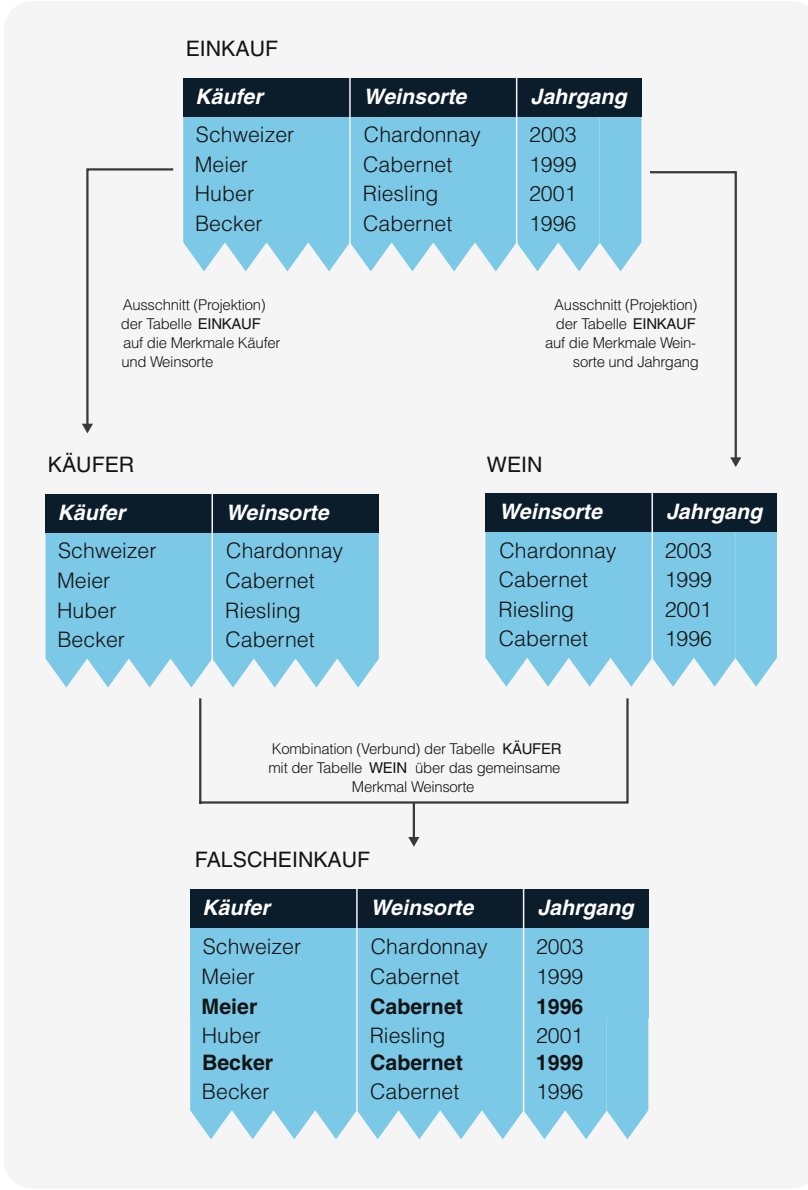

Abb. 2.14 Unerlaubte Zerlegung der Tabelle EINKAUF

Meier und Becker dieselben Weinpräferenzen besitzen und je den Cabernet 1996 wie 1999 eingekauft haben (die fälschlichen Tabelleneinträge für Meier und Becker sind in Abb. 2.14 durch fette Schreibweise hervorgehoben).

Das Zerlegen der Tabelle EINKAUF in die Tabelle KÄUFER geschieht, indem man die Einkaufstabelle auf die beiden Merkmale Käufer und Weinsorte reduziert. Der dazu

notwendige Projektionsoperator ist ein Filteroperator, der Tabellen vertikal in Teiltabellen zerlegt (vgl. Abb. 3.3 resp. Abschn. 3.2.3). Entsprechend erhält man die Teiltabelle WEIN durch die Projektion der Tabelle EINKAUF auf die beiden Merkmale Weinsorte und Jahrgang. Neben dem Zerlegen von Tabellen in Teiltabellen (Projektion) können Teiltabellen auch zu Tabellen kombiniert werden (Verbundoperator). Ein Beispiel ist in Abb. 2.14 aufgezeigt: Die beiden Tabellen KÄUFER und WEIN können dank dem gemeinsamen Merkmal Weinsorte kombiniert werden, indem alle Tupeleinträge aus der Tabelle KÄUFER mit den entsprechenden Einträgen aus der Tabelle WEIN ergänzt werden. Als Resultat bekommt man die Verbundstabelle FALSCHEINKAUF (zum Verbund oder Join siehe Abb. 3.3 resp. Abschn. 3.2.3). Die Tabelle listet alle Einkäufe der Personen Schweizer, Meier, Huber und Becker. Allerdings enthält sie auch die beiden Einkäufe (Meier, Cabernet, 1996) und (Becker, Cabernet, 1999), die gar nie getätigt wurden. Abhilfe für solche Konfliktfälle schafft die fünfte Normalform resp. das Studium der Verbundabhängigkeit.

Fünfte Normalform (5NF)
Eine Tabelle ist in fünfter Normalform, wenn sie *keine Verbundabhängigkeit* aufweist.

Die fünfte Normalform (oft auch Projektion-Verbund-Normalform genannt) gibt an, unter welchen Umständen eine Tabelle problemlos in Teiltabellen zerlegt und gegebenenfalls ohne Einschränkung (Informationskonflikt) wieder rekonstruiert werden kann. Die Zerlegung wird mit Hilfe des Projektionsoperators durchgeführt, die Rekonstruktion mit dem Verbundoperator.

Um bei der Rekonstruktion keine falschen Sachverhalte zu erhalten, muss auf Verbundabhängigkeit geprüft werden: Eine Tabelle R mit den Merkmalen A, B und C erfüllt die *Verbundabhängigkeit* (engl. *join dependency*), falls die projizierten Teiltabellen R1(A,B) und R2(B,C) beim Verbund über das gemeinsame Merkmal B die Ursprungstabelle R ergeben. Man spricht in diesem Zusammenhang auch von einem *verlustlosen Verbund* (engl. *lossless join*). Wie bereits diskutiert, ist die Tabelle EINKAUF aus Abb. 2.14 nicht verbundabhängig und somit nicht in fünfter Normalform.

In Abb. 2.15 ist die Tabelle EINKAUF gegeben, hier in vierter Normalform. Da sie das Kriterium der Verbundabhängigkeit verletzt, muss sie in die fünfte Normalform überführt werden. Dies ist möglich, falls die drei Teiltabellen KÄUFER, WEIN und PRÄFERENZ durch entsprechende Projektionen aus der Tabelle EINKAUF gewonnen werden. Eine Nachprüfung bestätigt, dass ein Verbund der drei Tabellen tatsächlich die Ursprungstabelle EINKAUF ergibt. Dazu müssen die beiden Tabellen KÄUFER und WEIN über das gemeinsame Merkmal Weinsorte kombiniert werden. Die Resultatstabelle kann dann mit einem weiteren Verbund über das Merkmal Jahrgang mit der Tabelle PRÄFERENZ zur Einkaufstabelle kombiniert werden.

Allgemein ist die Frage von Interesse, ob man beim Datenbankentwurf anstelle von Zerlegungsregeln oder Normalformen (analytisches Vorgehen) umgekehrt auch synthetisch vorgehen kann. Tatsächlich gibt es Algorithmen, die das Zusammensetzen von Teiltabellen zu

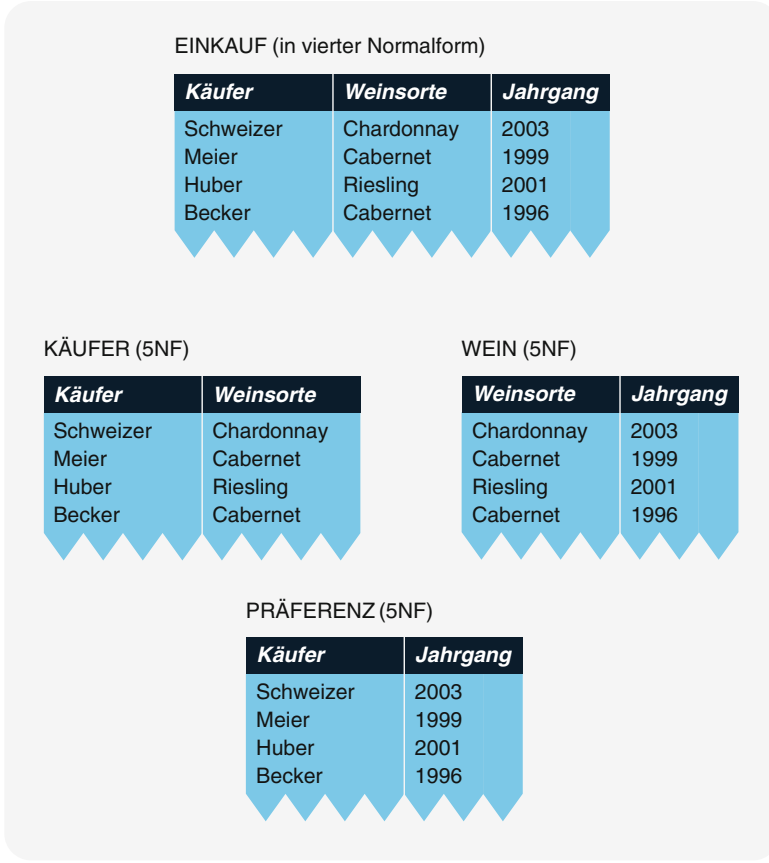

Abb. 2.15 Beispiel von Tabellen in fünfter Normalform

Tabellen in dritter oder höherer Normalform ermöglichen. Solche Zusammensetzungsregeln erlauben, ausgehend von einem Satz von Abhängigkeiten entsprechende Datenbankschemas aufbauen zu können (synthetisches Vorgehen). Sie begründen formell, dass der Entwurf relationaler Datenbankschemas sowohl top down (analytisch) als auch bottom up (synthetisch) erfolgreich vorgenommen werden kann. Leider unterstützen nur wenige CASE-Werkzeuge beide Vorgehensweisen; der Datenbankarchitekt tut deshalb gut daran, die Korrektheit seines Datenbankentwurfs manuell zu überprüfen.

2.3.2 Abbildungsregeln für relationale Datenbanken

In diesem Abschnitt behandeln wir die Abbildung des Entitäten-Beziehungsmodells auf ein relationales Datenbankschema. Wir halten dabei fest, wie sich *Entitäts- und Beziehungsmengen* durch Tabellen darstellen lassen.

Unter einem *Datenbankschema* (engl. *database schema*) versteht man eine Datenbankbeschreibung, d. h. die Spezifikation von Datenstrukturen mitsamt ihren zugehörigen Integritätsbedingungen. Ein relationales Datenbankschema enthält die Definition der Tabellen, der Merkmale und der Primärschlüssel. Integritätsbedingungen legen Einschränkungen für die Wertebereiche, für die Abhängigkeiten zwischen verschiedenen Tabellen (referentielle Integrität gemäß Abschn. 2.3.3) sowie für die eigentlichen Datenvorkommen fest.

Beim Überführen eines Entitäten-Beziehungsmodells auf ein relationales Datenbankschema sind die Abbildungsregeln R1 und R2 von Bedeutung (vgl. Abb. 2.16).

Regel R1 (Entitätsmengen)
Jede *Entitätsmenge muss als eigenständige Tabelle* mit einem eindeutigen Primärschlüssel definiert werden. Als Primärschlüssel der Tabelle dient entweder der entsprechende Schlüssel der Entitätsmenge oder ein spezifischer Schlüsselkandidat. Die übrigen Merkmale der Entitätsmengen gehen in die korrespondierenden Attribute der Tabellen über.

Die Definition einer Tabelle (Abschn. 1.2.1) verlangt einen eindeutigen Primärschlüssel. Nun kann es sein, dass in einer Tabelle mehrere *Schlüsselkandidaten* (engl. *candidate keys*) vorliegen, die allesamt die Forderung nach Eindeutigkeit und Minimalität erfüllen. Dann entscheiden die Datenarchitekten, welchen der Schlüsselkandidaten sie als Primärschlüssel verwenden möchten.

Regel R2 (Beziehungsmengen)
Jede *Beziehungsmenge kann als eigenständige Tabelle* definiert werden, wobei die Identifikationsschlüssel der zugehörigen Entitätsmengen als *Fremdschlüssel* in dieser Tabelle auftreten müssen. Der Primärschlüssel der Beziehungsmengentabelle kann der aus den Fremdschlüsseln zusammengesetzte Identifikationsschlüssel sein oder ein anderer Schlüsselkandidat, beispielsweise in Form eines künstlichen Schlüssels. Weitere Merkmale der Beziehungsmenge erscheinen als zusätzliche Attribute in der Tabelle.

Als *Fremdschlüssel* (engl. *foreign key*) einer Tabelle wird ein Merkmal bezeichnet, das in derselben oder in einer anderen Tabelle als Identifikationsschlüssel auftritt. Somit dürfen Identifikationsschlüssel in weiteren Tabellen wiederverwendet werden, um die gewünschten Beziehungen zwischen den Tabellen herstellen zu können.

In Abb. 2.16 zeigen wir die Anwendung der Regeln R1 und R2 anhand unseres konkreten Beispiels. Jede Entitätsmenge ABTEILUNG, MITARBEITER und PROJEKT wird in ihre entsprechende Tabelle ABTEILUNG, MITARBEITER und PROJEKT überführt. Auf analoge Art definieren wir zu jeder Beziehungsmenge ABTEILUNGSLEITER, UNTERSTELLUNG und ZUGEHÖRIGKEIT eine entsprechende Tabelle. Die Tabellen ABTEILUNGSLEITER und UNTERSTELLUNG verwenden die Abteilungsnummer und die Mitarbeiternummer als Fremdschlüssel. Die Tabelle ZUGEHÖRIGKEIT entlehnt

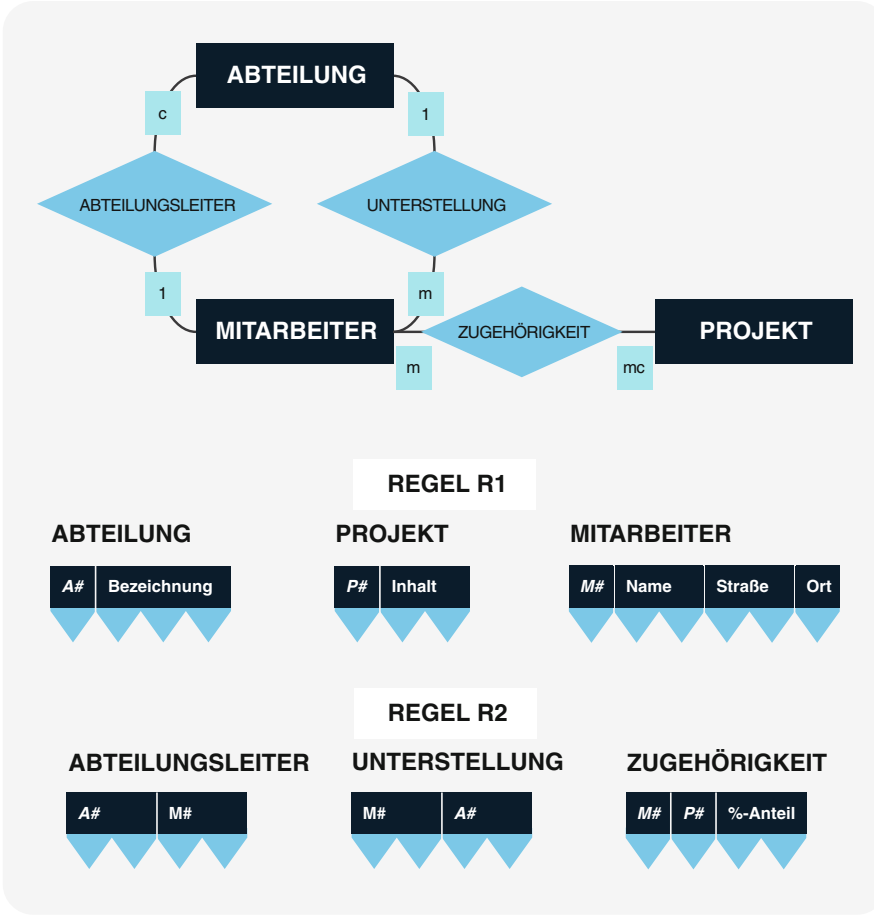

Abb. 2.16 Abbildung von Entitäts- und Beziehungsmengen in Tabellen

die Identifikationsschlüssel den beiden Tabellen MITARBEITER und PROJEKT und führt das Merkmal «%-Anteil» als weiteres Beziehungsmerkmal auf.

Da zu jeder Abteilung genau ein Abteilungsleiter gehört, genügt die Abteilungsnummer A# in der Tabelle ABTEILUNGSLEITER als Identifikationsschlüssel. Auf analoge Art erklären wir die Mitarbeiternummer M# als Identifikationsschlüssel in der Tabelle UNTERSTELLUNG. Die Mitarbeiternummer genügt hier, da jeder Mitarbeitende genau einer Abteilung unterstellt ist.

Im Gegensatz zu den Tabellen ABTEILUNGSLEITER und UNTERSTELLUNG müssen bei der Tabelle ZUGEHÖRIGKEIT die beiden Fremdschlüssel der Mitarbeiter- und Projektnummer als zusammengesetzter Identifikationsschlüssel definiert werden. Der

Grund liegt in der Tatsache, dass ein Mitarbeitender mehrere Projekte bearbeiten kann und dass umgekehrt in jedes Projekt mehrere Mitarbeitende mit einbezogen sein können.

Die Anwendung der Regeln R1 und R2 führt nicht in jedem Fall zu einem optimalen relationalen Datenbankschema. Störend wirkt, dass bei diesem Vorgehen unter Umständen eine hohe Anzahl von Tabellen entsteht. Es stellt sich die Frage, ob es sinnvoll ist, gemäß Abb. 2.16 für die Funktion des Abteilungsleiters eine eigene Tabelle zu verlangen. Wie im nächsten Abschnitt erläutert, könnten wir aufgrund der Abbildungsregel R5 tatsächlich auf die Tabelle ABTEILUNGSLEITER verzichten. Die Funktion «Abteilungsleiter» würde in der Tabelle ABTEILUNG lediglich als zusätzliche Merkmalskategorie stehen und für jede Abteilung die Mitarbeiternummer des jeweiligen Abteilungschefs anführen.

Abbildungsregeln für Beziehungsmengen

Aufgrund der Mächtigkeit von Beziehungen lassen sich drei Abbildungsregeln formulieren, die die Beziehungsmengen des Entitäten-Beziehungsmodells im entsprechenden relationalen Datenbankschema tabellarisch ausdrücken. Um die Anzahl der Tabellen nicht unnötig zu erhöhen, beschränkt man sich mit Hilfe der Regel 3 vorerst auf diejenigen Beziehungsmengen, die *in jedem Fall eine eigene Tabelle* verlangen:

Regel R3 (netzwerkartige Beziehungsmengen)
Jede komplex-komplexe Beziehungsmenge muss als eigenständige Tabelle definiert werden. Dabei treten mindestens die Identifikationsschlüssel der zugehörigen Entitätsmengen als Fremdschlüssel auf. Der Primärschlüssel der Beziehungsmengentabelle ist entweder der aus den Fremdschlüsseln zusammengesetzte Identifikationsschlüssel oder ein anderer Schlüsselkandidat. Die weiteren Merkmale der Beziehungsmenge gehen in Attribute der Tabelle über.

Die Regel R3 schreibt vor, dass die Beziehungsmenge ZUGEHÖRIGKEIT aus Abb. 2.17 als eigenständige Tabelle auftreten muss, versehen mit einem Primärschlüssel. Als Identifikationsschlüssel der Tabelle ZUGEHÖRIGKEIT wird hier der zusammengesetzte Schlüssel ausgewählt, der die Fremdschlüsselbeziehungen zu den Tabellen MITARBEITER und PROJEKT ausdrückt. Das Merkmal «%-Anteil» beschreibt das prozentuale Ausmaß dieser Zugehörigkeit.

Wie wir gesehen haben, könnten wir gemäß Regel R2 für die Beziehungsmenge UNTERSTELLUNG eine eigenständige Tabelle definieren, und zwar mit den beiden Fremdschlüsseln Abteilungs- und Mitarbeiternummer. Dies wäre dann sinnvoll, wenn wir eine Matrixorganisation unterstützen und die eindeutige Unterstellungseigenschaft mit dem Assoziationstyp 1 in nächster Zukunft aufgeben möchten; zwischen ABTEILUNG und MITARBEITER wäre in diesem Fall eine komplex-komplexe Beziehung festgelegt. Sind wir hingegen überzeugt, dass keine Matrixorganisation vorgesehen ist, so können wir aufgrund der einfach-komplexen Beziehung die Regel R4 anwenden:

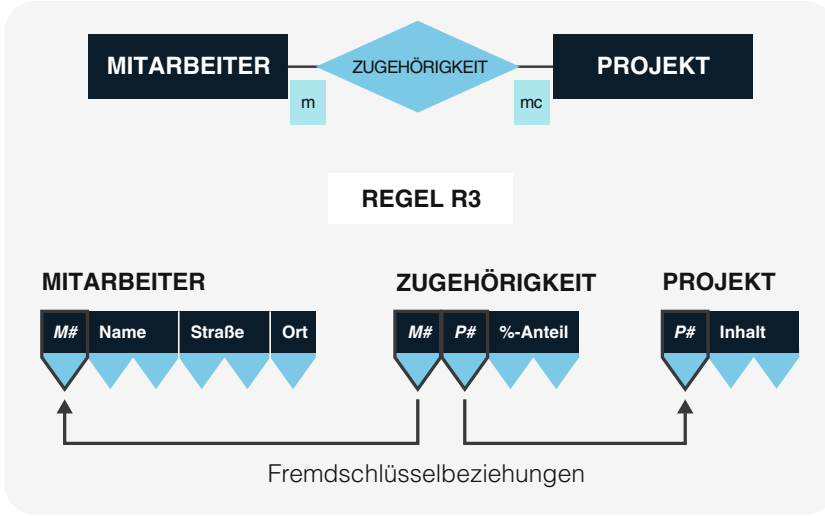

Abb. 2.17 Abbildungsregel für komplex-komplexe Beziehungsmengen

Regel R4 (hierarchische Beziehungsmengen)
Eine *einfach-komplexe Beziehungsmenge kann ohne eine eigenständige Beziehungsmengentabelle* durch die beiden Tabellen der zugeordneten Entitätsmengen ausgedrückt werden. Dazu bewirkt die einfache Assoziation (d. h. Assoziationstyp 1 oder c), dass der Primärschlüssel der referenzierten Tabelle als Fremdschlüssel in der Ausgangstabelle verwendet wird, ergänzt mit einem entsprechenden *Rollennamen*.

In Abb. 2.18 verzichten wir gemäß Regel R4 auf eine eigenständige Tabelle UNTERSTELLUNG. Anstelle einer zusätzlichen Beziehungsmengentabelle erweitern wir die Tabelle MITARBEITER um den Fremdschlüssel «A#_Unterstellung», der die Abteilungsnummer der Unterstellungsbeziehung pro Mitarbeitenden angibt. Die Fremdschlüsselbeziehung wird durch ein Merkmal gegeben, das sich aus dem entliehenen Identifikationsschlüssel A# und dem Rollennamen «Unterstellung» zusammensetzt.

Liegen «einfach-komplexe» Beziehungsmengen vor, so ist das Entleihen des Fremdschlüssels auf eindeutige Art möglich. In Abb. 2.18 führen wir gemäß Regel R4 die Abteilungsnummer als Fremdschlüssel in der Tabelle MITARBEITER. Würden wir umgekehrt die Mitarbeiternummer in der Tabelle ABTEILUNG vorsehen, müssten wir für jeden Mitarbeitenden einer Abteilung die Abteilungsbezeichnung wiederholen. Solche überflüssige oder redundante Informationen sind unerwünscht und widersprechen der Theorie der Normalformen (hier Verletzung der zweiten Normalform, siehe Abschn. 2.3.1).

Regel R5 (einfach-einfache Beziehungsmengen)
Eine *einfach-einfache Beziehungsmenge kann ohne eine eigenständige Tabelle* durch die beiden Tabellen der zugeordneten Entitätsmengen ausgedrückt werden, indem einer der

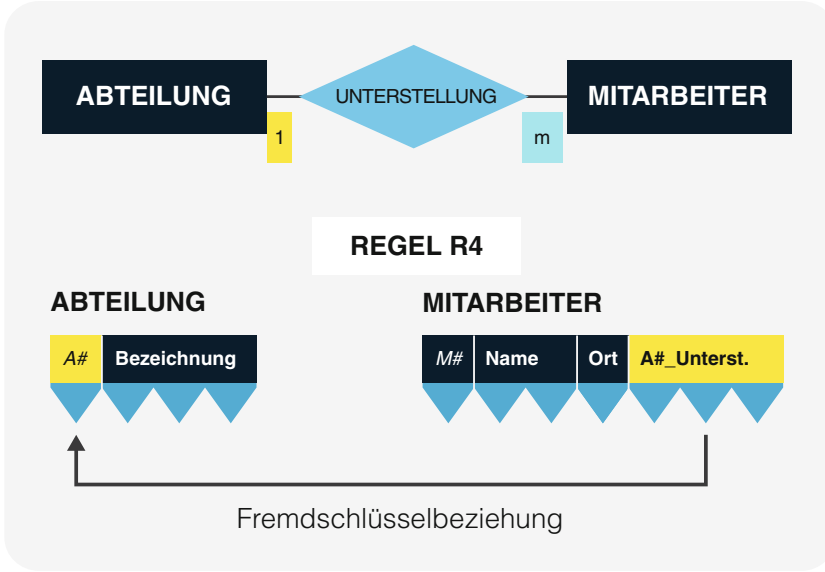

Abb. 2.18 Abbildungsregel für einfach-komplexe Beziehungsmengen

Identifikationsschlüssel der referenzierten Tabelle als Fremdschlüssel in die Ausgangstabelle eingebracht wird, wiederum ergänzt um den entsprechenden Rollennamen.

Auch hier ist nicht unbedeutend, aus welcher der beiden Tabellen solche Fremdschlüsselkandidaten entliehen werden. Normalerweise bevorzugen wir eindeutige Assoziationstypen, damit in der Ausgangstabelle die Fremdschlüssel mit ihren Rollennamen für jedes Tupel aufgenommen werden können (Vermeidung von sogenannten Nullwerten, vgl. Abschn. 3.6).

In Abb. 2.19 ergänzen wir die Tabelle ABTEILUNG um die Mitarbeiternummer des Abteilungsleiters. Die Beziehungsmenge ABTEILUNGSLEITER wird damit durch das Merkmal «M#_Abteilungsleiter» ausgedrückt. Jeder Eintrag in diesem fremdbezogenen Merkmal mit der Rolle «Abteilungsleiter» zeigt, wer der jeweilige Abteilungsleiter ist.

Würden wir anstelle dieses Vorgehens die Abteilungsnummer in der Tabelle MITARBEITER führen, so müssten wir für die meisten Mitarbeitenden Nullwerte auflisten. Lediglich bei denjenigen Mitarbeitenden, die eine Abteilung leiten, könnten wir die entsprechende Abteilungsnummer einsetzen. Da Nullwerte in der Praxis oft zu Problemen führen, sucht man solche zu vermeiden. Aus diesem Grund führen wir die Rolle des Abteilungsleiters in der Tabelle ABTEILUNG. Bei (1,c)- oder (c,1)-Beziehungen können wir auf diese Weise Nullwerte bei den Fremdschlüsseln komplett ausschließen. Bei (c,c)-Beziehungen kann nach dem Grundsatz entschieden werden, dass möglichst wenig Nullwerte entstehen.

Abbildungsregeln für Generalisation und Aggregation

Treten in einem Entitäten-Beziehungsmodell Generalisationshierarchien oder Aggregationsstrukturen auf, so müssen diese ebenfalls in ein relationales Datenbankschema

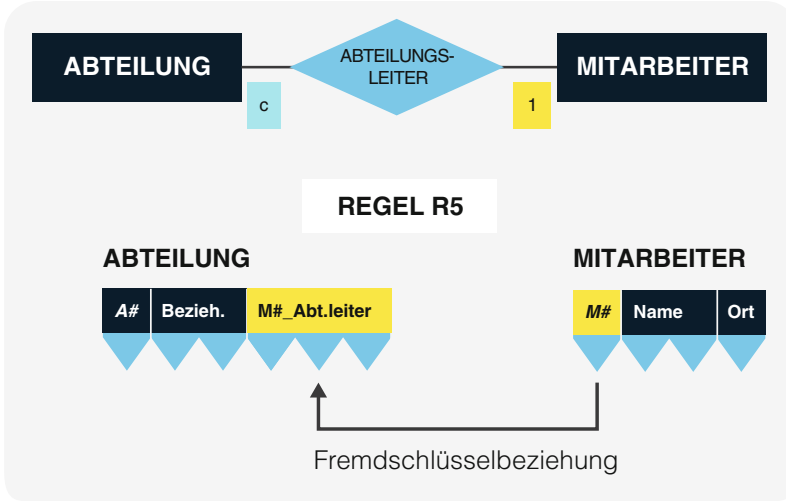

Abb. 2.19 Abbildungsregel für einfach-einfache Beziehungsmengen

überführt werden. Obwohl die bereits bekannten Assoziationstypen bei diesen speziellen Arten von Beziehungen erscheinen, unterscheiden sich die entsprechenden Abbildungsregeln von den bisher besprochenen.

Regel R6 (Generalisation)
Jede *Entitätsmenge einer Generalisationshierarchie* verlangt eine *eigenständige Tabelle*, wobei der Primärschlüssel der übergeordneten Tabelle auch Primärschlüssel der untergeordneten Tabellen wird.

Da das Relationenmodell die Beziehungsstruktur einer Generalisation nicht direkt unterstützt, müssen die Eigenschaften dieser Beziehungshierarchie indirekt nachgebildet werden. Bei einer *überlappend-unvollständigen, überlappend-vollständigen, disjunkt-unvollständigen oder disjunkt-vollständigen Generalisation* müssen die Identifikationsschlüssel der Spezialisierungen immer mit denjenigen der übergeordneten Tabelle übereinstimmen. Die Eigenschaft von überlappenden Spezialisierungen verlangt keine speziellen Prüfregeln, hingegen muss die Disjunktheit im Relationenmodell nachvollzogen werden. Eine Möglichkeit besteht darin, in der übergeordneten Tabelle ein Merkmal «Kategorie» mitzuführen. Dieses Merkmal entspricht einer Klassenbildung und drückt aus, um welche Spezialisierung es sich handelt. Darüber hinaus muss bei einer disjunkten und vollständigen Generalisation garantiert werden, dass pro Eintrag in der übergeordneten Tabelle genau ein Eintrag in einer Tabelle der Spezialisierung vorliegt und umgekehrt.

In Abb. 2.20 zeigen wir die Mitarbeiterinformation als Generalisation. Die Regel R6 führt zu den Tabellen MITARBEITER, FÜHRUNGSKRAFT, FACHSPEZIALIST und LEHRLING. In den von der Tabelle MITARBEITER abhängigen Tabellen muss

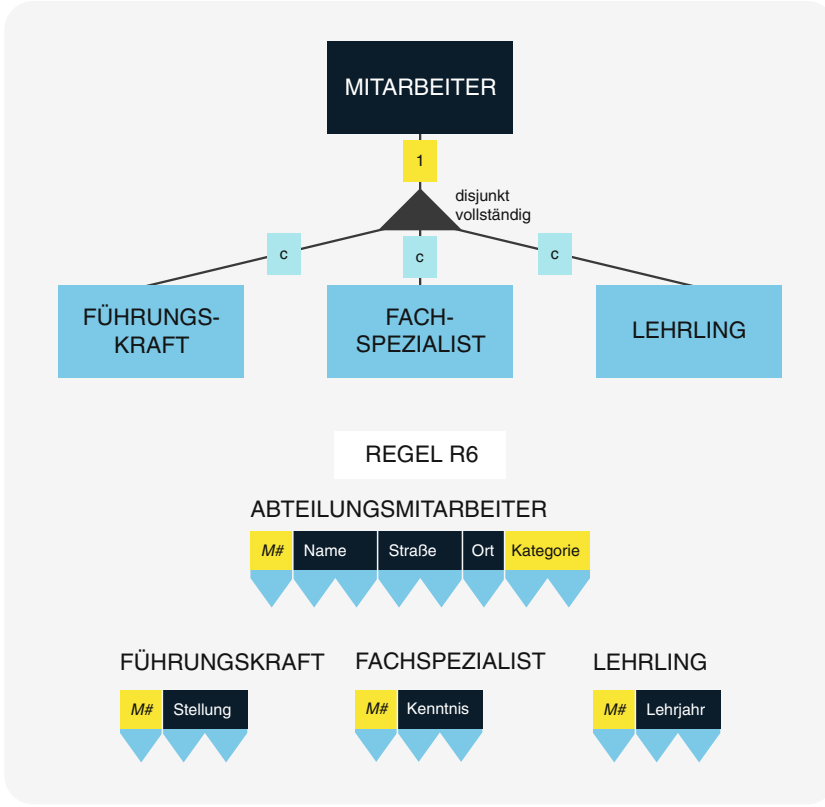

Abb. 2.20 Generalisation in Tabellenform

derselbe Identifikationsschlüssel M# verwendet werden. Damit ein bestimmter Mitarbeitender nicht mehreren Kategorien gleichzeitig angehört, führen wir das Merkmal «Kategorie» ein. Dieses kann die Werte «Führungskraft», «Fachspezialist» oder «Lehrling» annehmen. Es garantiert somit die Eigenschaft einer disjunkten Generalisationshierarchie (gemäß Abschn. 2.2.3), wonach die einzelnen Entitätsmengen der Spezialisierung sich nicht gegenseitig überlappen dürfen. Die Eigenschaft der Vollständigkeit kann nicht explizit im Datenbankschema ausgedrückt werden und bedarf einer speziellen Integritätsbedingung.

Regel R7 (Aggregation)
Bei einer Aggregation müssen sowohl die Entitätsmenge als auch die Beziehungsmenge je als *eigenständige Tabelle* definiert werden, falls der Beziehungstyp *komplex-komplex* ist. Die Tabelle der Beziehungsmenge enthält in diesem Fall zweimal den Schlüssel aus der Tabelle der zugehörigen Entitätsmenge als zusammengesetzten Identifikationsschlüssel, mit entsprechenden Rollennamen. Im Falle einer *einfach-komplexen* Beziehung

(hierarchische Struktur) kann die Entitätsmenge mit der Beziehungsmenge zu einer *einzigen Tabelle* kombiniert werden.

Im Beispiel der Abb. 2.21 weist die Konzernstruktur die Mächtigkeit (mc,mc) auf. Gemäß Regel R7 müssen zwei Tabellen FIRMA und KONZERNSTRUKTUR festgelegt werden. Die Beziehungsmengentabelle KONZERNSTRUKTUR zeigt durch Identifikationsschlüssel in jedem Eintrag an, welche die direkt abhängigen Teile einer Firmengruppe bzw. welche die direkt übergeordneten Gruppen eines bestimmten Firmenteils sind. Neben den beiden Fremdschlüsselbeziehungen vervollständigt ein Beteiligungsmerkmal die Tabelle KONZERNSTRUKTUR.

Eine hierarchische Aggregationsstruktur ist in Abb. 2.22 illustriert. In der Tabelle ARTIKEL werden die einzelnen Komponenten mit ihren Materialeigenschaften aufgeführt. Die Tabelle STÜCKLISTE definiert die hierarchische Beziehungsstruktur der jeweiligen Baugruppen. So ist der Artikel A7 aus zwei Teilgruppen zusammengesetzt, und zwar aus A9 und A11. Die Teilgruppe A11 wiederum setzt sich aus den Teilen A3 und A4 zusammen.

Wenn die Stückliste eine Baumstruktur hat, also wenn jedes Unterteil genau ein Oberteil besitzt, können die beiden Tabellen ARTIKEL und STÜCKLISTE in einer einzigen Tabelle ARTIKELSTRUKTUR zusammengefasst werden. Dabei würde man zu den Artikeleigenschaften je die Artikelnummer des eindeutig übergeordneten Artikels aufführen.

2.3.3 Strukturelle Integritätsbedingungen

Unter dem Begriff *Integrität* oder *Konsistenz* (engl. *integrity, consistency*) versteht man die Widerspruchsfreiheit von Datenbeständen. Eine Datenbank ist integer oder konsistent, wenn die gespeicherten Daten fehlerfrei erfasst sind und den gewünschten Informationsgehalt korrekt wiedergeben. Die Datenintegrität ist dagegen verletzt, wenn Mehrdeutigkeiten oder widersprüchliche Sachverhalte zu Tage treten. Bei einer konsistenten Tabelle MITARBEITER setzen wir beispielsweise voraus, dass die Namen der Mitarbeitenden, Straßenbezeichnungen, Ortsangaben etc. korrekt sind und real existieren.

Strukturelle Integritätsbedingungen zur Gewährleistung der Integrität sind solche Regeln, die durch das Datenbankschema selbst ausgedrückt werden können. Bei relationalen Datenbanken zählen die folgenden Integritätsbedingungen zu den strukturellen dazu:

- **Eindeutigkeitsbedingung**: Jede Tabelle besitzt einen Identifikationsschlüssel (Merkmal oder Merkmalskombination), der jedes Tupel in der Tabelle auf eindeutige Art bestimmt.
- **Wertebereichsbedingung**: Die Merkmale einer Tabelle können nur Datenwerte aus einem vordefinierten Wertebereich annehmen.
- **Referenzielle Integritätsbedingung**: Jeder Wert eines Fremdschlüssels muss effektiv als Schlüsselwert in der referenzierten Tabelle existieren.

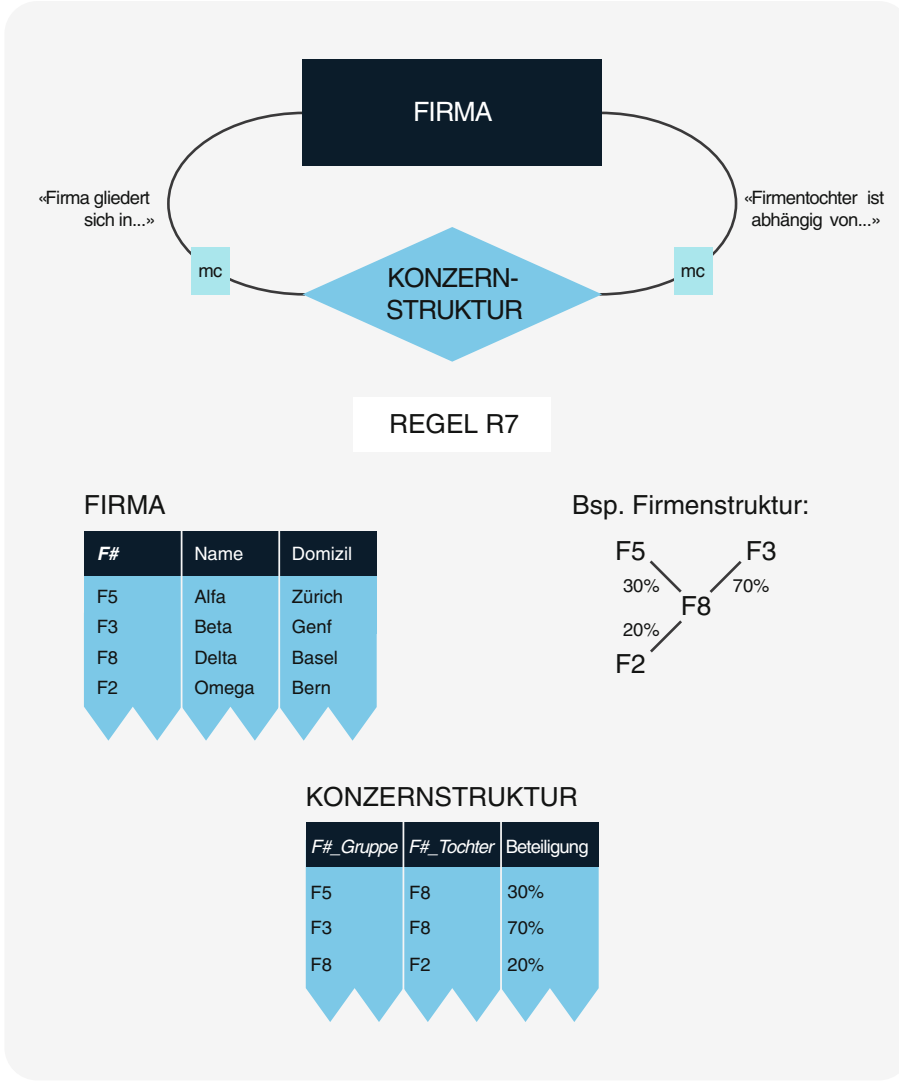

Abb. 2.21 Netzwerkartige Firmenstruktur in Tabellenform

Die *Eindeutigkeitsbedingung* verlangt für jede Tabelle einen festgelegten Schlüssel. Innerhalb derselben Tabelle können mehrere Schlüsselkandidaten vorkommen. In diesem Fall muss aufgrund der Eindeutigkeitsbedingung ein Schlüssel als Primärschlüssel deklariert werden. Das Prüfen auf Eindeutigkeit von Primärschlüsseln selbst wird vom Datenbanksystem vorgenommen.

Die *Wertebereichsbedingung* kann hingegen nicht vollständig vom Datenbanksystem gewährleistet werden. Zwar können die Wertebereiche einzelner Tabellenspalten spezifiziert

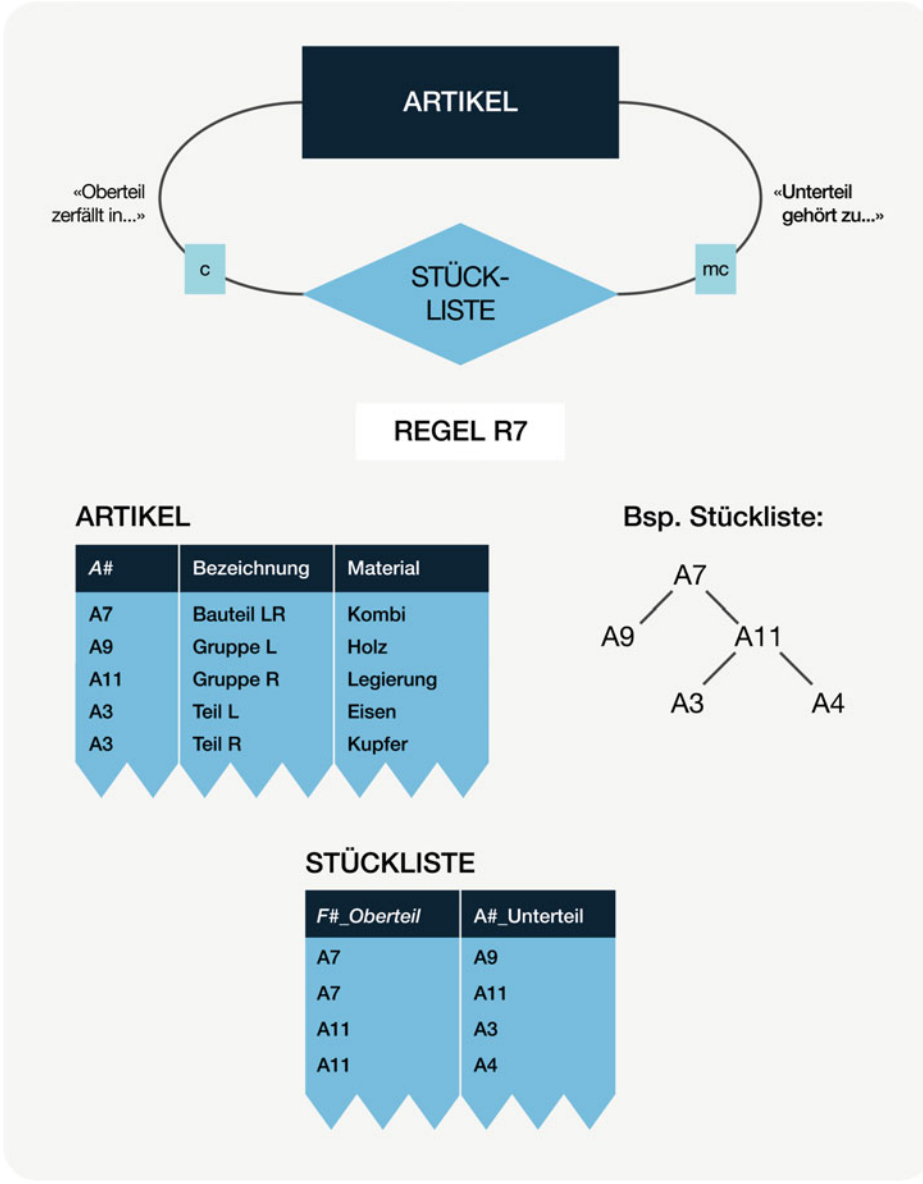

Abb. 2.22 Hierarchische Artikelstruktur in Tabellenform

werden, doch decken solche Angaben nur einen geringen Teil der Validierungsregeln ab. Bei der Prüfung von Ortsbezeichnungen oder Straßenangaben auf Korrektheit ist es mit der Definition eines Wertebereiches nicht getan. So sagt beispielsweise die Spezifikation «CHARACTER (20)» für eine Zeichenkette nichts über sinnvolle Orts- oder Straßennamen aus.

Wie weit man bei einer Validierung von Tabelleninhalten gehen möchte, bleibt größtenteils den Anwendenden überlassen.

Eine bedeutende Unterstützung für die Wertebereichsbedingung bieten *Aufzählungstypen*. Dabei werden sämtliche Datenwerte, die das Merkmal annehmen kann, in einer Liste angegeben. Beispiele von Aufzählungstypen sind durch die Merkmalskategorien (Attribute) BERUF = {ProgrammiererIn, AnalytikerIn, OrganisatorIn} oder JAHRGANG = {1950..1990} gegeben. Die meisten heutigen Datenbanksysteme unterstützen solche Validierungsregeln.

Eine wichtige Klasse von Prüfregeln wird unter dem Begriff *referenzielle Integrität* (engl. *referential integrity*) zusammengefasst. Eine relationale Datenbank erfüllt die referenzielle Integritätsbedingung, wenn jeder Wert eines Fremdschlüssels als Wert beim zugehörigen Primärschlüssel vorkommt. Die Abb. 2.23 macht dies deutlich: Die Tabelle ABTEILUNG hat die Abteilungsnummer A# als Primärschlüssel. Dieser wird in der Tabelle MITARBEITER als Fremdschlüssel mit dem Merkmal «A#_Unterstellung» verwendet, um die Abteilungszugehörigkeit der Mitarbeitenden zu fixieren. Die Fremd-Primärschlüssel-Beziehung erfüllt die Regel der referenziellen Integrität, falls alle Abteilungsnummern des Fremdschlüssels aus der Tabelle MITARBEITER in der Tabelle ABTEILUNG als Primärschlüsselwerte aufgeführt sind. In unserem Beispiel verletzt also keine Unterstellung die Regel der referenziellen Integrität.

Nehmen wir an, wir möchten in die Tabelle MITARBEITER, wie sie Abb. 2.23 zeigt, ein neues Tupel «M20, Müller, Riesweg, Olten, A7» einfügen. Unsere Einfügeoperation wird abgewiesen, falls das Datenbanksystem die referenzielle Integrität unterstützt. Der Wert A7 wird als ungültig erklärt, da er in der referenzierten Tabelle ABTEILUNG nicht vorkommt.

Die Gewährleistung der referenziellen Integrität hat neben der Einfügeproblematik natürlich Einfluss auf die übrigen Datenbankoperationen. Wird ein Tupel in einer Tabelle gelöscht, das von anderen Tupeln aus einer Fremdtabelle referenziert wird, so sind mehrere Varianten von Systemreaktionen möglich:

- **Restriktive Löschung**: Bei der *restriktiven Löschregel* (engl. *restricted deletion*) wird eine Löschoperation nicht ausgeführt, solange das zu löschende Tupel auch nur von einem Tupel einer anderen Tabelle referenziert wird. Möchten wir das Tupel «A6, Finanz» in Abb. 2.23 entfernen, so wird nach der restriktiven Löschregel unsere Operation verweigert, da die beiden Mitarbeitenden Schweizer und Becker in der Abteilung A6 tätig sind.

Anstelle der restriktiven Löschung kann für die Löschoperation eine *fortgesetzte Löschregel* (engl. *cascaded deletion*) bei der Spezifikation der beiden Tabellen MITARBEITER und ABTEILUNG verlangt werden:

- **Fortgesetzte Löschung**: Eine solche bewirkt, dass bei der Löschung eines Tupels *sämtliche abhängigen Tupel entfernt* werden. In unserem Beispiel aus Abb. 2.23

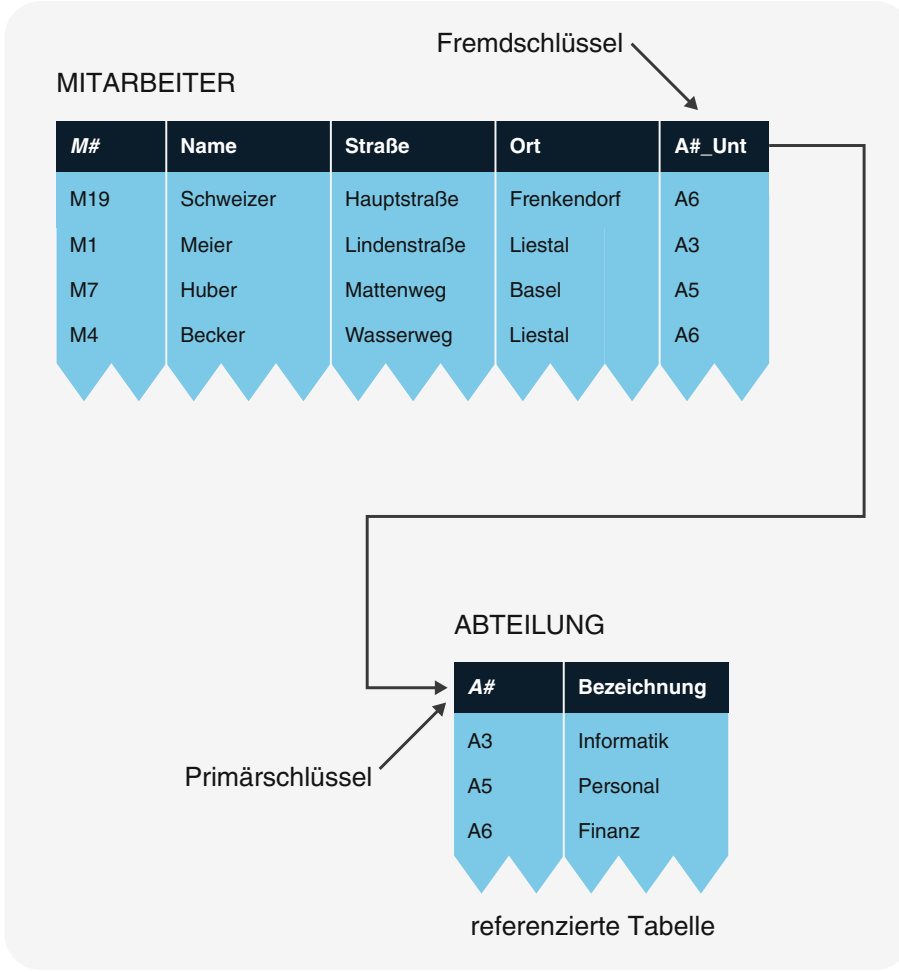

Abb. 2.23 Gewährleistung der referenziellen Integrität

bedeutet eine fortgesetzte Löschregel bei der Eliminierung des Tupels (A6, Finanz), dass gleichzeitig die beiden Tupel (M19, Schweizer, Hauptstraße, Frenkendorf, A6) und (M4, Becker, Wasserweg, Liestal, A6) entfernt werden.

Eine weitere Löschregelvariante unter Berücksichtigung der referenziellen Integrität macht es möglich, dass referenzierte Fremdschlüssel bei Löschvorgängen auf den Wert «unbekannt» gesetzt werden Diese dritte Löschregel wird nach der Behandlung der sogenannten Nullwerte (Abschn. 3.6) durch Integritätsbedingungen in Abschn. 3.7 näher erläutert. Schließlich können an Änderungsoperationen einschränkende Bedingungen geknüpft werden, die die referenzielle Integrität jederzeit garantieren.

2.4 Umsetzung im Graphenmodell

2.4.1 Eigenschaften von Graphen

Die Graphentheorie ist ein umfangreiches Fachgebiet, das aus vielen Anwendungsgebieten nicht mehr wegzudenken ist. Sie findet überall dort Gebrauch, wo netzwerkartige Strukturen analysiert oder optimiert werden müssen. Stichworte dazu sind Rechnernetzwerke, Transportsysteme, Robotereinsätze, Energieleitsysteme, elektronische Schaltungen, soziale Netze oder betriebswirtschaftliche Themengebiete wie Konzernstrukturen, Ablauforganisation, Kundenmanagement, Logistik, Prozessmanagement etc.

Ein Graph ist durch die Menge seiner Knoten und Kanten gegeben sowie einer Zuordnung unter diesen Mengen.

Ungerichteter Graph
Ein *ungerichteter Graph* (engl. *undirected graph*) $G = (V,E)$ besteht aus einer Knotenmenge V (engl. *vertices*) und einer Kantenmenge E (engl. *edges*), wobei jeder Kante zwei nicht notwendigerweise verschiedene Knoten zugeordnet sind.

Graphdatenbanken basieren oft auf dem Modell der gerichteten, gewichteten Graphen. Im Moment interessieren wir uns noch nicht für die Art und die Beschaffung der Knoten und Kanten, sondern beschränken uns auf das abstrakte allgemeine Modell eines ungerichteten Graphen. Trotzdem können auf dieser Abstraktionsstufe viele Eigenschaften von Netzstrukturen untersucht werden, beispielsweise:

- Wie viele Kanten muss man durchlaufen, um von einem Knoten zu einem anderen zu gelangen?
- Gibt es zwischen zwei Knoten einen Weg?
- Kann man die Kanten eines Graphen so durchlaufen, dass man jeden Knoten einmal besucht?
- Kann man einen Graphen so zeichnen, dass sich keine zwei Kanten in der Ebene schneiden?

Bedeutend ist, dass diese grundlegenden Fragen mit der Graphentheorie gelöst und in unterschiedlichen Anwendungsgebieten verwendet werden können.

Zusammenhängender Graph
Ein Graph ist *zusammenhängend* (engl. *connected graph*), wenn es *zwischen je zwei Knoten einen Weg* gibt.

Nehmen wir eines der ältesten Graphenprobleme als Anschauungsbeispiel für die Mächtigkeit der Graphentheorie:

Das Entscheidungsproblem für Brückentraversierung (Eulerweg)

1736 hat der Mathematiker Leonhard Euler anhand der sieben Brücken von Königsberg herausgefunden, dass nur dann ein Weg existiert, der jede Brücke genau einmal überquert, wenn jeder Knoten einen geraden Grad besitzt.

Grad eines Knoten

Der *Grad* (engl. *degree*) eines Knoten ist die *Anzahl der von ihm ausgehenden Kanten* resp. die Anzahl der zum Knoten inzidenten Kanten.

Das Entscheidungsproblem für einen sogenannten Eulerweg ist also simpel: Ein Graph G ist eulersch, wenn G zusammenhängend ist und jeder Knoten einen geraden Grad besitzt.

In Abb. 2.24 (links) ist ein Straßennetz mit 13 Brücken gegeben. Als Knoten dienen hier Stadtteile, die Kanten symbolisieren Brückenverbindungen unter den Stadtteilen. In diesem Beispiel ist jeder Knoten gerade und wir folgern, dass es einen Eulerschen Weg geben muss. Wie können wir diesen finden?

Der Algorithmus von Fleury aus dem Jahr 1883 gibt eine einfache Anleitung:

Algorithmus von Fleury

1. Wähle einen beliebigen Knoten als aktuellen Knoten aus
2. Wähle eine beliebige inzidente Kante aus und markiere sie (z. B. durch fortlaufende Nummern oder Buchstaben)
3. Wähle den Endknoten als aktuellen Knoten aus
4. Gehe zu Schritt (2)

In Abb. 2.24 (rechts) ist mit der Hilfe des Algorithmus ein Eulerweg bestimmt worden. Natürlich gibt es unterschiedliche Lösungen. Zudem muss der Weg nicht immer in einem Zyklus enden. Wann gelangen wir auf einem Weg wieder zum Startpunkt? Die Antwort lässt sich leicht aus der Abb. 2.24 ableiten, wenn wir die Grade der Knoten näher analysieren.

Mit diesem einfachen Beispiel von Euler wird offensichtlich, dass sich Lösungen mit Hilfe der Graphentheorie für unterschiedliche Konstellationen berechnen lassen. Hier setzen wir einfach voraus, dass ein zusammenhängender Graph beliebiger Komplexität vorliegt. Falls dieser für alle Knoten einen geraden Grad aufweist, gibt es einen Eulerweg.

In Analogie zum Eulerweg geht es beim Hamiltonkreisproblem um die Frage, ob bei einem Graphen alle Knoten anstelle der Kanten genau einmal besucht werden können. Obwohl die Problemstellungen einander ähnlich sind, ist das Hamiltonkreisproblem algorithmisch anspruchsvoll.[5]

[5] Es ist NP-vollständig, d. h. es zählt zur Klasse von Problemen, die sich nicht-deterministisch in Polynomialzeit lösen lassen.

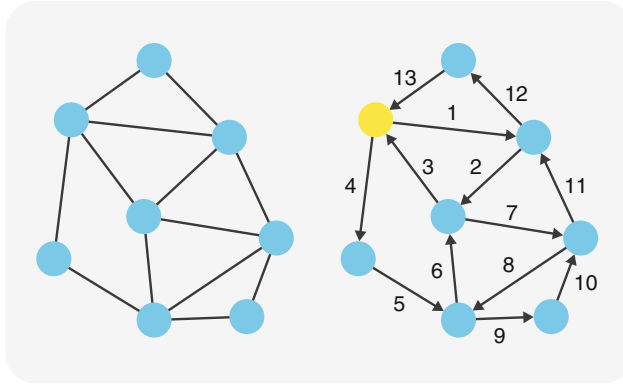

Abb. 2.24 Ein Eulerweg zur Überquerung von 13 Brücken

Berechnung kürzester Wege nach Dijkstra

Edsger W. Dijkstra hat 1959 in einer dreiseitigen Notiz einen Algorithmus beschrieben, um die kürzesten Pfade in einem Netzwerk zu berechnen. Dieser Algorithmus, oft als Dijkstra-Algorithmus bezeichnet, benötigt einen gewichteten Graphen (Gewichte der Kanten: Beispielsweise Wegstrecken mit Maßen wie Laufmeter oder Minuten) und einen Startknoten, um von ihm aus zu einem beliebigen Knoten im Netz die kürzeste Wegstrecke zu berechnen.

Gewichteter Graph

Ein *gewichteter Graph* (engl. *weighted graph*), manchmal attributierter Graph oder Eigenschaftsgraph genannt, ist ein Graph, dessen Knoten oder Kanten mit Eigenschaften (engl. *properties*) versehen sind.

Nehmen wir als Beispiel eines gewichteten Graphen den in Abb. 2.25 gezeigten kantengewichteten Graphen. Dieser Graph repräsentiert ein kleines U-Bahnnetz, wobei die Knoten den Haltestellen und die Kanten den Verbindungsstrecken zwischen den Haltestellen entsprechen. Die ‚Gewichte' der Kanten geben die Distanz zwischen je zwei Haltestellen an, hier mit der Maßeinheit von Kilometern.

Gewicht eines Graphen

Das *Gewicht eines Graphen* (engl. *weight of a graph*) ist die *Summe sämtlicher Gewichte*, d. h. die Summe der Knoten- resp. der Kantengewichte.

Obige Definition kann natürlich für Teilgraphen, Bäume oder Pfade als Teilmengen eines gewichteten Graphen verwendet werden. Was nun interessant ist, ist die Suche nach Teilgraphen mit maximalem oder minimalem Gewicht. Für unser U-Bahn-Beispiel aus

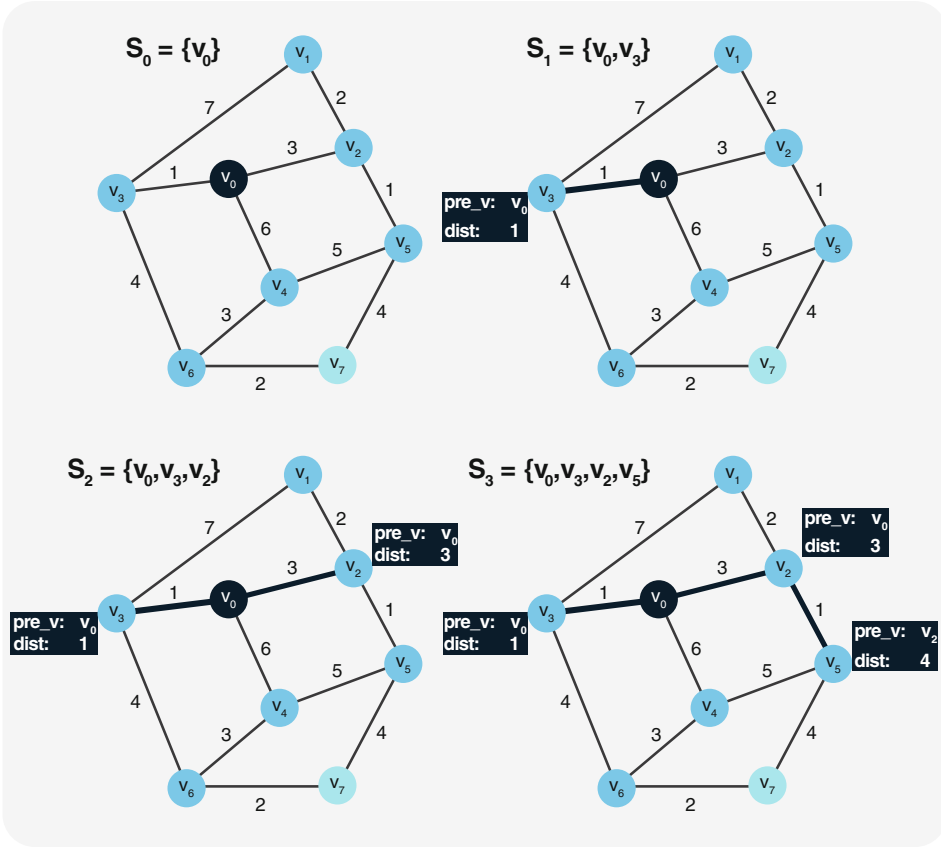

Abb. 2.25 Iteratives Vorgehen zur Erstellung der Menge $S_k(v)$

Abb. 2.25 suchen wir das kleinste Gewicht zwischen den beiden Stationen v_0 und v_7. Mit anderen Worten geht es um den kürzesten Weg, der von der Haltestelle v_0 zur Haltestelle v_7 führt.

Dijkstras Idee zur Lösung des Problems war, denjenigen Kanten zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus versprechen. Setzen wir also einen ungerichteten Graphen G = (V,E) voraus, der für die Kanten positive Gewichte besitzt. Für einen Knoten v_i betrachten wir die zu diesem Knoten inzidenten Knoten v_j und berechnen die Menge $S_k(v)$ wie folgt: Wir greifen denjenigen Nachbarknoten vj heraus, der am nächsten zu v_i liegt und fügen diesen zur Menge $S_k(v)$ hinzu.

Betrachten wir dazu das U-Bahn-Beispiel in Abb. 2.25: Die Ausgangshaltestelle (Startknoten) ist v_0, das Ziel ist die Station v_7. Zur Initialisierung setzen wir die Menge $S_k(v)$ zu

S_0:={v_0}. In einem ersten Schritt $S_1(v)$ besuchen wir nun alle Kanten, die mit v_0 verbunden sind, d. h. v_2, v_3 und v_4. Wir greifen diejenige Kante heraus, die den kürzesten Weg zwischen v_0 und v_i (mit i = 2,3 und 4) verspricht und erhalten folgende Menge: $S_1 = \{v_0,v_3\}$. Entsprechend fahren wir fort und besuchen alle inzidenten Kanten von $S_1(v)$, nämlich v_2, v_4, v_1 und v_6. Da zwischen v_0 und v_2 der kürzeste Weg besteht, nehmen wir den Knoten v_2 in unsere Menge $S_2(v)$ auf und erhalten $S_2 = \{v_0,v_3,v_2\}$. In Analogie erhalten wir $S_3 = \{v_0,v_3,v_2,v_5\}$.

Beim nächsten Schritt $S_4(v)$ können wir entweder den Knoten v_1 oder den Knoten v_6 auswählen, da beide Pfade von v_0 aus einem Maß von 5 Kilometern entsprechen. Wir entscheiden uns für v_1 und erhalten $S_4 = \{v_0,v_3,v_2,v_5,v_1\}$. In analoger Weise werden die nächsten Mengen $S_5 = \{v_0,v_3,v_2,v_5,v_1,v_6\}$ und $S_6 = \{v_0,v_3,v_2,v_5,v_1,v_6,v_4\}$ konstruiert, bis wir $S_7 = \{v_0,v_3,v_2,v_5,v_1,v_6,v_4,v_7\}$ mit dem Zielknoten v_7 erhalten.

Der Start v_0 ist dank der konstruierten Menge $S_7(v)$ mit dem Ziel v_7 verbunden; die entsprechende Strecke entspricht dem kürzesten Weg. Dieser hat die Länge von 7 Kilometern, d. h. 1 Kilometer von v_0 bis v_3, 4 Kilometer von v_3 bis v_6 und 2 Kilometer von v_6 bis v_7. Da die Lösung alle Knoten im U-Bahnnetz enthält, können die kürzesten Wege vom Start v_0 zu allen Stationen v_i mit i = 1,..,7 abgelesen werden.

In Abb. 2.26 ist ersichtlich, dass der Dijkstra-Algorithmus einen Lösungsbaum konstruiert (vgl. fette Verbindungen ausgehend vom Start v_0 resp. Baumstruktur). Die einzelnen Knoten im Baum werden jeweils mit dem Vorgängerknoten (pre_v) und der Gesamtdistanz (dist) annotiert. Beispielsweise wird im Knoten v_5 der Vorgängerknoten v_2 eingetragen sowie die Gesamtdistanz von Kilometern (d. h. 3 plus 1 Kilometer), die vom Weg von v_0 bis v_5 zurückgelegt werden muss.

Der Algorithmus von Dijkstra kann nun für positiv gewichtete Graphen hergeleitet werden. Dabei werden alle Knoten mit den Attributen ‚Vorgängerknoten' und ‚Distanz' (Gesamtdistanz vom Startpunkt zum Knoten) versehen.

Der Algorithmus lautet wie folgt:

Algorithmus von Dijkstra

1. Initialisierung: Trage im Startknoten die Distanz 0 und in allen übrigen Knoten die Distanz unendlich ein. Setze die Menge S_0 := {pre_v: Startknoten, dist: 0}
2. Iteriere S_k, solange es noch unbesuchte Knoten gibt und erweitere in jedem Iterationsschritt die Menge S_k wie folgt:
   - 2a. Berechne für alle Nachbarknoten des aktuellen Knotens die Summe der jeweiligen Kantengewichte.
   - 2b. Wähle denjenigen Nachbarknoten mit der kleinsten Summe aus.
   - 2c. Ist die Summe der Kantengewichte kleiner als der gespeicherte Distanzwert im gefundenen Nachbarknoten, so setze den aktuellen Knoten als Vorgänger (pre_v) und trage die aktuelle Distanz (dist) in Sk ein.

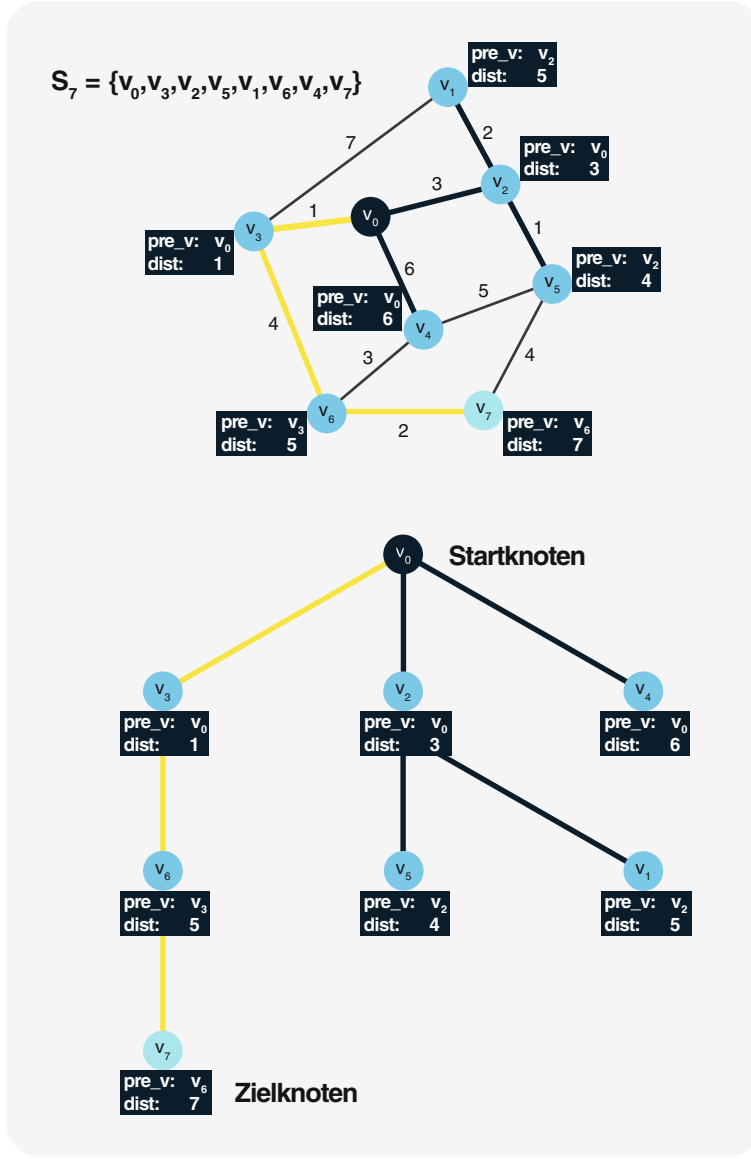

Abb. 2.26 Kürzeste U-Bahnstrecke von Haltestelle v_0 nach v_7

Hier ist ersichtlich, dass immer denjenigen Kanten gefolgt wird, die den kürzesten Streckenabschnitt vom aktuellen Knoten aus garantieren. Andere Kanten resp. Knoten werden erst dann berücksichtigt, wenn alle kürzeren Streckenabschnitte bereits mit einbezogen wurden. Dieser Ansatz garantiert, dass beim Erreichen eines bestimmten Knotens kein kürzerer Weg zu ihm existieren kann (Greedy Algorithmus[6]). Das iterative Vorgehen wird solange wiederholt, bis entweder die Distanz vom Start- zum Zielknoten

bekannt ist oder bis alle Distanzen vom Startknoten zu allen übrigen Knoten berechnet worden sind.

Wo liegt das nächste Postamt?

Eine klassische Frage, die mittels Graphentheorie beantwortet werden kann, ist die Suche nach dem nächsten Postamt. In einer Stadt mit vielen Postämtern (analog dazu Diskotheken, Restaurants oder Kinos etc.) möchte ein Webnutzer das nächste Postamt abrufen. Welches liegt am nächsten zu seinem jetzigen Standort? Dies sind typische Suchfragen, die standortabhängig beantwortet werden sollten (engl. *location-based services*).

Seien n Punkte (Postämter) in der Ebene gegeben, d. h. die Menge $M = \{P_1,P_2,\ldots,P_n\}$, so bestimme man zu einem Anfragepunkt q (Standort des Webnutzers) den nächsten Punkt aus M. Am einfachsten und schnellsten kann die Suchfrage beantwortet werden, wenn man das Stadtgebiet geschickt in Regionen einteilt:

Voronoi-Diagramm

Ein *Voronoi-Diagramm* (engl. *Voronoi diagram*) unterteilt eine Ebene mit einer Menge $M = \{P_1,P_2,\ldots,P_n\}$ von n Punkten in *Äquivalenzklassen*, indem es zu jedem Punkt P_i sein Voronoi-Polygon V_i zuordnet.

Mit der Hilfe von Voronoi-Diagrammen kann die Suche nach dem nächsten Postamt auf das Bestimmen des Voronoi-Polygons zurückgeführt werden, welches den Anfragepunkt enthält (Punktlokalisierung). Alle Punkte innerhalb eines Voronoi-Polygons P_i sind äquivalent in dem Sinne, dass sie näher beim Postamt P_i als bei allen übrigen Postämtern liegen.

Voronoi-Polygon

Für eine Menge $M = \{P_1,P_2,\ldots,P_n\}$ von Punkten ist das *Voronoi-Polygon* (engl. *voronoi polygon*) des Punkte P_i die Menge aller Punkte in der Ebene, die *näher bei P_i liegen als bei P_j* für alle $j \neq i$.

Wie konstruiert man ein Voronoi-Polygon zu einem Punkt P_i? Ganz einfach: Man nimmt die Mittelsenkrechten zu den nächstgelegenen Punkten P_j und bestimmt die Halbräume $H(P_i,P_j)$, die den Punkt P_i einschließen (siehe Abb. 2.27). Der Durchschnitt sämtlicher Halbräume des Punktes P_i bildet das Voronoi-Polygon V_i.

Die Berechnung von Voronoi-Diagrammen kann aufwendig sein, vor allem wenn viele Postämter in einer Großstadt oder Agglomeration existieren. Aus diesem Grund verwendet man Algorithmen, die den Berechnungsaufwand reduzieren. Eine bekannte Methode folgt dem Prinzip Divide et Impera (lateinisch für Teile und Herrsche), indem das

[6] Bei Greedy Algorithmen werden schrittweise Folgezustände ausgewählt, die das beste Ergebnis mit der Hilfe einer Bewertungsfunktion versprechen.

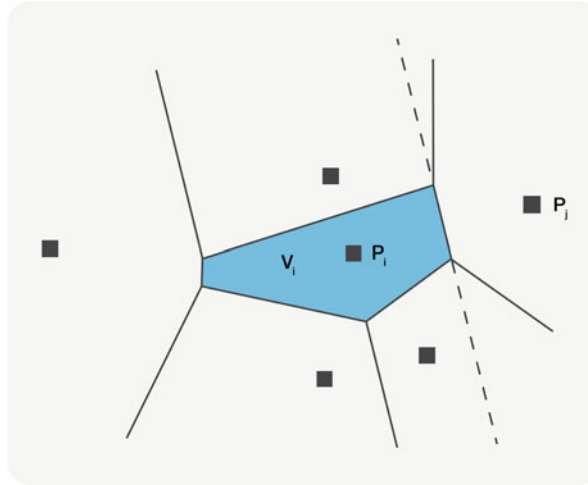

Abb. 2.27 Konstruktion des Voronoi-Polygons durch Halbräume

Problem so lange in Teilprobleme aufgeteilt wird (Divide), bis man diese auf einfache Art und Weise lösen kann (Impera).

Shamos und Hoey haben 1975 einen Algorithmus nach Divide et Impera vorgeschlagen, der die Menge $M = \{P_1, P_2, \ldots, P_n\}$ aufteilt, um rekursiv das Voronoi-Diagramm VD (M) aus den Teilmengen $VD(M_1)$ und $VD(M_2)$ zu erhalten. Der anspruchsvolle Schritt ist, zu zeigen, dass die Vereinigung zweier Teillösungen $VD(M_1)$ und $VD(M_2)$ in linearer Zeit möglich ist.

In Abb. 2.28 zeigen wir den Lösungsansatz zur Vereinigung zweier Teilprobleme. Nehmen wir die Trennlinie T und bezeichnen mit T^+ den rechts der Trennlinie liegenden Teil der Ebene und mit T^- den links von T liegenden Teil. Die Trennlinie selbst wird mit der Hilfe der konvexen Hülle von M_1 und M_2 schrittweise aufgebaut. Aufgrund der Eigenschaft, dass die Voronoi-Polygone konvex sind, beansprucht der Vereinigungsschritt von $VD(M_1)$ mit $VD(M_2)$ einen linearen Aufwand.

Voronoi-Diagramme gehen u. a. auf den russischen Mathematiker Gerogi Feodosjewitsch Woronoi zurück, der 1908 diese Diagramme für den n-dimensionalen Raum verallgemeinerte. Die Berechnung des Voronoi-Diagramms kann auch mit der Hilfe eines zum Diagramm dualen Graphen berechnet werden. Verbindet man alle Zentren der Voronoi-Polygone miteinander, so entsteht eine sogenannte Delaunay-Triangulation, die vom russischen Mathematiker Boris Nikolajewitsch Delone 1934 vorgestellt wurde. Da Voronoi-Diagramme und Delaunay-Triangulationen dual zueinander sind, gelten alle Eigenschaften des Voronoi-Graphen für den dualen Delaunay-Graphen und umgekehrt.

Beide Ansätze, Voronoi-Diagramme wie Delaunay-Triangulationen, werden in vielen wissenschaftlichen Gebieten zur Lösungssuche herangezogen. In der Informatik werden sie für assoziative Suche, Clusterberechnungen, Planung, Simulation, Vermeidung von Roboter-Kollisionen u. Ä. verwendet.

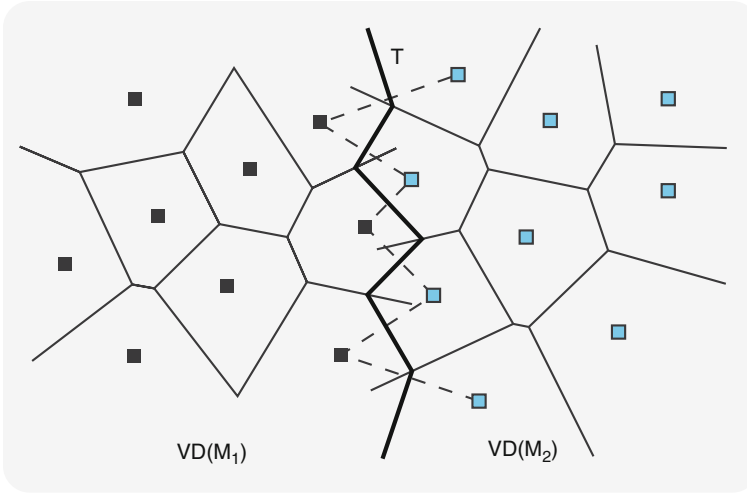

Abb. 2.28 Trennlinie T zweier Voronoi-Diagramme $VD(M_1)$ und $VD(M_2)$

Analyse von Beziehungen in sozialen Netzen

Die Graphentheorie kann dazu benutzt werden, Beziehungen zwischen Mitgliedern einer Gemeinschaft zu analysieren. Ein *soziales Netz* (engl. *social net*) oder eine Gemeinschaft ist eine Gruppe von Webnutzern, die in sozialer Interaktion stehen. Die Gruppe der Mitglieder bestimmt die Teilnehmerschaft und die Bedürfnisse oder Interessen, die durch die Gemeinschaft angestrebt werden.

Unter *Reputation* (engl. *reputation*) wird der Ruf eines sozialen Netzes oder eines einzelnen Mitglieds verstanden. Mit der Hilfe der Soziometrie wird in der empirischen Sozialforschung versucht, die Beziehungen zwischen den Mitgliedern eines sozialen Netzes zu erfassen. Die Graphentheorie mit unterschiedlichen Metriken und Kennzahlen hilft dabei, eine soziologische Netzwerkanalyse durchzuführen.

Soziogramm

Ein *Soziogramm* (engl. *sociogram*) stellt die Beziehungen einer sozialen Gruppe oder eines sozialen Netzes in einem Graphen dar. Die Knoten repräsentieren einzelne Mitglieder, die Kanten drücken *unterschiedliche Einschätzungen* unter den Mitgliedern aus.

Kennzahlen zur Analyse von Soziogrammen orientieren sich an der relativen Position einzelner Akteure und den Relationen unter den Mitgliedern. Die folgenden Kennzahlen stehen im Vordergrund:

- **Grad**: Der Grad (oft Gradzentralität genannt) drückt aus, wie viele Verbindungen von einem Mitglied zu den anderen bestehen; d. h. die Gradzentralität entspricht dem Grad eines Knotens. Bei ausgehenden Kanten wird der äußere Grad (engl. *out degree*), bei eingehenden der innere Grad (engl. *in degree*) berücksichtigt. Mitglieder mit hohem

Grad (engl. *local heroes*) fallen auf, obwohl der Grad wenig über die Bedeutung oder Stellung des Akteurs verrät.

- **Betweenness**: Unter Betweenness Zentralität (engl. *betweenness centrality*) versteht man die Häufigkeit, mit der ein Mitglied auf dem kürzesten Weg (geodätischer Pfad) zwischen zwei Akteuren liegt. Die Kennzahl berücksichtigt nicht nur die direkten, sondern auch die indirekten Beziehungen im sozialen Netzwerk. Zum Beispiel lässt sich berechnen, über wen die meisten Informationen im Netz ausgetauscht werden.
- **Closeness**: Die Kennzahl Closeness Zentralität (engl. *closeness centrality*) misst die Kantenverbindungen eines Mitglieds zu allen Akteuren des Netzwerkes. Oft summiert man die Pfaddistanzen zu allen Mitgliedern und bildet den Durchschnitt.
- **Dichte**: Die Dichte (engl. *density*) eines Netzwerkes ist das Verhältnis der vorhandenen Beziehungen im Vergleich zur maximalen Anzahl möglicher Beziehungen. Der Wert liegt zwischen 0 (keine Beziehungen) und 1 (alle Akteure sind mit allen anderen verbunden).
- **Clique**: Eine Clique ist eine Gruppe von mindesten drei Mitgliedern, die alle miteinander in gegenseitiger Beziehung stehen. Jedes Mitglied der Clique ist mit jedem anderen in Kontakt und umgekehrt.

In Abb. 2.29 ist ein Soziogramm einer Schulklasse gegeben, wobei die Pfeile positive Sympathien unter den Schülerinnen und Schülern ausdrücken. Die hellen Knoten entsprechen Mädchen, die dunklen Knaben. Ein gerichteter Pfeil von v_5 zu v_{11} drückt aus, dass das Mädchen Nr. 5 seine Kollegin Nr. 11 sympathisch findet. Findet im Gegenzug das Mädchen Nr. 11 seine Mitschülerin Nr. 5 ebenfalls sympathisch, so wird die Kante im Graphen als Doppelpfeil gezeichnet.

Bei der Betrachtung des Soziogramms als gerichteter Graph fällt auf, dass nur wenige Mädchen wie Knaben viele Sympathiebezeugungen kriegen. So erhält beispielsweise das Mädchen Nr. 11 (Knoten v_{11}) fünf eingehende Kanten, der Knabe Nr. 7 (v_7) gar deren sechs. Jedenfalls springen die sympathischsten Schülerinnen und Schüler im Graphen des Soziogramms sofort ins Auge.

Adjazenzmatrix

Die *Adjazenzmatrix* (engl. *adjacency matrix*) drückt einen Graphen als Matrix aus und gibt an, *welcher Knoten mit welcher Kante verbunden* ist. Für einen zusammenhängenden Graphen des Grades n ergibt sich eine (nxn)-Matrix.

Neben der Darstellung des Soziogramms als Graph kann dieses also als Matrix ausgedrückt und danach beliebig geordnet werden. In Abb. 2.29 (unten) ist die Adjazenzmatrix (wer ist mit wem in Beziehung) des Schülersoziogramms gegeben. Die beiden Darstellungen sind äquivalent, wobei die Spalten und Zeilen der Matrix teilweise sortiert wurden. Für Schulklassen bestimmten Alters ist es offensichtlich, dass die Knaben primär unter sich (vgl. Quadrant links oben) und die Mädchen ebenfalls unter sich (Quadrant rechts unten) Sympathiebezeugungen aussprechen. Die Clusteranalyse macht diese Verhaltensweise deutlich.

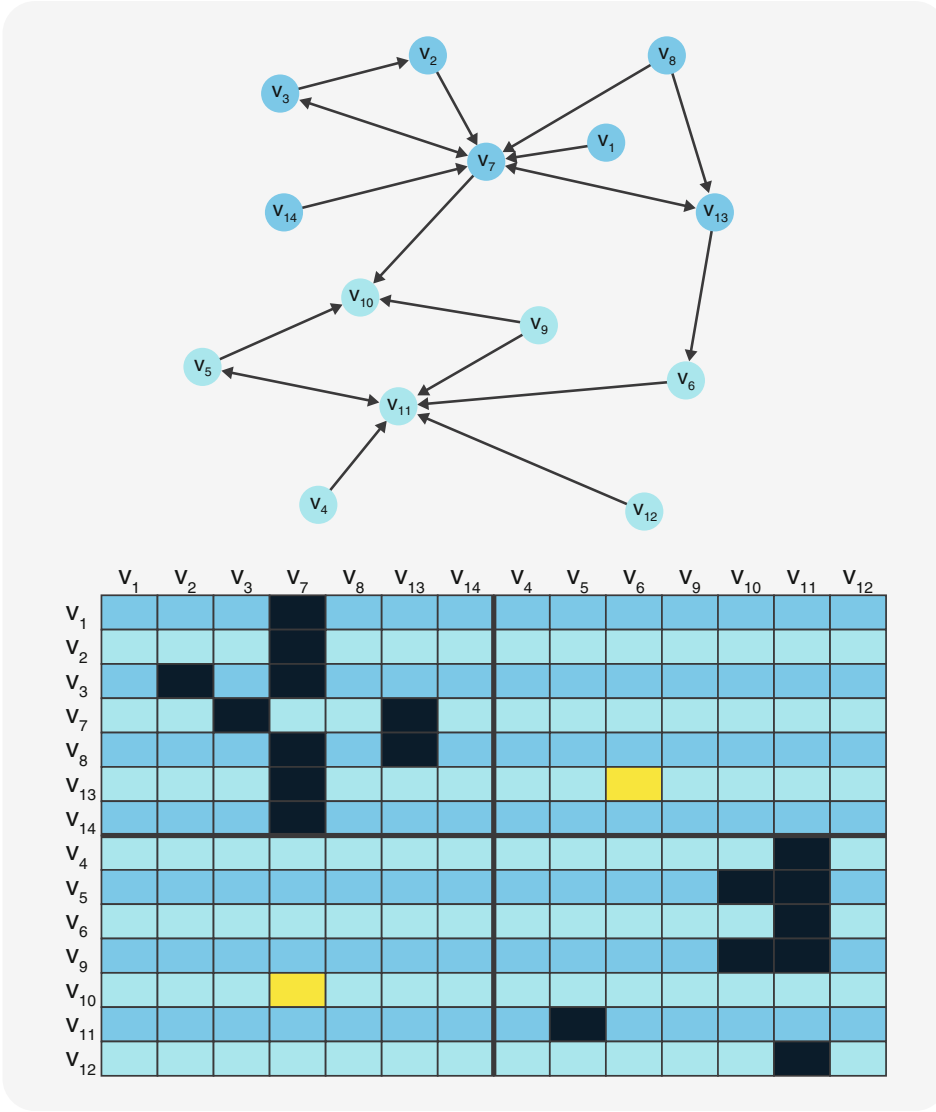

Abb. 2.29 Soziogramm einer Schulklasse als Graph resp. Adjazenzmatrix

Bei der Betrachtung des Soziogramms resp. der Adjazenzmatrix fällt auf, dass je nur ein Mädchen (Quadrant links unten) resp. ein Knabe (Quadrant rechts oben) den Mut hatten, einen Mitschüler des anderen Geschlechts zu bewerten. So bezeugt das Mädchen v_{10} ihre Sympathie dem Knaben v_7 gegenüber, nicht aber umgekehrt.

Die Interpretation von Soziogrammen ist ein breites Anwendungsfeld geworden. Insbesondere lassen sich individuelle Eigenschaften wie gruppendynamische Aspekte analysieren:

- **Star**: Ein Star in einem Soziogramm ist ein Gruppenmitglied, das viele positive Bewertungen (Pfeile) auf sich zieht. Als Maß dient der Knotengrad.
- **Isolat**: Mitglieder, die keine positiven Sympathien in einer Gruppe erheischen, werden als Isolate bezeichnet. In Abb. 2.29 ist der Knabe v_1 oder das Mädchen v_4 ein Isolat, neben weiteren Schülerinnen und Schüler.
- **Geist**: Ein Mitglied einer Gruppe wird als Geist (engl. *ghost*) bezeichnet, wenn es weder positive noch negative Sympathien auf sich ziehen kann. Diese Wortwahl soll ausdrücken, dass diese Mitglieder in der Gemeinschaft sozusagen nicht wahrgenommen werden.
- **Mutual Choice**: Die gegenseitige Sympathiebekennung ist ein wichtiges Merkmal bei Gemeinschaften: Je mehr davon vorliegen, je positiver ist das soziale Klima in der Gruppe. Umgekehrt können negative Beziehungen die Entwicklungsfähigkeit einer Gemeinschaft behindern.
- **Chain**: Eine Kette liegt vor, wenn ein Mitglied ein anderes nominiert, welches ein anderes nominiert etc. Ketten können zu Staren führen.
- **Insel**: Wenn Paare oder Subgruppen von der Gemeinschaft weitgehend abgeschottet sind, spricht man von Inseln.
- **Triade**: Sind drei Mitglieder involviert, die sich gegenseitig bewerten, so liegt eine Triade vor. Bei mehr als drei Mitgliedern mit dieser Eigenschaft spricht man von einem Kreis oder Zirkel.

Die Abb. 2.30 zeigt alle Optionen einer Triadenbildung. Ausgewogenen Konstellationen (engl. *balanced triades*, siehe linke Seite der Abb. 2.30) sind solche, bei welchen sich alle drei Mitglieder untereinander positiv bewerten (Fall B1) oder bei welchen eine positive Paarbewertung und zwei negative vorliegen (Fälle B2, B3 und B4). Die beiden Mitglieder A und B finden sich im Fall B2 gegenseitig sympathisch und gleichzeitig bekunden beide Mühe mit dem Mitglied C, welches umgekehrt ebenfalls keine Sympathien gegenüber A oder B aufbringen kann.

In der Abb. 2.30 rechts sind die unausgewogenen oder un-balanced Triaden U1 bis U4 dargestellt. In den Fällen U1 bis U3 liegen zwar zwei positive Paarbeziehungen vor, doch diese werden durch die dritte negative Paarbeziehung belastet. A in U1 findet die beiden Mitglieder B und C sympathisch (und umgekehrt), doch herrscht zwischen B und C eine negative Stimmung. Da sich B wie C nicht gut verstehen, kann dies die Beziehungen von A zu B resp. von A zu C belasten.

In Gemeinschaften mit bekannten Persönlichkeiten verwendet man oft einfache Kennzahlen, um die Nähe zu dieser Persönlichkeit zu charakterisieren. Zum Beispiel könnte man für Lotfi Zadeh, den Begründer der unscharfen Logik (vgl. Fuzzy Datenbanken in Kap. 6.8), die fiktive Zadeh-Zahl (ZZ) einführen. Diese gibt an, wie nahe man in der Forschergemeinde der Fuzzy Logic beim Urheber liegt. Zadeh selbst kriegt die Zadeh-Zahl 0; Forscher, die mit ihm publiziert haben, erhalten die 1; und Forscher, die mit diesen publiziert haben, bekommen die 2 etc. Demnach würden zur ersten Generation die Zadeh-Schüler Kosko oder Pedrycz ($ZZ = 1$) aufgeführt, da diese mit Lotfi Zadeh publiziert

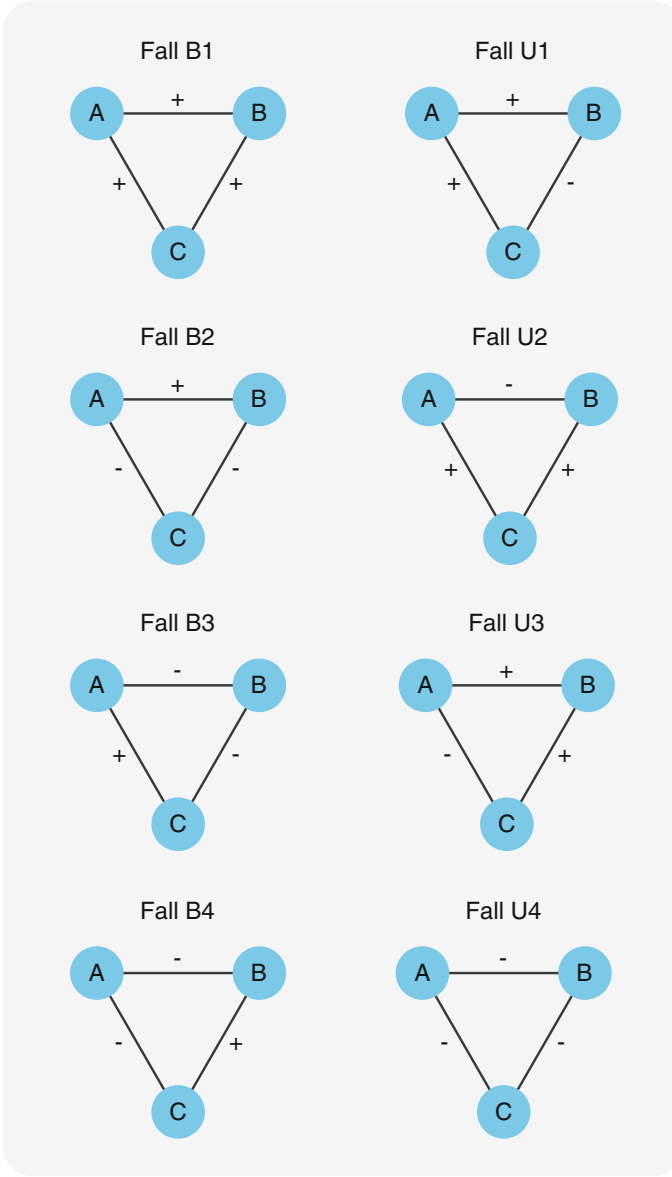

Abb. 2.30 Balancierte (Fall B1 bis B4) und unbalancierte Triaden (U1 bis U4)

haben. Zur zweiten Generation könnte man u. a. Cudré-Mauroux, Meier und Portmann (ZZ = 2) zählen, da diese mit dem Kollegen Pedrycz einen Forschungsbeitrag veröffentlicht haben. Schließlich würden Fasel und Kaufmann zur dritten Generation (ZZ = 3) zählen etc. Bekannt sind solche Zahlen in unterschiedlichen Gemeinschaften; zum

Beispiel existiert in der Schauspieler-Gemeinde die Bacon-Zahl für Schauspieler, die direkt oder indirekt mit Kevin Bacon (Bacon-Zahl 0) einen Film gedreht haben.

Um Graphen oder leistungsfähige Graphalgorithmen in Softwareprogrammen nutzen zu können, bedient man sich meistens Methoden der linearen Algebra. Ist der Graph ungerichtet, so liegt eine symmetrische Adjazenzmatrix vor. Mit der Hilfe der Grade der Knoten lassen sich die entsprechenden Bäume eines Graphen herleiten wie auch andere Eigenschaften. Beispielweise kann die relative Wichtigkeit eines Knotens evaluiert werden, was zur Verlinkung von Webseiten im Internet genutzt wird (Page Ranking).

Zur Bestimmung der Erreichbarkeit von Knoten summiert man die n Potenzen einer Adjazenzmatrix (Einheitsmatrix entspricht nullter Potenz). So erhält man eine Matrix, die für jeden Knoten die Anzahl der Schritte zur Erreichbarkeit dieses Knoten angibt.

2.4.2 Abbildungsregeln für Graphdatenbanken

In Analogie zu den Abbildungsregeln zum Erhalt von Tabellen aus einem Entitäten-Beziehungsmodell (Regeln R1 bis R7) werden in diesem Abschnitt die Regeln G1 bis G7 für eine Graphdatenbank vorgestellt. Die Frage lautet, wie werden Entitäts- und Beziehungsmengen in Knoten und Kanten eines Graphen überführt?

In Abb. 2.31 ist das bekannte Entitäten-Beziehungsmodell für das Projektmanagement gegeben (vgl. Abb. 2.4 resp. 2.16). Die erste Abbildungsregel G1 widmet sich der Überführung von Entitätsmengen in Knoten und lautet wie folgt:

Regel G1 (Entitätsmengen)
Jede *Entitätsmenge muss als eigenständiger Knoten* in der Graphdatenbank definiert werden. Die Merkmale der Entitätsmengen werden als Eigenschaften der Knoten geführt.

In Abb. 2.31 werden die Entitätsmengen ABTEILUNG, MITARBEITER und PROJEKT in den entsprechenden Knoten der Graphdatenbank abgebildet, wobei die Merkmale den Knoten angehängt werden (attributierte Knoten).

Die Regel G2 überführt Beziehungsmengen in Kanten:

Regel G2 (Beziehungsmengen)
Jede *Beziehungsmenge kann als ungerichtete Kante* in der Graphdatenbank definiert werden. Eigenschaften der Beziehungsmengen werden den Kanten zugeordnet (attributierte Kanten).

Wenden wir die Regel G2 auf die Beziehungsmengen ABTEILUNGSLEITER, UNTERSTELLUNG und ZUGEHÖRIGKEIT an, so erhalten wir die folgenden Kantenkonstellationen: ABTEILUNGSLEITER und UNTERSTELLUNG zwischen den Knoten A (für ABTEILUNG) und M (für MITARBEITER) sowie ZUGEHÖRIGKEIT zwischen den Knoten M und P (für PROJEKT).

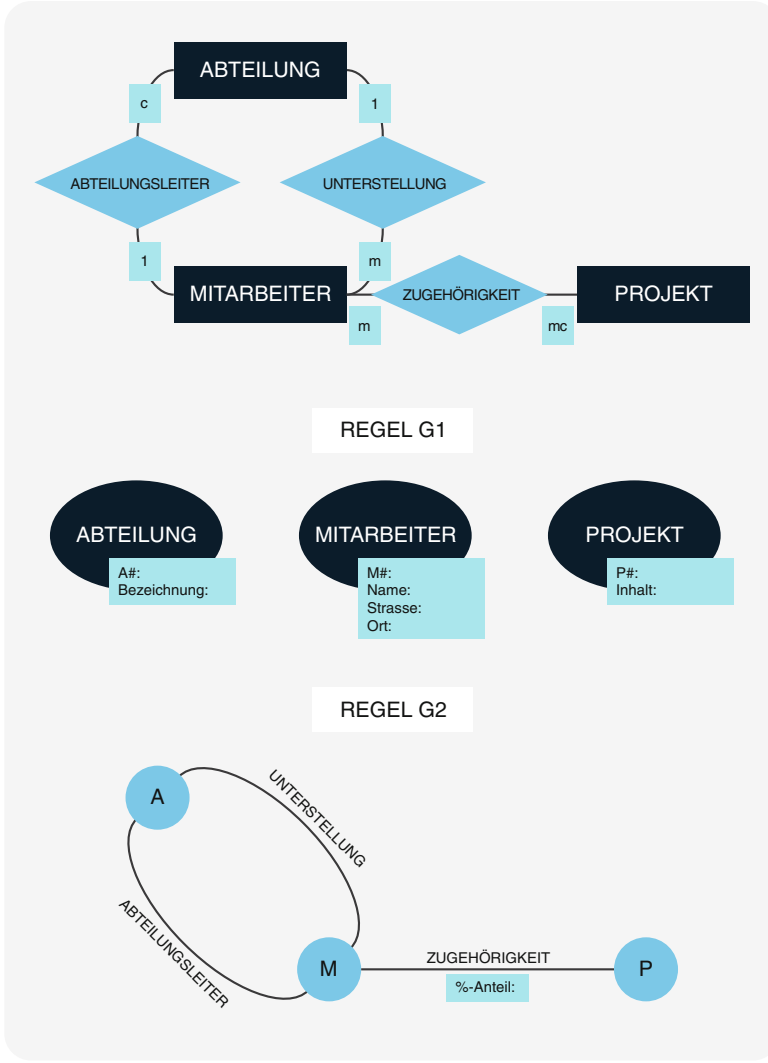

Abb. 2.31 Abbildung von Entitäts- und Beziehungsmengen auf Graphen

Beziehungsmengen können als gerichtete Kanten dargestellt werden. In den folgenden Abbildungsregeln G3 (für netzwerkartige Beziehungsmengen), G4 (hierarchische) und G5 (einfach-einfache) konzentrieren wir uns auf gerichtete Kantenkonstellationen. Diese gelangen zur Anwendung, falls man eine bestimmte Assoziation einer Beziehung resp. die Richtung der entsprechenden Kante hervorheben möchte.

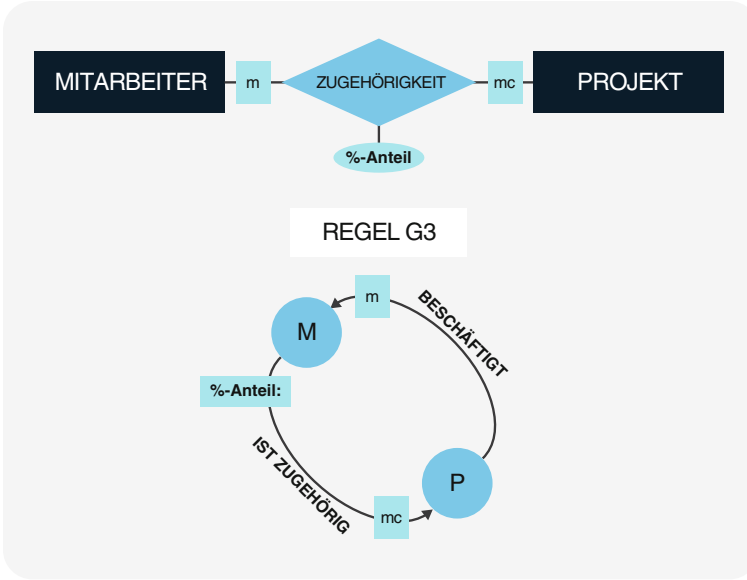

Abb. 2.32 Abbildungsregel für netzwerkartige Beziehungsmengen

Abbildungsregeln für Beziehungsmengen

Zuerst werden komplex-komplexe bzw. netzwerkartige Beziehungsmengen analysiert. In Abb. 2.32 ist die Regel G3 für solche Konstellationen dargestellt.

Regel G3 (netzwerkartige Beziehungsmengen)
Jede *komplex-komplexe Beziehungsmenge* kann durch *zwei gerichtete Kanten* ausgedrückt werden, indem die Assoziationen der Beziehung den Namen der Kante liefern und die entsprechenden *Assoziationstypen an den Pfeilspitzen* annotiert werden. Eigenschaften der Beziehungsmengen können einer oder beiden Kanten angehängt werden.

Die Regel G3 wird in Abb. 2.32 auf die Beziehungsmenge der Projektzugehörigkeiten appliziert, indem die netzwerkartige Beziehungsmenge ZUGEHÖRIGKEIT durch die beiden Kanten IST_ZUGEHÖRIG und BESCHÄFTIGT ausgedrückt wird. Die erste Kante führt von den Mitarbeitenden (M) zu den Projekten (P) und enthält die Eigenschaft ‚%-Anteil', d. h. den Beschäftigungsrad der jeweiligen Mitarbeitenden im zugeordneten Projekt. Da nicht alle Mitarbeitenden an Projekten arbeiten, wird der Assoziationstyp ‚mc' an die Pfeilspitze gehängt. Die Kante BESCHÄFTIGT führt von den Projekten (P) zu den Mitarbeitenden (M) und kriegt den Assoziationstypen m. Analog zu den beiden gerichteten Kanten könnte ein Doppelpfeil zwischen den Knoten M und P etabliert werden, beispielsweise mit dem Namen ZUGEHÖRIGKEIT und dem Merkmal %-Anteil.

Falls erwünscht, ist es erlaubt, für netzwerkartige Beziehungsmengen einen eigenständigen Knoten zu definieren. Gegenüber dem Relationenmodell fällt auf, dass

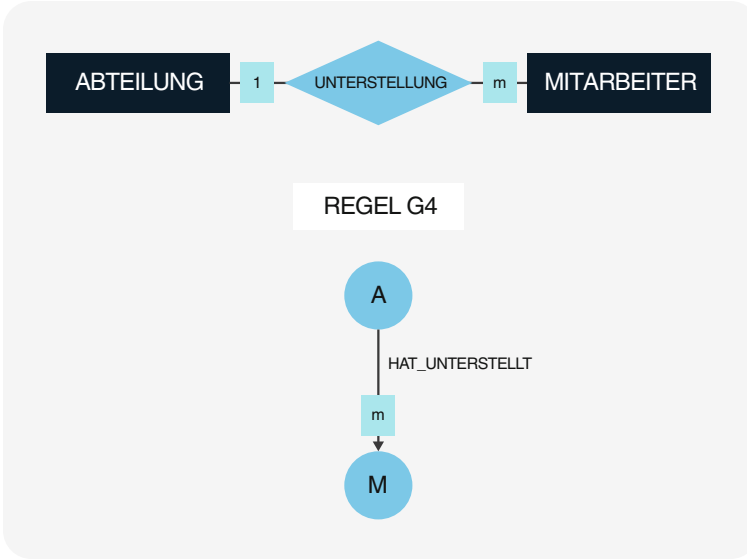

Abb. 2.33 Abbildungsregel für hierarchische Beziehungsmengen

ein Graphenmodell eine Vielzahl von Optionen für Entitäts- und Beziehungsmengen zulässt: ungerichteter Graph, gerichteter Graph, Beziehungsmengen als Kanten, Beziehungsmengen als Knoten etc. Die Regeln G3, G4 und G5 suggerieren hingegen, für Beziehungsmengen gerichtete Kanten zu verwenden. Damit bezwecken wir, die Graphdatenbank möglichst einfach und verständlich zu definieren, damit deskriptive Abfragesprachen für Graphen vom gelegentlichen Nutzer intuitiv eingesetzt werden können.

Regel G4 (hierarchische Beziehungsmengen)
Eine *einfach-komplexe Beziehungsmenge* kann als *gerichtete Kante* zwischen den Knoten etabliert werden, indem die *Richtung vom Wurzelknoten zum Blattknoten* gewählt und der *mehrfache Assoziationstyp* (m oder mc) an der Pfeilspitze annotiert wird.

In Abb. 2.33 ist die hierarchische Unterstellung der Mitarbeitenden einer Abteilung aufgezeigt. Die gerichtete Kante HAT_UNTERSTELLT führt vom Wurzelknoten A (für ABTEILUNG) zum Blattknoten M (MITARBEITER). Zudem wird der Assoziationstyp m am Pfeilende angehängt, da jede Abteilung mehrere Mitarbeitende beschäftigt. Alternativ dazu könnte die Kante IST_UNTERSTELLT von den Mitarbeitenden zur Abteilung führen, dann allerdings mit dem Assoziationstypen 1, da jeder Mitarbeitende genau in einer Abteilung arbeitet.

Nun bleibt noch der Fall für einfach-einfache Beziehungsmengen:

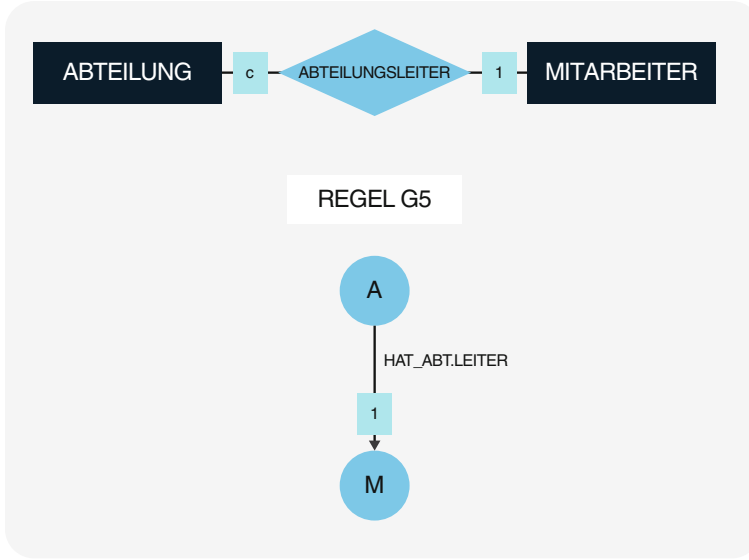

Abb. 2.34 Abbildungsregel für einfach-einfache Beziehungsmengen

Regel G5 (einfach-einfache Beziehungsmengen)
Jede *einfach-einfache Beziehungsmenge* kann als *gerichtete Kante* zwischen den entsprechenden Knoten etabliert werden. Dabei soll die Richtung so gewählt werden, dass der *Assoziationstyp* an der Pfeilspitze *nach Möglichkeit eindeutig* ist.

Die Abb. 2.34 zeigt das Beispiel für die Bestimmung der Abteilungsleiter: Die Beziehungsmenge ABTEILUNGSLEITER wird zur gerichteten Kante HAT_ABTEILUNGSLEITER, die vom Knoten ABTEILUNG (abgekürzt durch A) zum Knoten MITARBEITER (M) führt. Dabei wird die Pfeilspitze mit ‚1' annotiert, da jede Abteilung genau eine Abteilungsleiterin oder einen Abteilungsleiter besitzt. Selbstverständlich wäre als Alternative die umgekehrte Richtung von den Mitarbeitenden zur Abteilung ebenfalls zulässig, indem man eine Kante IST_ABTEILUNGSLEITER einführen und die Pfeilspitze mit dem Assoziationstypen ‚c' versehen würde.

Das Graphenmodell ist flexibel und bietet viele Optionen, da keine Normalformen die Wahl einschränken. Nachteilig kann sein, dass die Anwendenden diese Freiheiten zu extensiv nutzen, was zu komplexen, eventuell redundanten Graphkonstellationen führen kann. Die vorgeschlagenen Abbildungsregeln für Entitätsmengen (G1), Beziehungsmengen (G2, G3, G4 und G5) sowie diejenigen für Generalisation (G6) und Aggregation (G7) sollen deshalb eine Leitlinie darstellen, von der im Einzelfall und abhängig von der Anwendung abgewichen werden kann.

Abbildungsregeln für Generalisation und Aggregation

Im Folgenden werden Generalisationshierarchien sowie Aggregationsnetze oder -hierarchien auf Graphen abgebildet.

Regel G6 (Generalisation)
Die *Superentitätsmenge* der Generalisation wird zu einem *Doppelknoten*, die *Subentitätsmengen* zu normalen *Knoten*. Dabei wird die Generalisierungshierarchie durch *Spezialisierungskanten* ergänzt.

Die Abb. 2.35 zeigt die Spezialisierung der Mitarbeitenden als Führungskräfte, Fachspezialisten oder Lehrlinge. Der Knoten M wird als Doppelknoten dargestellt; dieser repräsentiert die Superentitätsmenge MITARBEITER. Die Subentitätsmengen FK (als Abkürzung für FÜHRUNGSKRAFT), FS (FACHSPEZIALIST) und L (LEHRLING) werden zu Knoten, wobei je eine Kante von M zu den Knoten FK, FS und L gezogen werden. Die Namen der Kanten lauten IST_FÜHRUNGSKRAFT, IST_FACHSPEZIALIST resp. IST_LEHRLING.

Das Graphenmodell vermag Generalisierungshierarchien elegant darzustellen, da die involvierten Entitätsmengen als Knoten und die IS_A-Beziehungen als Kanten auftreten. Im Gegensatz zum Relationenmodell müssen hier nicht künstliche Attribute (vgl. Merkmal „Kategorie“ in Abschn. 2.3.2) eingeführt werden. Zudem ist nicht notwendig, einen Primärschlüssel für den Doppelknoten zu definieren und denselben als Primärschlüssel in den Teilknoten zu verwenden.

Regel G7 (Aggregation)
Bei einer *netzwerkartigen oder hierarchischen Aggregationsstruktur* wird die *Entitätsmenge als Knoten* und die *Beziehungsmenge als Kante* dargestellt, wobei die Pfeilspitze mit dem Assoziationstypen mc annotiert wird. Eigenschaften der Knoten werden bei den Knoten, Eigenschaften der Kanten bei den Kanten angehängt.

In der Abb. 2.36 ist ein Firmenkonglomerat gegeben: Die Entitätsmenge FIRMA wird zum Knoten FIRMA, die Beziehungsmenge KONZERNSTRUKTUR zur gerichteten Kante IST_TOCHTER, wobei die Pfeilspitze mit mc annotiert wird. Der Assoziationstyp mc drückt aus, dass jede Firma eventuell an unterschiedlichen Töchtern beteiligt ist. Die Eigenschaften der Entitäts- und Beziehungsmenge gehen über in die Eigenschaften des Knotens und der Kante.

Die Abbildungsregel G7 vermag auch hierarchische Aggregationsstrukturen abzubilden. In Abb. 2.37 ist die Stückliste gegeben, die bereits in den Abschn. 2.2.3 resp. 2.3.2 diskutiert wurden. Die Entitätsmenge ARTIKEL wird zum Knoten ARTIKEL, die Beziehungsmenge STÜCKLISTE wird zur gerichteten Kante ZERFÄLLT_IN. Die Eigenschaften der Artikel wie der Stückliste werden am Knoten resp. an der Kante angehängt.

Es ist offensichtlich, dass sowohl Generalisationshierarchien wie Aggregationsstrukturen (Netze, Bäume) auf einfache Art und Weise in einer Graphdatenbank gespeichert werden. Im Gegensatz zu relationalen Datenbanken haben Graphdatenbanken den Vorteil, dass sich die vielfältigsten Strukturen wie Netze, Bäume oder Kettenfolgen direkt in Teilgraphen verwalten lassen.

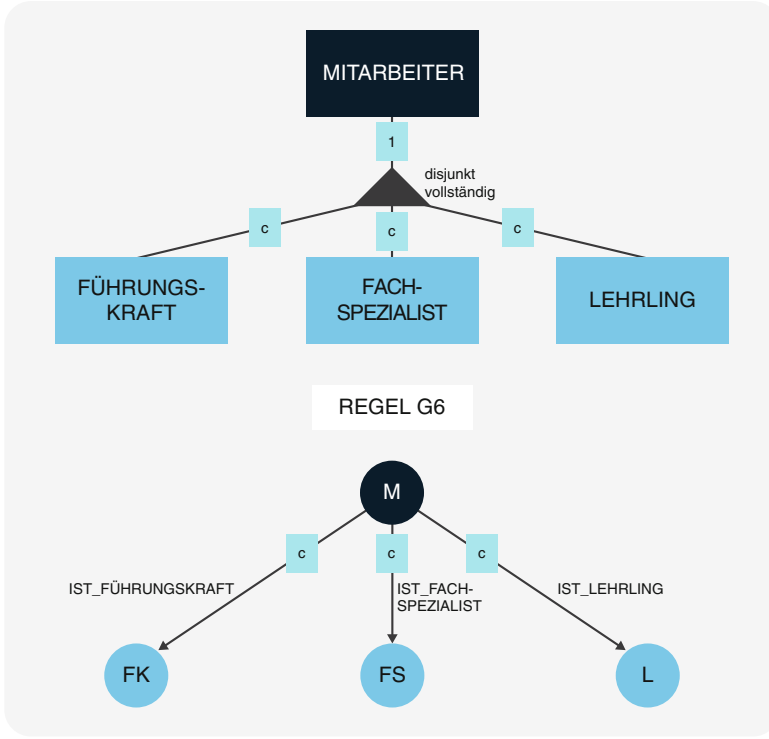

Abb. 2.35 Generalisation als baumartiger Teilgraph

2.4.3 Strukturelle Integritätsbedingungen

Sowohl bei relationalen Datenbanken (vgl. Abschn. 2.3.3) wie bei Graphdatenbanken existieren strukturelle Integritätsbedingungen. Darunter versteht man, dass Grapheigenschaften vom Datenbanksystem selbst garantiert werden. Wichtige Konsistenzbedingungen sind:

- **Eindeutigkeitsbedingung**: Jeder Knoten und jede Kante kann im Graphen eindeutig identifiziert werden. Mit sogenannten Pfadausdrücken (siehe Kap. 3) können beliebige Kanten oder Knoten aufgesucht werden.
- **Wertebereichsbedingung**: Die Merkmale der Knoten sowie die Merkmale der Kanten unterliegen den spezifizierten Datentypen, d. h. sie stammen aus wohldefinierten Wertebereichen.
- **Zusammenhang**: Ein Graph wird zusammenhängend genannt, wenn es zu je zwei Knoten im Graphen einen Pfad gibt. Die Graphdatenbank garantiert den Zusammenhang für Graphen oder Teilgraphen.

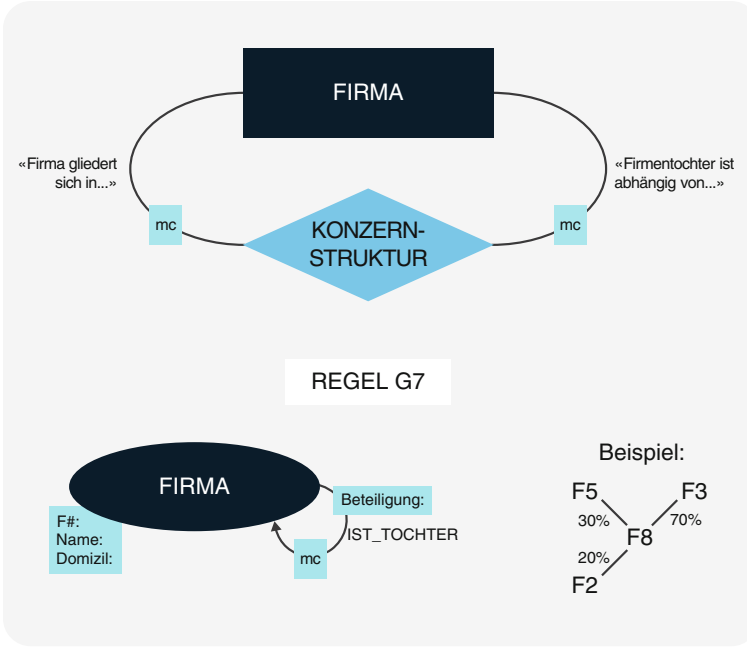

Abb. 2.36 Netzwerkartige Firmenstruktur als Graph

- **Baumstrukturen**: Spezielle Graphen wie Bäume können von der Graphdatenbank verwaltet werden. Dabei wird garantiert, dass die Baumstruktur bei der Veränderung von Knoten oder Kanten erhalten bleibt.
- **Dualität**: Sei G = (V,E) ein planarer Graph.[7] Der zu G duale Graph G* = (V*,E*) wird dadurch erhalten, dass man zu jeder Fläche des Graphen G einen Knoten im dualen Graphen G* platziert. Danach verbindet man die so erhaltenen Knoten V* miteinander und erhält die Kanten E*. Zwei Knoten in G* sind durch genauso viele Kanten verbunden, wie die entsprechenden Flächen aus G gemeinsame Kanten aufweisen. Eine Graphdatenbank kann einen planaren Graphen G in den dualen Graphen G* überführen und umgekehrt.

Zusammenhänge, Baumstrukturen und Dualitätsprinzipien sind wichtige Eigenschaften in der Graphentheorie. Liegt eine Dualität vor, so kann gefolgert werden, dass wahre Aussagen eines Graphen in wahre Aussagen des dualen Graphen übergehen und umgekehrt. Dieses wichtige mathematische Grundprinzip macht man sich in der Informatik zunutze, indem man Problemstellungen in den dualen Raum verlagert, weil dort ein

[7] Eine Darstellung eines Graphen in der Ebene, der keine Kantenüberschneidungen aufweist, wird planarer Graph genannt.

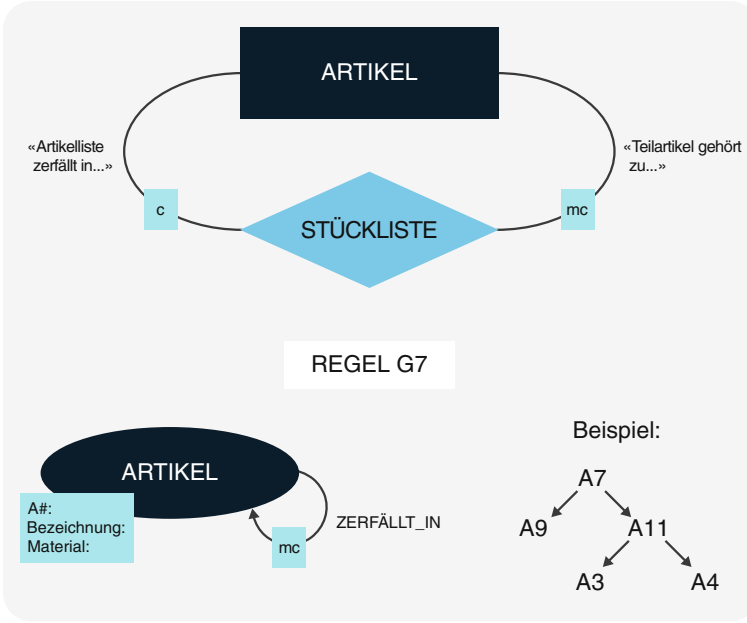

Abb. 2.37 Hierarchische Artikelstruktur als baumartiger Teilgraph

Lösungsansatz leichter zu finden ist. Können Eigenschaften im dualen Raum hergeleitet werden, so gelten diese auch im Ursprungsraum.

Ein kleines Beispiel dazu: Für das Postamtproblem (Abschn. 2.4.1) ging es darum, für jedes Postamt das Voronoi-Polygon zu konstruieren. Dieses erhält man, indem man alle Halbräume der jeweils konstruierten Mittelsenkrechten zwischen benachbarten Postämtern miteinander schneidet. Der ‚Durchschnitt von Halbräumen' kann im dualen Raum (Geraden in der Ebene werden als Punkte aufgefasst) auf das Problem der ‚Bestimmung der konvexen Hülle einer Punktmenge' zurückgeführt werden.[8] Da es effiziente Algorithmen für die Berechnung der konvexen Hüllen gibt, ist der Halbraumschnitt bereits gelöst.

2.5 Unternehmensweite Datenarchitektur

Verschiedene Untersuchungen haben offengelegt, dass während der Definitions- und Aufbauphase eines Informationssystems die künftigen Anwendenden zwar komplexe Funktionen und ausgeklügelte Verfahren priorisiert verlangen, beim Gebrauch der An-

[8] Kevin Brown hat sich 1979 in seiner Dissertation über ‚Geometric Transformations for Fast Geometric Algorithms' duale Räume zu Nutze gemacht; die hier angesprochene Idee zur Konstruktion von Voronoi-Polygonen stammt von ihm.

wendungssysteme hingegen der *Aussagekraft der Daten* (Abschn. 1.5) größeres Gewicht beimessen. Die Datenarchitekten tun deshalb gut daran, wenn sie sich gleich zu Beginn die folgenden Fragen stellen:

Welche Daten sollen in Eigenregie gesammelt, welche von externen Datenlieferanten bezogen werden? Wie lassen sich unter Berücksichtigung nationaler oder internationaler Gegebenheiten die Datenbestände klassifizieren und strukturieren? Wer ist verantwortlich für Unterhalt und Pflege der geografisch verstreuten Daten? Welche Auflagen bestehen im internationalen Umfeld bezüglich Datenschutz und Datensicherheit? Welche Rechte und Pflichten gelten für Datenaustausch und -weitergabe? Dies sind genau die Fragestellungen, die die Bedeutung der Datenarchitektur für das Unternehmen untermauern und die entsprechenden Datenmodelle in den Brennpunkt der Betrachtung rücken.

Aufgrund überbordender Anforderungen von Anwenderseite her werden heute die Analyse- und Entwurfsarbeiten meistens nur für einzelne Zusatzfunktionen oder bestenfalls für bestimmte Anwendungsgebiete vorangetrieben. Dieses Vorgehen birgt beim Einsatz von SQL- und NoSQL-Datenbanken die Gefahr in sich, dass eine Fülle von Datenbeständen lediglich ad hoc oder nur aus einem lokalen Anwendungsbedarf heraus definiert werden. Eine *unkontrollierte Inflation von SQL- und NoSQL-Datenbanken* mit sich gegenseitig überlappenden oder mehrdeutigen Datenbeständen ist die Folge und führt zwangsläufig zu einem Datenchaos. Anwendungsübergreifende Auswertungen können kaum oder nur mit großem Aufwand vorgenommen werden.

Eine *unternehmensweite Datenarchitektur* (engl. *corporate-wide data architecture*) schafft hier Abhilfe, da sie die wichtigsten Entitätsmengen und Beziehungen aus langfristiger und umfassender Sicht des Unternehmens beschreibt. Sie soll einzelnen Geschäftsfeldern und lokalen Datensichten übergeordnet bleiben und das Verständnis für die Gesamtzusammenhänge des Unternehmens fördern. Diese Datenarchitektur und davon abgeleitete Datenmodelle bilden die Grundlage für eine abgestimmte Entwicklung von Informationssystemen.

Die Abb. 2.38 stellt den Zusammenhang zwischen den bereichsübergreifenden und den anwendungsspezifischen Datenmodellen schematisch dar. Die unternehmensweite Datenarchitektur beschreibt die für das Unternehmen notwendigen Datenklassen und ihre Beziehungen. Daraus abgeleitet werden geschäftsbereichsspezifische Datenmodelle entwickelt, die pro Anwendung zu konzeptionellen Datenbankschemas führen. Solche Verfeinerungsschritte lassen sich in der Praxis natürlich nicht stur nach der Methode Top Down vollziehen, da die hierfür aufzubringende Zeit fehlt. Bei Änderungen bestehender Informationssysteme oder bei der Entwicklung neuer Anwendungen werden die konzeptionellen Datenbankschemas vielmehr nach der Methode Bottom Up mit den teils lückenhaft vorhandenen Geschäftsbereichsdatenmodellen und mit der unternehmensweiten Datenarchitektur abgestimmt und dadurch schrittweise auf die längerfristige Unternehmensentwicklung hin ausgerichtet.

Neben den in Eigenregie zu entwickelnden Geschäftsbereichsdatenmodellen existieren *Branchendatenmodelle,* die auf dem Softwaremarkt käuflich erworben werden können. Bei der Nutzung solcher Standardisierungsbemühungen reduziert sich der Aufwand für

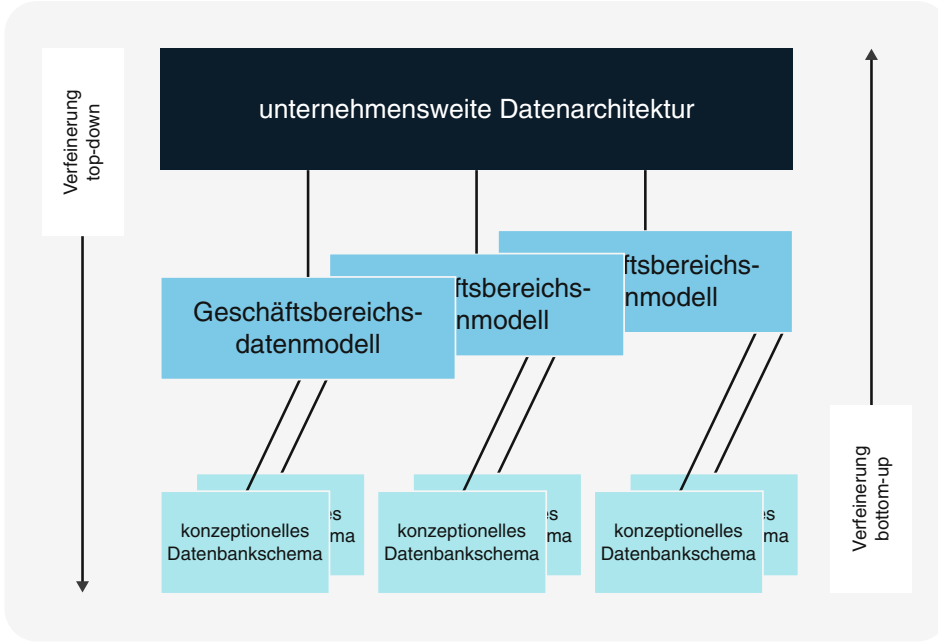

Abb. 2.38 Abstraktionsstufen der unternehmensweiten Datenarchitektur

die Integration käuflich erworbener Anwendungssoftware. Gleichzeitig vereinfachen Branchenmodelle über Konzernbereiche oder Firmen hinweg den Austausch von Informationen.

Wir beschreiben exemplarisch und lückenhaft eine unternehmensweite Datenarchitektur, indem wir uns auf die Entitätsmengen PARTNER, ROHSTOFF, PRODUKT, KONTRAKT und GESCHÄFTSFALL beschränken (vgl. Abb. 2.39):

- **PARTNER**: Darunter versteht man alle natürlichen und juristischen Personen, an denen das Unternehmen Interesse zeigt und über die für die Abwicklung der Geschäfte Informationen benötigt werden. Insbesondere zählen Kunden, Mitarbeitende, Lieferanten, Aktionäre, öffentlich-rechtliche Körperschaften, Institutionen und Firmen zur Entitätsmenge PARTNER.
- **ROHSTOFF**: Mit ROHSTOFF werden Rohwaren, Metalle, Devisen, Wertschriften oder Immobilien bezeichnet, die der Markt anbietet und die im Unternehmen eingekauft, gehandelt oder veredelt werden. Allgemein können sich solche Güter auf materielle wie auf immaterielle Werte beziehen. Beispielsweise könnte sich eine Beratungsfirma mit bestimmten Techniken und entsprechendem Know-how eindecken.
- **PRODUKT**: Damit wird die Produkt- oder Dienstleistungspalette des Unternehmens definiert. Auch hier können je nach Branche die produzierten Artikel materiell oder immateriell sein. Der Unterschied zur Entitätsmenge ROHSTOFF liegt darin, dass mit

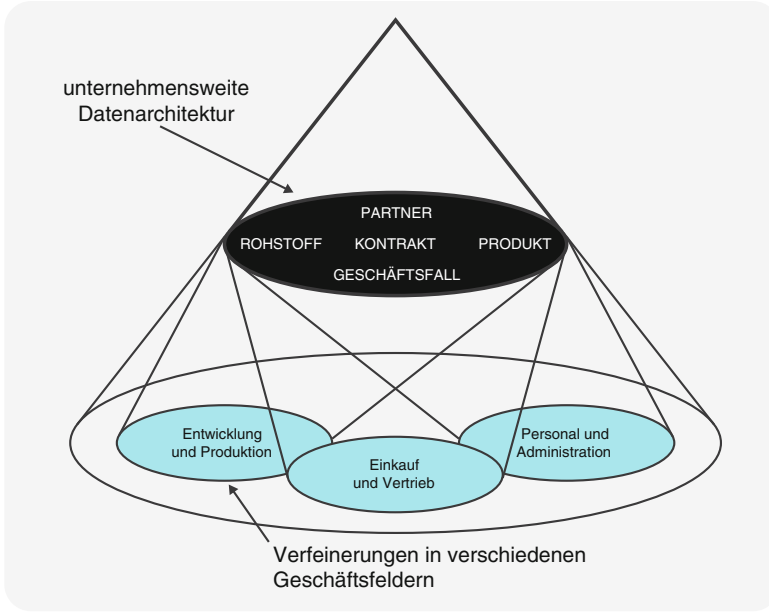

Abb. 2.39 Datenorientierte Sicht der Geschäftsfelder

PRODUKT die unternehmensspezifische Entwicklung und Herstellung von Waren oder Dienstleistungen charakterisiert wird.

- **KONTRAKT**: Unter einem KONTRAKT wird eine rechtlich verbindliche Übereinkunft verstanden. Zu dieser Entitätsmenge zählen sowohl Versicherungs-, Verwaltungs- und Finanzierungsvereinbarungen wie auch Handels-, Beratungs-, Lizenz- und Verkaufsverträge.
- **GESCHÄFTSFALL**: Ein GESCHÄFTSFALL bedeutet innerhalb einer Kontraktabwicklung einen einzelnen geschäftsrelevanten Schritt; ein Vorkommnis. Das kann zum Beispiel eine einzelne Zahlung, eine Buchung, eine Faktur oder eine Lieferung sein. Die Entitätsmenge GESCHÄFTSFALL hält die Bewegungen auf obigen Entitätsmengen fest.

Neben den verallgemeinerten Entitätsmengen PARTNER, ROHSTOFF, PRODUKT, KONTRAKT und GESCHÄFTSFALL müssen *die wichtigsten Beziehungen aus der Sicht des Unternehmens* charakterisiert werden. So lässt sich beispielsweise festlegen, welche Rohwaren über welche Partner bezogen werden (engl. *supply chain management*), von wem welche firmeneigenen Produkte produziert werden oder welche Konditionen in einem Kontrakt bezüglich Partner oder Waren gelten sollen.

Selbstverständlich können mit einem solch groben Ansatz einer unternehmensweiten Datenarchitektur noch keine Informationssysteme aufgebaut oder entwickelt werden.

Vielmehr müssen die Entitätsmengen und ihre Beziehungen schrittweise verfeinert werden, und zwar anhand einzelner Geschäftsfelder oder bestimmter Anwendungsbereiche. Bei diesem datenorientierten Vorgehen ist wesentlich, dass jede Konkretisierung eines *anwendungsspezifischen Datenmodells in Abstimmung mit der unternehmensweiten Datenarchitektur* vorgenommen wird. Nur so kann die Entwicklung von Informationssystemen kontrolliert und im Sinne der längerfristigen Ziele des Unternehmens realisiert werden.

2.6 Rezept zum Datenbankentwurf

In diesem Abschnitt fassen wir unsere Erkenntnisse zur Datenmodellierung in einem Vorgehensplan rezeptartig zusammen. Die Abb. 2.40 zeigt die zehn Entwurfsschritte, die je nach Phase der Projektentwicklung unterschiedlich durchlaufen werden. Praktische Erfahrungen zeigen, dass in jeder *Vorstudie* die Datenanalyse mit einem groben Entitäten-Beziehungsmodell erarbeitet werden sollte. Im *Grob- oder Detailkonzept* werden dann die Analyseschritte verfeinert, eine SQL- oder NoSQL-Datenbank oder eine Kombination derselben erstellt und auf Konsistenz und Implementierungsaspekte hin untersucht.

Die Entwurfsschritte lassen sich wie folgt charakterisieren: Zuerst müssen bei der Datenanalyse die *relevanten Informationssachverhalte* schriftlich in einer Liste festgehalten werden. In den weiteren Entwurfsschritten kann die Liste in Abstimmung mit den künftigen Anwendenden ergänzt und verfeinert werden, da das Entwurfsverfahren ein iterativer Prozess ist. Im zweiten Schritt werden die *Entitäts- und Beziehungsmengen* bestimmt sowie ihre Identifikationsschlüssel und Merkmalskategorien festgelegt. Das erhaltene Entitäten-Beziehungsmodell wird vervollständigt, indem die verschiedenen Assoziationstypen eingetragen werden. Nun können auch *Generalisationshierarchien und Aggregationsstrukturen* speziell im dritten Schritt hervorgehoben werden. Im vierten Schritt wird das Entitäten-Beziehungsmodell mit der *unternehmensweiten Datenarchitektur* verglichen und abgestimmt, damit die Weiterentwicklung von Informationssystemen koordiniert und den längerfristigen Zielen entsprechend vorangetrieben werden kann.

Mit dem fünften Schritt wird das Entitäten-Beziehungsmodell in eine *SQL- und/oder NoSQL-Datenbank* überführt. Dabei werden die erläuterten *Abbildungsregeln* R1 bis R7 resp. G1 bis G7 für Entitätsmengen, Beziehungsmengen, Generalisation und Aggregation benutzt. Im sechsten Schritt wird der Datenbankentwurf verfeinert, indem Integritätsbedingungen formuliert werden. Das Festlegen solcher Konsistenzbedingungen umfasst strukturelle Integritätsbedingungen sowie weitere Validierungsregeln, die systemnah implementiert werden. Dies soll erlauben, dass nicht jeder Programmierer individuell die Integrität überprüfen muss. Im siebten Schritt wird der Datenbankentwurf auf

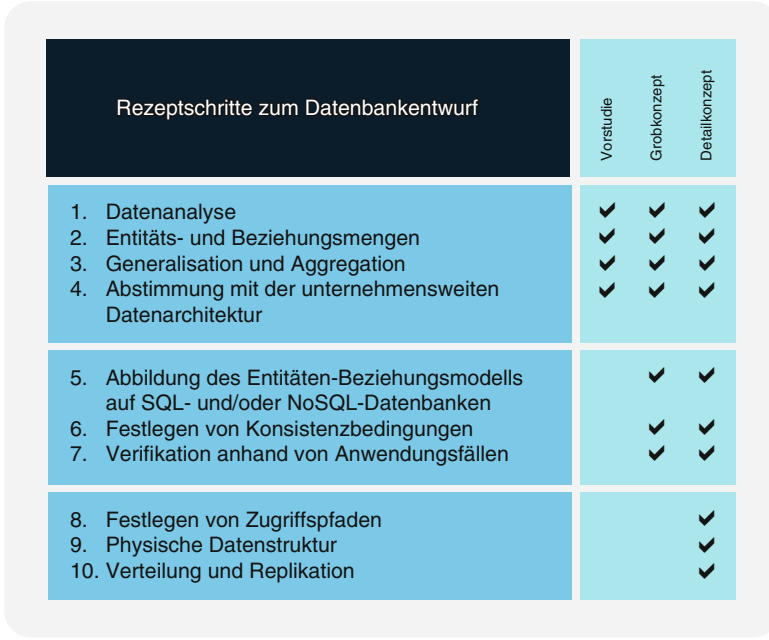

Abb. 2.40 Vom Groben zum Detail in acht Entwurfsschritten

Vollständigkeit hin überprüft, indem man wichtige Anwendungsfälle (Use Cases, vgl. Unified Modeling Language[9]) entwickelt und mit deskriptiven Abfragesprachen prototypartig umsetzt.

Im achten Schritt werden die *Zugriffspfade* für die wichtigsten applikatorischen Funktionen festgehalten. Die häufigsten Merkmale für künftige Datenbankzugriffe müssen analysiert und in einer Zugriffsmatrix aufgezeigt werden. Das Bestimmen eines eigentlichen Mengengerüsts sowie das Definieren der *physischen Datenstruktur* geschieht im neunten Schritt. Schließlich erfolgt die physische *Verteilung der Datenbestände* und die Auswahl möglicher *Replikationsoptionen* im zehnten Schritt. Beim Einsatz von NoSQL-Datenbanken muss hier u. a. abgewogen werden, ob Verfügbarkeit und Ausfalltoleranz gegenüber strenger Konsistenzgewährung bevorzugt werden soll oder nicht (vgl. CAP-Theorem in Abschn. 4.3.1).

Das in Abb. 2.40 gezeigte Rezept beschränkt sich im Wesentlichen auf die Datenaspekte. Neben den Daten spielen natürlich auch Funktionen beim Aufbau von Informationssystemen eine große Rolle. So sind in den letzten Jahren CASE-Werkzeuge

[9] Die Unified Modeling Language oder UML ist eine ISO-standardisierte Modellierungssprache für die Spezifikation, Konstruktion und Dokumentation von Software. Ein Entitäten-Beziehungsmodell kann auf einfache Art in ein Klassendiagramm überführt werden und umgekehrt.

(CASE = Computer Aided Software Engineering) entstanden, die nicht nur den Datenbankentwurf, sondern auch den Funktionsentwurf unterstützen. Wer sich für die Methodik der Anwendungsentwicklung interessiert, findet im nächsten Abschnitt weiterführende Literatur.

2.7 Literatur

Das Entitäten-Beziehungsmodell wurde durch die Arbeiten von Senko und Chen bekannt (vgl. Chen 1976). Seit 1979 gibt es regelmäßig internationale Konferenzen, bei denen Erweiterungen und Verfeinerungen des Entitäten-Beziehungsmodells vorgeschlagen und gemeinsam diskutiert werden.

Viele CASE-Tools verwenden zur Datenmodellierung das Entitäten-Beziehungsmodell, wobei unterschiedliche grafische Symbole für Entitätsmengen, Beziehungsmengen oder Assoziationstypen verwendet werden; vgl. z. B. die Untersuchungen von Balzert (1993), Olle et al. (1988) und Martin (1990). Eine Übersicht über weitere logische Datenmodelle geben Tsichritzis und Lochovsky (1982).

Blaha und Rumbaugh (2004); Booch (2006) sowie Coad und Yourdon (1991) behandeln den objektorientierten Entwurf. Ferstl und Sinz (1991) zählen zu den deutschsprachigen Autoren, die das objektorientierte Vorgehen für die Entwicklung von Informationssystemen vorschlagen. Balzert (2004) kombiniert Methodenansätze von Coad, Booch und Rumbaugh für die objektorientierte Analyse. Einen Vergleich objektorientierter Analysemethoden gibt Stein (1994). Vetter (1998) propagiert ebenfalls den objektorientierten Ansatz für die Datenmodellierung. Eine Einführung in die Unified Modeling Language (UML) primär für die Softwareentwicklung stammt von Hitz et al. (2005).

Smith und Smith (1977) haben die Konzepte der Generalisation und Aggregation für den Datenbankbereich eingeführt. Diese Strukturkonzepte sind vor allem auf dem Gebiet der wissensbasierten Systeme seit langem bekannt, beispielsweise bei der Beschreibung semantischer Netze, vgl. dazu Findler (1979).

Das Studium der Normalformen hat dem Datenbankbereich eine eigentliche Datenbanktheorie beschert (Fagin 1979). Zu den Standardwerken mit eher theoretischem Anstrich zählen die Werke Maier (1983); Ullman (1982, 1988) und Paredaens et al. (1989). Dutka und Hanson (1989) geben kompakt und anschaulich eine zusammenfassende Darstellung der Normalformen. Die Standardwerke von Date (2004), von Elmasri und Navathe (2015), von Kemper und Eickler (2013) oder von Silberschatz et al. (2010) widmen einen wesentlichen Teil der Normalisierung.

Zur Graphentheorie existieren einige Grundlagenwerke: Diestel (2006) vermittelt die wichtigsten Methoden und wagt sich gar an den Minorensatz,[10] der auf die algorithmische

[10] Der Minorensatz von Robertson und Seymour besagt, dass die endlichen Graphen durch die Minorenrelation wohlquasigeordnet sind.

Graphentheorie fundamentale Auswirkungen hat. Einen algorithmischen Zugang zur Graphentheorie vermittelt das Werk von Turau (2009). Eine anwendungsorientierte Einführung in die Graphentheorie stammt von Tittmann (2011). Englischsprachige Werke zur Graphentheorie stammen von Marcus (2008) und van Steen (2010); letzteres gibt interessante Anwendungsoptionen zur Theorie, u. a. für Computernetzwerke, Optimierungsfragen oder soziale Netze. Brüderlin und Meier (2001) vermitteln in ihrem Werk Grundlagen zur Computergrafik und Computergeometrie.

Dijkstra (1959) hat in der Zeitschrift Numerische Mathematik seinen Algorithmus publiziert. Einen Überblick über Voronoi-Diagramme und grundlegende geometrische Datenstrukturen stammt von Aurenhammer (1991). Liebling und Pournin (2012) vergleichen die Voronoi-Diagramme und Delaunay-Triangulationen als siamesische Zwillinge. Die Dissertation von Brown (1979) befasst sich mit geometrischen Transformationen und Algorithmen. Die Dissertation von Shamos (1978) gibt einen Zugang zur algorithmischen Computergeometrie; der in Abschn. 2.41 erwähnte rekursive Algorithmus zum Erhalt von Voronoi-Diagrammen stammt von Shamos und Hoey (1975).

Fragen der unternehmensweiten Datenarchitektur werden in Dippold et al. (2005), Meier et al. (1991); Scheer (1997) und Silverston (2001) diskutiert. Aufgaben und Pflichten der Datenmodellierung und -administration umschreiben Meier und Johner (1991) sowie Ortner et al. (1990).

Datenbanksprachen

3

3.1 Interaktion mit einer Datenbank

Zum erfolgreichen Betreiben einer Datenbank ist eine Datenbanksprache notwendig, mit der die unterschiedlichen Anforderungen der Benutzenden abgedeckt werden können. Relationale Abfrage- und Manipulationssprachen haben den Vorteil, dass man mit ein und derselben Sprache Datenbanken erstellen, Benutzerrechte vergeben oder Tabelleninhalte verändern und auswerten kann. Graphbasierte Sprachen bieten vordefinierte Routinen zum Lösen von Graphproblemen wie beispielsweise für das Aufsuchen von kürzesten Pfaden.

Die *Datenbankadministratorin* oder der *Datenbankadministrator* verwaltet mit einer Datenbanksprache die für das Unternehmen gültigen Datenbeschreibungen, z. B. für Tabellen und Merkmale. Sinnvollerweise werden sie dabei durch ein Data-Dictionary-System (siehe Glossar) unterstützt. Zusammen mit den *Datenarchitekten* sorgen sie dafür, dass die Beschreibungsdaten im Sinne der unternehmensweiten Datenarchitektur einheitlich und konsistent verwaltet und nachgeführt werden, eventuell unter Zuhilfenahme eines geeigneten CASE[1]-Werkzeuges. Neben der Kontrolle der Datenformate legen sie Berechtigungen fest, durch die sich einerseits die Datenbenutzung auf einzelne Tabellen oder sogar einzelne Merkmale, andererseits eine bestimmte Operation wie das Löschen oder Verändern einer Tabelle auf einen Benutzerkreis einschränken lässt.

Die *Datenbankspezialistin* oder der *Datenbankspezialist* definiert, installiert und überwacht die Datenbanken mittels dafür vorgesehener Systemtabellen. Diese bilden den Systemkatalog, der zur Betriebszeit des Datenbanksystems alle notwendigen Datenbankbeschreibungen und statistischen Angaben umfasst. Mit den Systeminformationen können

[1] CASE = Computer Aided Software Engineering.

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_3

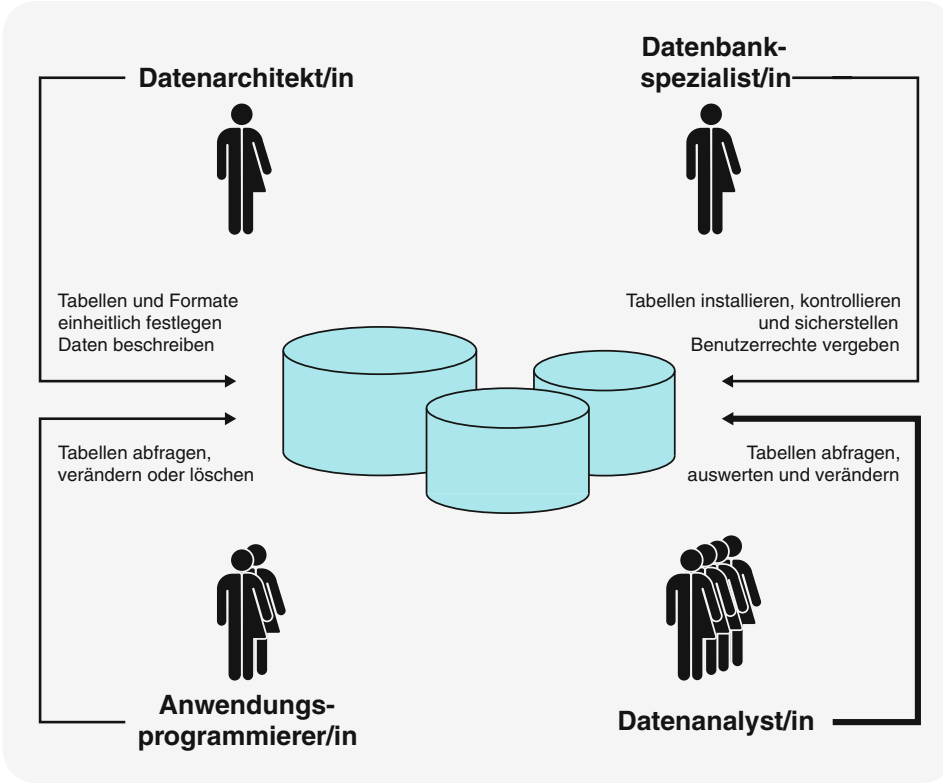

Abb. 3.1 Verwendung einer Datenbanksprache, Beispiel SQL

sich die Datenbankspezialisten durch vorgegebene Abfragen ein aktuelles Bild über sämtliche Datenbanken machen, ohne sich um die eigentlichen Tabellen mit den Benutzerdaten im Detail zu kümmern. Aus Datenschutzgründen sollte ihnen der Zugriff auf Daten im Betrieb nur in Ausnahmesituationen erlaubt sein, beispielsweise zur Fehlerbehebung (Abb. 3.1).

Die *Anwendungsprogrammiererin* oder der *Anwendungsprogrammierer* benutzt die Datenbanksprache, um neben Auswertungen auch Veränderungen auf den Datenbanken vornehmen zu können. Da eine relationale Abfrage- und Manipulationssprache mengenorientiert ist, benötigen die Programmierenden ein *Cursorkonzept* (vgl. Abschn. 3.5.1), um den Datenbankzugriff in ihre Softwareentwicklung zu integrieren. Dies erlaubt ihnen, eine Menge von Tupeln satzweise in einem Programm abzuarbeiten. Via Programmierschnittstellen können sie die Datenbanksprache in ihre Softwareentwicklung einbetten. Für die eigentlichen Tests ihrer Anwendungen stehen ihnen die relationale Sprache ebenfalls zur Verfügung. Sie können damit auf einfache Art ihre Testdatenbanken kontrollieren und diese an die zukünftigen Anwendenden in Form von Prototypen weitergeben.

Schließlich verwenden gelegentliche Benutzende und vor allem *Datenanalyst*en, oft Data Scientists genannt, Datenbanksprachen für ihre *täglichen Auswertungsbedürfnisse*. Datenanalysten sind Spezialistinnen und Spezialisten in der fachlichen Auswertung von Datenbanken mit einer Datenbanksprache. Anwendende verschiedener Fachabteilungen, die bezüglich Informatik beschränkte Kenntnisse besitzen und bei Sachfragen Auswertungen haben möchten, geben den Auftrag an die Datenanalysten, eine Datenbankanalyse mit der entsprechenden fachlichen Fragestellung durchzuführen.

Wie wir gesehen haben, können verschiedene Benutzergruppen ihre Bedürfnisse mit einer relationalen Datenbanksprache abdecken. Dabei verwenden die Datenanalysten dieselbe Sprache wie Anwendungsentwickler, Datenbankadministratoren oder Datenarchitekten. Datenbankanwendungen oder -auswertungen wie auch technische Arbeiten im Bereich Sicherstellen und Reorganisieren von Datenbanken können also in einer einheitlichen Sprache vorgenommen werden. Dadurch wird der Ausbildungsaufwand reduziert. Überdies lassen sich Erfahrungen zwischen den verschiedenen Benutzergruppen besser austauschen.

3.2 Die Relationenalgebra

3.2.1 Übersicht über Operatoren

Die meisten kommerziell genutzten Datenbanksysteme basieren heute noch auf SQL resp. Derivaten davon; also auf Sprachen, die sich an Tabellen und an Operationen auf Tabellen orientieren. Die *Relationenalgebra* (engl. *relational algebra*) bildet den *formalen Rahmen für die relationalen Datenbanksprachen*. Sie definiert einen Satz von algebraischen Operatoren, die immer auf Relationen wirken. Zwar verwenden die meisten der heutigen relationalen Datenbanksprachen diese Operatoren nicht direkt, sie können aber nur dann im Sinne des Relationenmodells als relational vollständige Sprachen bezeichnet werden, wenn das ursprüngliche Potenzial der Relationenalgebra erhalten bleibt.

Im Folgenden wird für zwei beliebige Relationen R und S eine Übersicht über die Operatoren der Relationenalgebra gegeben, aufgeteilt nach mengenorientierten und relationenorientierten Operatoren. Sämtliche Operatoren verarbeiten entweder eine Tabelle oder zwei Tabellen und erzeugen wieder eine Relation. Diese Einheitlichkeit (algebraische Eigenschaft) macht es möglich, mehrere Operatoren miteinander zu kombinieren und auf Relationen wirken zu lassen.

Die mengenorientierten Operatoren entsprechen den bekannten Mengenoperationen (vgl. Abb. 3.2 und nächsten Abschn. 3.2.2). Zu diesen zählen die Vereinigung, symbolisiert durch das Spezialzeichen $\cup$, der Durchschnitt $\cap$, die Subtraktion $\setminus$ und das kartesische Produkt $\times$. Aufgrund eines Verträglichkeitskriteriums können zwei Relationen R und S miteinander vereinigt ($R \cup S$) oder geschnitten ($R \cap S$) oder voneinander subtrahiert ($R \setminus S$) werden. Zusätzlich lassen sich je zwei beliebige Relationen R und S bedingungslos

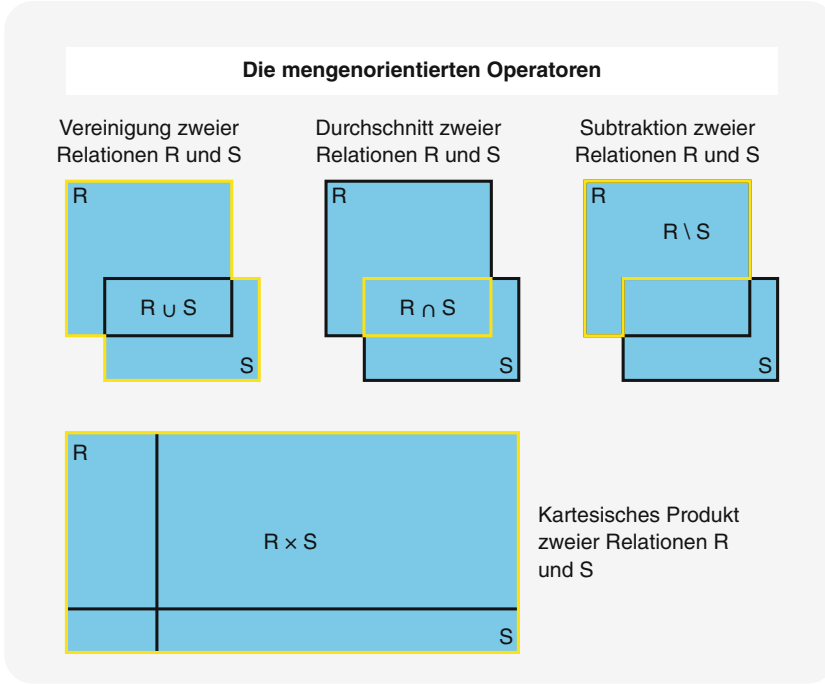

Abb. 3.2 Vereinigung, Durchschnitt, Differenz und kartesisches Produkt von Relationen

miteinander multiplizieren ($R \times S$). Das Resultat solcher Mengenoperationen ist wiederum eine Menge von Tupeln, also eine Relation.

Die *relationenorientierten Operatoren* gemäß Abb. 3.3 wurden von Ted Codd speziell für Relationen festgelegt; sie werden ausführlich in Abschn. 3.2.3 erläutert. Mit Hilfe des Projektionsoperators, abgekürzt durch den Buchstaben π (griechisch Pi), lassen sich Relationen auf Teilmengen reduzieren. So bildet der Ausdruck $\pi_M(R)$ eine Untermenge der Relation R anhand einer Menge M von Merkmalen. Der Selektionsoperator $\sigma_F(R)$, symbolisiert durch σ (griechisch Sigma), trifft eine Auswahl von Tupeln aus der Relation R anhand eines Selektionskriteriums oder einer sogenannten Formel F. Der Verbundoperator (engl. *join*), gegeben durch das Spezialzeichen |X|, kombiniert zwei Relationen zu einer neuen. So lassen sich die beiden Relationen R und S durch die Operation $R|X|_P S$ verbinden, wobei P die entsprechende Verbundbedingung (Verbundprädikat) angibt. Schließlich berechnet die Divisionsoperation $R \div S$ eine Teiltabelle, indem sie die Relation R durch die Relation S dividiert. Der Divisionsoperator wird durch das Spezialzeichen $\div$ dargestellt.

In den nächsten beiden Abschnitten erläutern wir die mengen- und relationenorientierten Operatoren der Relationenalgebra anhand anschaulicher Beispiele.

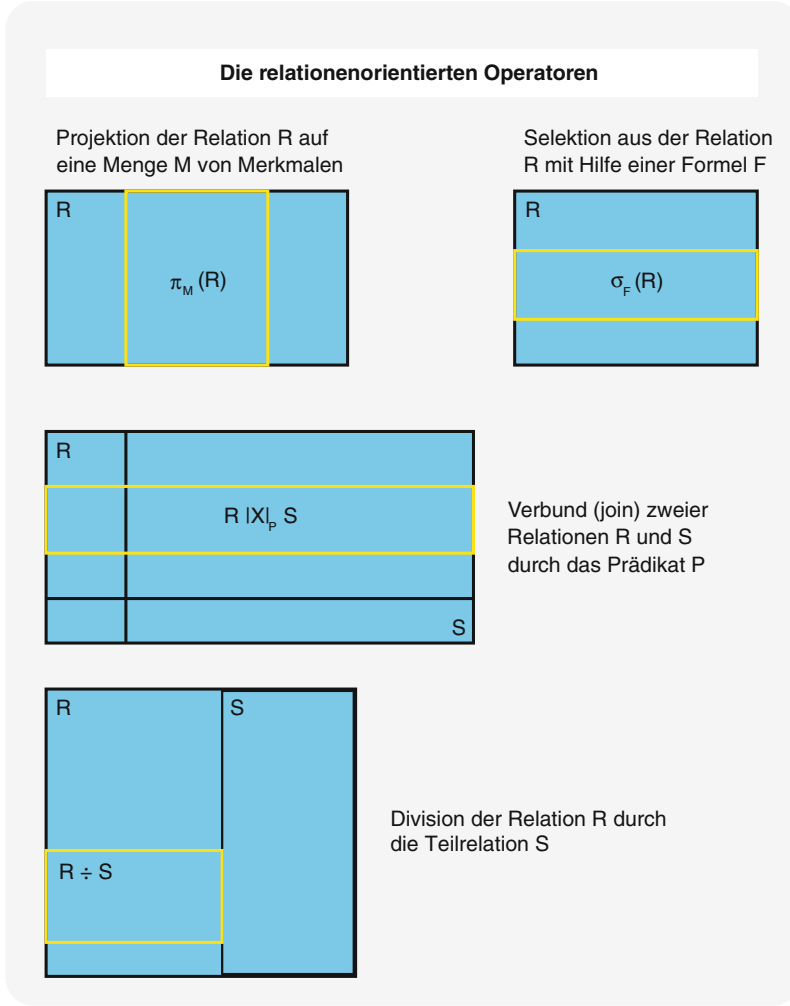

Abb. 3.3 Projektion, Selektion,Verbund und Division von Relationen

3.2.2 Die mengenorientierten Operatoren

Da jede Relation eine Menge von Datensätzen (Tupeln) darstellt, können verschiedene Relationen mengentheoretisch miteinander verknüpft werden. Von zwei Relationen lässt sich jedoch nur dann eine Vereinigung, ein Durchschnitt oder eine Differenz berechnen, wenn diese vereinigungsverträglich sind.

Vereinigungsverträglichkeit
Zwei Relationen sind *vereinigungsverträglich* (engl. *union compatible*), wenn folgende zwei Bedingungen erfüllt sind: Beide Relationen weisen die gleiche Anzahl Merkmale auf und die Datenformate der korrespondierenden Merkmalskategorien sind identisch.

Betrachten wir dazu das in Abb. 3.4 dargestellte Beispiel: Aus einer Mitarbeiterdatei ist für die beiden firmeneigenen Clubs je eine Tabelle definiert worden, die neben der Mitarbeiternummer den Namen und die Adresse enthält. Obwohl die Merkmalsnamen teilweise unterschiedlich lauten, sind die beiden Tabellen SPORTCLUB und FOTOCLUB vereinigungsverträglich. Sie weisen dieselbe Anzahl Merkmale auf, wobei die Merkmalswerte aus ein und derselben Mitarbeiterdatei stammen und somit auch über gleiche Wertebereiche definiert sind.

Allgemein werden zwei vereinigungsverträgliche Relationen R und S mengentheoretisch durch die *Vereinigung* (engl. *union*) R∪S kombiniert, indem *sämtliche Einträge aus R und sämtliche Einträge aus S in die Resultattabelle eingefügt* werden. Gleichzeitig werden identische Datensätze eliminiert; diese sind in der Resultatmenge R∪S aufgrund der Merkmalsausprägungen nicht mehr unterscheidbar.

Die Tabelle CLUBMITGLIEDER (Abb. 3.5) ist eine Vereinigung der Tabellen SPORTCLUB und FOTOCLUB. Jedes Resultattupel kommt entweder in der Tabelle SPORTCLUB oder FOTOCLUB oder in beiden gleichzeitig vor. Das Clubmitglied Huber erscheint nur einmal, da identische Einträge in der Vereinigungsmenge nicht zugelassen sind.

Die übrigen mengenorientierten Operatoren sind auf analoge Art definiert: Zum *Durchschnitt* (engl. *intersection*) R∩S zweier vereinigungsverträglicher Relationen R und S zählen nur diejenigen Einträge, die sowohl in R als auch in S vorhanden sind. In unserem Tabellenauszug ist nur Huber sowohl im SPORT- als auch im FOTOCLUB Aktivmitglied.

Die erhaltene Resultatmenge SPORTCLUB∩FOTOCLUB ist einelementig, da genau eine Person eine Doppelmitgliedschaft aufweist.

Schließlich können vereinigungsverträgliche Relationen voneinander subtrahiert werden. Die *Differenz* (engl. *difference*) R\S erhält man dadurch, dass man in R sämtliche Einträge entfernt, die in S enthalten sind. Auf unser Beispiel angewendet, bedeutet die Subtraktion SPORTCLUB\FOTOCLUB, dass wir in der Resultatrelation nur die beiden Mitglieder Meier und Schweizer finden. Das Mitglied Huber wird eliminiert, da es zusätzlich Mitglied im FOTOCLUB ist. Somit erlaubt der Differenzoperator die Bestimmung derjenigen Sportclubmitglieder, die nicht gleichzeitig dem Fotoclub angehören.

Allgemein besteht zwischen dem Durchschnitts- und dem Differenzoperator zweier vereinigungsverträglicher Relationen die folgende Beziehung:

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

SPORTCLUB

M#	Name	Straße	Wohnort
M1	Meier	Lindenstraße	Liestal
M7	Huber	Mattenweg	Basel
M19	Schweizer	Hauptstraße	Frenkendorf

FOTOCLUB

M#	Name	Straße	Wohnort
M4	Becker	Wasserweg	Liestal
M7	Huber	Mattenweg	Basel

Abb. 3.4 Vereinigungsverträgliche Tabellen SPORT- und FOTOCLUB

CLUBMITGLIEDER = SPORTCLUB ∪ FOTOCLUB

M#	Name	Straße	Wohnort
M1	Meier	Lindenstraße	Liestal
M7	Huber	Mattenweg	Basel
M19	Schweizer	Hauptstraße	Frenkendorf
M4	Becker	Wasserweg	Liestal

Abb. 3.5 Vereinigung der beiden Tabellen SPORTCLUB und FOTOCLUB

Somit kann die Durchschnittsbildung auf die Differenzbildung zurückgeführt werden. Diese Beziehung lässt sich leicht an unserem Beispiel der Sport- und Fotoclubmitglieder veranschaulichen.

Bei den mengenorientierten Operatoren fehlt noch das kartesische Produkt zweier beliebiger Relationen R und S, die keineswegs vereinigungsverträglich sein müssen. Unter dem *kartesischen Produkt* (engl. *cartesian product*) $R \times S$ zweier Relationen R und S versteht man die Menge aller möglichen Kombinationen von Tupeln aus R mit Tupeln aus S.

Als Beispiel betrachten wir in Abb. 3.6 die Tabelle WETTKAMPF, die eine Mitgliederkombination $(SPORTCLUB \backslash FOTOCLUB) \times FOTOCLUB$ darstellt. Diese Tabelle enthält also alle möglichen Kombinationen von Sportclubmitgliedern (die nicht zugleich im Fotoclub sind) mit den Fotoclubmitgliedern. Sie drückt eine typische Wettkampfzusammenstellung der beiden Clubs aus, wobei das Doppelmitglied Huber natürlich nicht gegen sich selbst antreten kann und aufgrund der Subtraktion SPORTCLUB\FOTOCLUB auf der Seite der Fotoclubmitglieder kämpft.

Diese Operation heisst kartesisches Produkt, da die jeweiligen Einträge der Ausgangstabellen miteinander multipliziert werden. Haben allgemein zwei beliebige Relationen R und S m beziehungsweise n Einträge, so hat das kartesische Produkt $R \times S$ insgesamt m mal n Tupeleinträge.

3.2.3 Die relationenorientierten Operatoren

Die relationenorientierten Operatoren ergänzen die mengenorientierten, wobei hier die jeweiligen Relationen – wie schon beim kartesischen Produkt – nicht vereinigungsverträglich sein müssen. Der *Projektionsoperator* (engl. *projection*) $\pi_M(R)$ bildet eine Teilrelation aus der Relation R aufgrund der durch M definierten Merkmalsnamen. Beispielsweise bedeutet für eine Relation R mit den Attributen (A,B,C,D) die Schreibweise

WETTKAMPF = (SPORTCLUB \ FOTOCLUB) × FOTOCLUB

M#	Name	Straße	Wohnort	*M#*	Name	Straße	Wohnort
M1	Meier	Lindenstraße	Liestal	M4	Becker	Wasserweg	Liestal
M1	Meier	Lindenstraße	Liestal	M7	Huber	Mattenweg	Basel
M19	Schweizer	Hauptstraße	Frenkendorf	M4	Becker	Wasserweg	Liestal
M19	Schweizer	Hauptstraße	Frenkendorf	M7	Huber	Wasserweg	Basel

Abb. 3.6 Relation WETTKAMPF als Beispiel eines kartesischen Produktes

$\pi_{A,C}(R)$, dass R auf die Merkmale A und C reduziert wird. Es ist erlaubt, in einer Projektion die Attributnamen in einer beliebigen Reihenfolge aufzulisten. So bedeutet $R' := \pi_{C,A}(R)$ die Projektion der Relation $R = (A,B,C,D)$ auf $R' = (C,A)$.

Das erste Beispiel π_{Ort}(MITARBEITER) in Abb. 3.7 listet alle Orte aus der Mitarbeitertabelle wiederholungsfrei in eine einspaltige Tabelle. Im zweiten Beispiel $\pi_{Unt,Name}$(MITARBEITER) erhalten wir eine Teiltabelle mit sämtlichen Abteilungsnummern und Namen der zugehörigen Mitarbeitenden.

Ein weiterer wichtiger Operator $\sigma_F(R)$ dient der *Selektion* (engl. *selection*) *von Tupeln aus der Relation R anhand der Formel F*. Eine Formel F besitzt eine bestimmte Anzahl von Merkmalsnamen oder konstanten Werten, die durch Vergleichsoperatoren wie $<$, $>$ oder $=$ sowie durch logische Operatoren wie AND, OR und NOT miteinander kombiniert werden können. Unter $\sigma_F(R)$ versteht man somit alle Tupel aus R, die die Selektionsbedingung F erfüllen.

Betrachten wir dazu die Beispiele der Abb. 3.8 zur Selektion von Tupeln aus der Tabelle MITARBEITER: Im ersten Beispiel werden alle Mitarbeitenden bestimmt, die die Bedingung ‚Ort = Liestal' erfüllen und somit in Liestal wohnen. Das zweite Beispiel mit der Bedingung ‚Unt = A6' selektiert nur die in der Abteilung A6 tätigen Mitarbeitenden. Schießlich kombiniert das dritte Beispiel die beiden ersten Selektionsbedingungen durch eine logische Verknüpfung anhand der Formel ‚Ort = Liestal AND Unt = A6'.

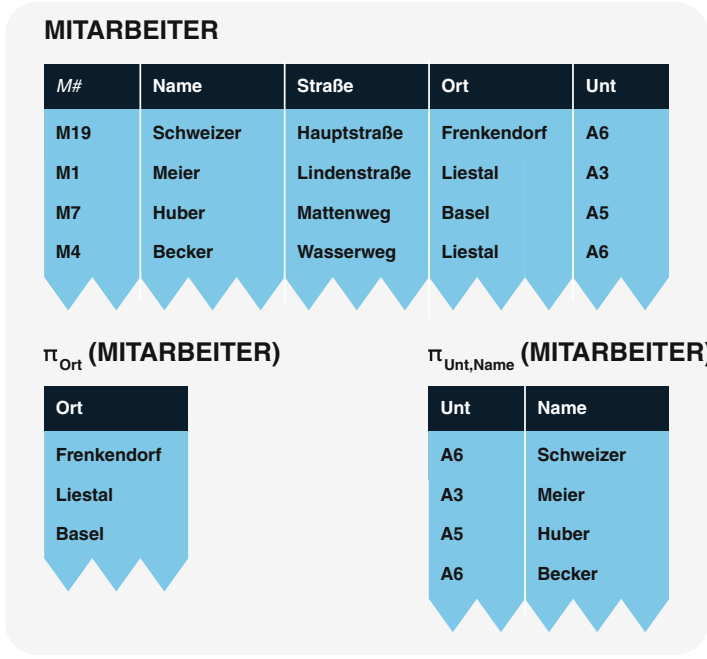

Abb. 3.7 Projektionsoperatoren am Beispiel MITARBEITER

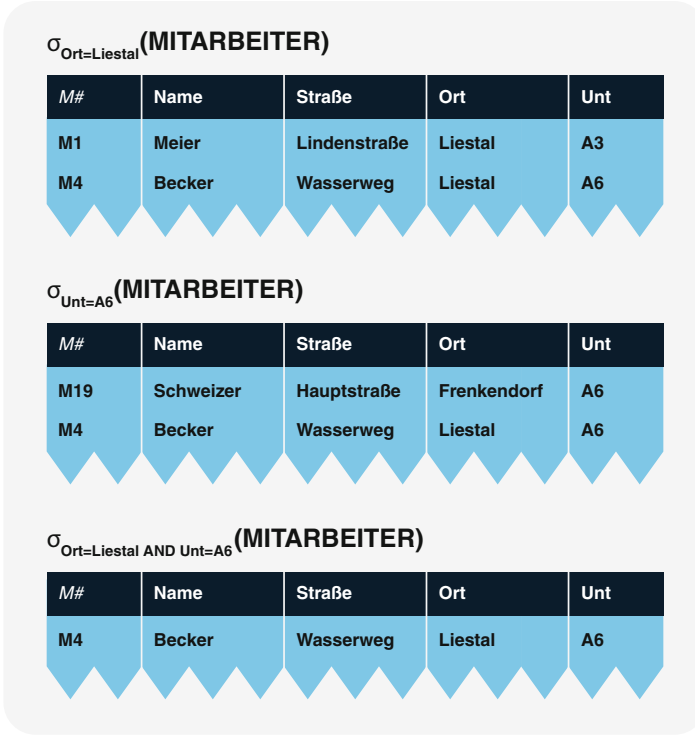

Abb. 3.8 Beispiele von Selektionsoperatoren

Dabei erhalten wir als Resulattabelle eine einelementige Relation, denn nur Mitarbeiter Becker stammt aus Liestal und arbeitet in der Abteilung mit der Nummer A6.

Natürlich lassen sich die bereits besprochenen Operatoren der Relationenalgebra auch miteinander kombinieren. Wenden wir zum Beispiel nach einer Selektion der Mitarbeitenden der Abteilung A6 durch $\sigma_{Unt=A6}$(MITARBEITER) anschliessend eine Projektion auf das Merkmal Ort durch den Operator $\pi_{Ort}(\sigma_{Unt=A6}$(MITARBEITER)) an, so erhalten wir als Resultattabelle die beiden Ortschaften Frenkendorf und Liestal.

Betrachten wir nun den *Verbundoperator* (engl. *join operator*), durch den sich zwei Relationen zu einer einzigen zusammenfügen lassen. Der Verbund $R|X|_P S$ der beiden Relationen R und S über das Prädikat P ist eine *Kombination aller Tupel aus R mit allen Tupeln aus S, die jeweils das Verbundprädikat P erfüllen*. Das Verbundprädikat enthält ein Merkmal aus der Relation R und eines aus S. Diese beiden Merkmale werden durch Vergleichsoperatoren $<$, $>$ oder $=$ in Beziehung gesetzt, damit die Relationen R und S kombiniert werden können. Enthält das Verbundprädikat P den Vergleichsoperator ‚=', so spricht man von einem *Gleichheitsverbund* (engl. *equi-join*).

Der Verbundoperator stößt oft auf Verständnisschwierigkeiten und kann dadurch zu falschen oder ungewollten Resultaten führen. Der Grund liegt meistens darin, dass das Prädikat für die Kombination zweier Tabellen vergessen oder falsch definiert wird.

Betrachten wir in Abb. 3.9 als Beispiel zwei Verbundoperatoren mit und ohne Angabe des Verbundprädikates. Mit der Spezifikation MITARBEITER$|X|_{Unt=A\#}$ABTEILUNG verknüpfen wir die beiden Tabellen MITARBEITER und ABTEILUNG, indem wir die Angaben der Mitarbeitenden mit den Angaben ihrer zugehörigen Abteilungen ergänzen.

Vergessen wir im Beispiel aus Abb. 3.9 die Angabe eines Verbundprädikates P und spezifizieren MITARBEITER $\times$ ABTEILUNG, so erhalten wir das kartesische Produkt der beiden Tabellen MITARBEITER und ABTEILUNG. Dieses Beispiel illustriert eine nicht sehr sinnvolle Kombination der beiden Tabellen, da sämtliche Mitarbeitenden sämtlichen Abteilungen gegenübergestellt werden. Wir finden also in der Resultattabelle auch die Kombination von Mitarbeitenden mit Abteilungen, denen sie organisatorisch gar nicht zugeteilt sind (vgl. dazu die Tabelle WETTKAMPF in Abb. 3.6).

Wie die Beispiele in Abb. 3.9 zeigen, ist der Verbundoperator |X| mit dem Verbundprädikat P nichts anderes als eine Einschränkung des kartesischen Produktes.

Allgemein drückt ein Verbund zweier Tabellen R und S ohne die Spezifikation des Verbundprädikats ein kartesisches Produkt der beiden Tabellen R und S aus, d. h., es gilt mit dem leeren Prädikat P = { }

$$R\left|X\right|_{P=\{\}}S \;=\; R \times S.$$

Verwenden wir bei der Selektion als Selektionsprädikat ein Verbundprädikat, so erhalten wir:

$$R\left|X\right|_{P}S \;=\; \sigma_{P}(R \times S).$$

Diese allgemeine Formel drückt aus, dass jeder Verbund in einem ersten Schritt durch ein kartesisches Produkt und in einem zweiten Schritt durch eine Selektion ausgedrückt werden kann.

Auf das Beispiel aus Abb. 3.9 bezogen können wir unseren gewünschten Ausdruck des Verbundes MITARBEITER$|X|_{Unt=A\#}$ABTEILUNG durch die folgenden zwei Schritte berechnen: Zuerst bilden wir das kartesische Produkt der beiden Tabellen MITARBEITER und ABTEILUNG. In der Tabelle dieses Zwischenresultats werden nun diejenigen Einträge durch die Selektion $\sigma_{Unt=A\#}$(MITARBEITER $\times$ ABTEILUNG) bestimmt, bei denen das Verbundprädikat Unt = A# erfüllt ist. Damit erhalten wir dieselben Tupel wie bei der direkten Berechnung des Verbundes MITARBEITER$|X|_{Unt=A\#}$ABTEILUNG (vgl. dazu die gelb hinterlegten Tupeleinträge in Abb. 3.9).

Eine *Division* (engl. *division*) der Relation R durch die Relation S kann nur durchgeführt werden, falls S als Teilrelation in R enthalten ist. Der Divisionsoperator R $\div$ S berechnet eine Teilrelation R′ aus R mit der Eigenschaft, dass die Kombinationen *aller*

MITARBEITER

M#	Name	Straße	Ort	Unt
M19	Schweizer	Hauptstraße	Frenkendorf	A6
M1	Meier	Lindenstraße	Liestal	A3
M7	Huber	Mattenweg	Basel	A5
M4	Becker	Wasserweg	Liestal	A6

ABTEILUNG

A#	Bezeichnung
A3	Informatik
A5	Personal
A6	Finanz

MITARBEITER $|\times|_{Unt=A\#}$ ABTEILUNG

M#	Name	Straße	Ort	Unt	A#	Bezeichnung
M19	Schweizer	Hauptstraße	Frenkendorf	A6	A6	Finanz
M1	Meier	Lindenstraße	Liestal	A3	A3	Informatik
M7	Huber	Mattenweg	Basel	A5	A5	Personal
M4	Becker	Wasserweg	Liestal	A6	A6	Finanz

MITARBEITER × ABTEILUNG

M#	Name	Straße	Ort	Unt	*A#*	Bezeichnung
M19	Schweizer	Hauptstraße	Frenkendorf	A6	A3	Informatik
M19	Schweizer	Hauptstraße	Frenkendorf	A6	A5	Personal
M19	Schweizer	Hauptstraße	Frenkendorf	A6	A6	Finanz
M1	Meier	Lindenstraße	Liestal	A3	A3	Informatik
M1	Meier	Lindenstraße	Liestal	A3	A5	Personal
M1	Meier	Lindenstrasse	Liestal	A3	A6	Finanz
M7	Huber	Mattenweg	Basel	A5	A3	Informatik
M7	Huber	Mattenweg	Basel	A5	A5	Personal
M7	Huber	Mattenweg	Basel	A5	A6	Finanz
M4	Becker	Wasserweg	Liestal	A6	A3	Informatik
M4	Becker	Wasserweg	Liestal	A6	A5	Personal
M4	Becker	Wasserweg	Liestal	A6	A6	Finanz

Abb. 3.9 Verbund zweier Tabellen mit und ohne Verbundprädikat

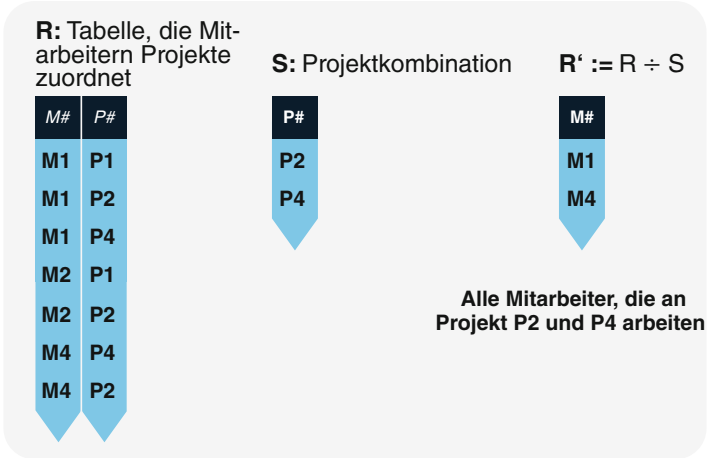

Abb. 3.10 Beispiel eines Divisionsoperators

Tupel r' aus R' mit den Tupeln s aus S in der Relation R liegen. Es muss also das kartesische Produkt $R' \times S$ in der Relation R enthalten sein.

Die in Abb. 3.10 dargestellte Tabelle R zeigt, welche Mitarbeitende an welchen Projekten arbeiten. Wir interessieren uns nun für alle Personen, die an *sämtlichen* Projekten aus S, d. h. den Projekten P2 und P4, arbeiten. Dazu definieren wir die Tabelle S mit den Projektnummern P2 und P4. Es ist offensichtlich, dass S in R enthalten ist, und wir können somit die Division $R' := R \div S$ berechnen. Die Division liefert als Resultat in der Tabelle R' die beiden Mitarbeitenden M1 und M4. Eine kleine Kontrollrechnung bestätigt, dass sowohl M1 als auch M4 gleichzeitig an den Projekten P2 und P4 arbeiten, da die Tupel (M1,P2), (M1,P4) bzw. (M4,P2) und (M4,P4) in der Tabelle R auftreten.

Man kann einen Divisionsoperator durch Projektions- und Differenzoperatoren sowie durch ein kartesisches Produkt ausdrücken. Damit zählt der Divisionsoperator neben dem Durchschnitts- und dem Verbundoperator zu den ersetzbaren Operatoren der Relationenalgebra.

Zusammenfassend bilden Vereinigung, Differenz, kartesisches Produkt, Projektions- und Selektionsoperatoren die kleinste Menge von Operatoren, die die Relationenalgebra voll funktionsfähig macht: Der Durchschnitts-, der Verbund- sowie der Divisionsoperator lassen sich, wenn auch manchmal umständlich durch diese fünf Operatoren der Relationenalgebra jederzeit herleiten.

Die Operatoren der Relationenalgebra sind nicht nur aus theoretischer Sicht interessant, sondern haben auch ihre praktische Bedeutung. So werden sie für die Sprachschnittstelle relationaler Datenbanksysteme verwendet, um Optimierungen vorzunehmen (vgl. Abschn. 5.3.2). Darüber hinaus gelangen sie beim Bau von Datenbankrechnern zur

Anwendung: Die Operatoren der Relationenalgebra oder abgewandelte Formen davon müssen nicht softwaremäßig realisiert werden, sondern lassen sich direkt in Hardwarekomponenten implementieren.

3.3 Relational vollständige Sprachen

Sprachen sind relational vollständig, wenn sie zumindest der Relationenalgebra ebenbürtig sind. Dies bedeutet, dass alle Operationen auf Daten, welche mit der relationalen Algebra ausgeführt werden können, auch von relational vollständigen Sprachen unterstützt werden.

Die in der Praxis gebräuchlichen Sprachen relationaler Datenbanksysteme orientieren sich an der Relationenalgebra. So kann die bereits erwähnte Sprache SQL, die Structured Query Language, als eine Umsetzung der Relationenalgebra angesehen werden (vgl. Abschn. 3.3.1). Unter QBE, Query by Example, versteht man eine Sprache, mit der man anhand von Hilfsgrafiken die eigentlichen Abfragen und Manipulationen durchführen kann (Abschn. 3.3.2). QBE unterstützt aber gleichzeitig das benutzerfreundliche Arbeiten mit Tabellen durch grafische Elemente.

SQL und QBE sind gleich mächtig wie die Relationenalgebra und gelten deshalb als *relational vollständige* Sprachen. Die relationale Vollständigkeit einer Datenbanksprache bedeutet, dass die Operatoren der Relationenalgebra in ihr darstellbar sind.

Kriterium der Vollständigkeit
Eine Datenbankabfragesprache heisst *relational vollständig* im Sinne der Relationenalgebra, wenn sie mindestens die mengenorientierten Operatoren Vereinigung, Differenz und kartesisches Produkt sowie die relationenorientierten Operatoren Projektion und Selektion ermöglicht.

Das Kriterium der Vollständigkeit ist das wichtigste Kriterium beim Überprüfen einer Datenbanksprache auf ihre relationale Tauglichkeit. Nicht jede Sprache, die mit Tabellen arbeitet, ist relational vollständig. Fehlt die Möglichkeit, verschiedene Tabellen über gemeinsame Merkmale zu kombinieren, so ist diese Sprache nicht mit der Relationenalgebra äquivalent. Daher verdient sie auch nicht die Auszeichnung als relational vollständige Datenbanksprache.

Die Relationenalgebra bildet die Grundlage primär für den Abfrageteil einer relationalen Datenbanksprache. Natürlich möchte man neben Auswertungsoperationen auch einzelne Tabellen oder Teile davon manipulieren können. Zu den Manipulationsoperationen zählt beispielsweise das Einfügen, Löschen oder Verändern von Tupelmengen. Aus diesem Grund müssen relational vollständige Sprachen um die folgenden Funktionen erweitert werden, um sie praxistauglich zu machen:

- Es müssen Tabellen und Merkmale definiert werden können.
- Es müssen Einfüge-, Veränderungs- und Löschoperationen vorgesehen werden.
- Die Sprache sollte Aggregatsfunktionen wie Summenbildung, Maximumbestimmung, Berechnung des Minimal- oder des Durchschnittswertes einer einzelnen Spalte der Tabelle enthalten.
- Tabellen sollten formatiert und nach verschiedenen Kriterien ausgedruckt werden können. So interessieren Sortierreihenfolgen oder Gruppenbrüche zum Darstellen von Tabellen.
- Sprachen für relationale Datenbanken müssen zwingend Sprachelemente für die Vergabe von Benutzerberechtigungen und für den Schutz der Datenbanken anbieten (vgl. Abschn. 3.8).
- Es wäre von Vorteil, wenn relationale Datenbanksprachen arithmetische Ausdrücke oder Berechnungen unterstützen würden.
- Relationale Datenbanksprachen sollten dem Mehrbenutzeraspekt Rechnung tragen (Transaktionsprinzip, vgl. Kap. 4) und Befehle für die Datensicherheit bereitstellen.

Durch die Beschreibung der Relationenalgebra haben wir den formalen Rahmen relationaler Datenbanksprachen abgesteckt. In der Praxis ist es nun so, dass diese formale Sprache nicht direkt eingesetzt wird. Vielmehr wurde schon früh versucht, relationale *Datenbanksprachen möglichst benutzerfreundlich* zu gestalten. Da die algebraischen Operatoren den Anwendenden eines Datenbanksystems in reiner Form nicht zumutbar sind, werden die Operatoren durch aussagekräftigere Sprachelemente dargestellt. Die zwei Sprachen SQL und QBE sollen dies in den folgenden Abschnitten beispielhaft demonstrieren.

3.3.1 SQL

Die Sprache SEQUEL (Structured English QUERY Language) wurde Mitte der Siebzigerjahre für «System R» geschaffen; dieses Testsystem war eines der ersten lauffähigen relationalen Datenbanksysteme. Das Prinzip von SEQUEL war eine relational vollständige Abfragesprache, welche nicht auf mathematischen Symbolen, sondern auf *englischen Wörtern* wie ‚select', ‚from', ‚where', ‚count', ‚group by' etc. basiert. Eine Weiterentwicklung dieser Sprache ist unter dem Namen SQL (Structured Query Language) erst durch die ANSI und später durch die ISO normiert worden. SQL ist seit Jahren die führende Sprache für Datenbankabfragen und -interaktionen.

Die Grundstruktur der Sprache SQL sieht gemäss Abschn. 1.2.2 wie folgt aus:

```
SELECT  gewünschte Merkmale     (Output)
FROM    Tabellen des Suchraums  (Input)
WHERE   Selektionsprädikat      (Processing)
```

Die SELECT-Klausel entspricht dem Projektionsoperator der Relationenalgebra, indem sie eine Liste von Merkmalen angibt. Aus der Abfrage $\pi_{Unt,Name}$(MITARBEITER) in Form eines Projektionsoperators der Relationenalgebra, wie in Abb. 3.7 dargestellt, macht SQL ganz einfach:

```
SELECT  Unt, Name
FROM    MITARBEITER
```

In der FROM-Klausel werden alle benötigten Tabellen aufgeführt. Beispielsweise wird das kartesische Produkt zwischen MITARBEITER und ABTEILUNG wie folgt in SQL dargestellt:

```
SELECT  M#,Name,Strasse,Ort,Unt,A#,Bezeichnung
FROM    MITARBEITER, ABTEILUNG
```

Dieser Befehl erzeugt die Kreuzproduktttabelle aus Abb. 3.9, analog wie die gleichwertigen Operatoren MITARBEITER|X|$_{P=\{\}}$ABTEILUNG oder MITARBEITER × ABTEILUNG.

Formuliert man in der WHERE-Klausel das Verbundprädikat ‚Unt = A#', so erhält man den Gleichheitsverbund zwischen den beiden Tabellen MITARBEITER und ABTEILUNG in SQL-Notation:

```
SELECT  M#,Name,Strasse,Ort,Unt,A#,Bezeichnung
FROM    MITARBEITER, ABTEILUNG
WHERE   Unt=A#
```

Qualifizierte Selektionen lassen sich beschreiben, indem in der WHERE-Klausel verschiedene Aussagen durch die logischen Operatoren AND und OR verknüpft werden. Die früher vorgenommene Selektion der Mitarbeitenden, $\sigma_{Ort=Liestal\ AND\ Unt=A6}$(MITARBEITER), dargestellt in Abb. 3.8, lautet in SQL:

```
SELECT  *
FROM    MITARBEITER
WHERE   Ort='Liestal' AND Unt='A6'
```

Ein Stern (*) in der SELECT-Klausel bedeutet, dass alle Merkmale der entsprechenden Tabellen selektiert werden; man bekommt also eine Resultattabelle mit den Merkmalen

M#, Name, Strasse, Ort und Unt (Unterstellung). Die WHERE-Klausel enthält das gewünschte Selektionsprädikat. Die Ausführung der obigen Abfrage durch das Datenbanksystem führt somit zu Mitarbeiter Becker aus Liestal, der in der Abteilung A6 tätig ist.

Neben den üblichen Operatoren der Relationenalgebra existieren bei SQL sogenannte *eingebaute Funktionen* (engl. *built-in functions*), die in der SELECT-Klausel verwendet werden. Zu diesen Funktionen gehören auch die sogenannten *Aggregatfunktionen*, welche ausgehend von einer Menge einen skalaren Wert berechnen. Dazu gehören COUNT für eine Zählung, SUM für eine Summenbildung, AVG für die Berechnung des Mittels (engl. *average*), MAX zur Bestimmung des Maximalwertes und MIN für die Feststellung des Minimalwertes.

Beispielsweise können alle Mitarbeitenden gezählt werden, die in der Abteilung A6 arbeiten. In SQL lautet diese Aufforderung wie folgt:

```
SELECT  COUNT (M#)
FROM    MITARBEITER
WHERE   Unt='A6'
```

Als Resultat erhält man eine einelementige Tabelle mit einem einzigen Wert 2, der gemäss Tabellenauszug für die beiden Personen Schweizer und Becker steht.

Zur Definition einer Tabelle in SQL steht ein CREATE TABLE-Befehl zur Verfügung. Die Tabelle MITARBEITER wird wie folgt spezifiziert:

```
CREATE TABLE  MITARBEITER
(  M#  CHAR(6) NOT NULL,
   Name  VARCHAR(20),
...)
```

Mit dem entgegengesetzten Befehl DROP TABLE können Tabellendefinitionen gelöscht werden. Dabei ist zu beachten, dass dieser Befehl auch sämtliche Tabelleninhalte und dazugehörende Benutzungsrechte eliminiert (vgl. Abschn. 3.8).

Ist die Tabelle MITARBEITER definiert, so können durch die folgende Anweisung neue Tupel eingefügt werden:

```
INSERT INTO MITARBEITER
VALUES ('M20', 'Müller', 'Riesweg', 'Olten', 'A6')
```

Eine Manipulation der Tabelle MITARBEITER ist durch eine UPDATE-Anweisung möglich:

```
UPDATE  MITARBEITER
SET     Ort = 'Basilea'
WHERE   Ort = 'Basel'
```

Diese Beispiel-Anweisung ersetzt in allen Tupeln der Tabelle MITARBEITER sämtliche Wohnorte mit dem Wert Basel durch den neuen Ortsnamen Basilea. Die Veränderungsoperation UPDATE ist ebenfalls mengenorientiert und verändert gegebenenfalls eine mehrelementige Menge von Tupeln.

Schließlich können ganze Tabellen oder Teile davon durch eine DELETE-Anweisung gelöscht werden:

```
DELETE FROM  MITARBEITER
WHERE Ort = 'Basilea'
```

Die Löschanweisung betrifft normalerweise eine Menge von Tupeln, falls das Selektionsprädikat auf mehrere Tabelleneinträge zutrifft. Im Falle referenzieller Integrität (vgl. Abschn. 3.7) kann sich eine solche Löschoperation auch auf abhängige Tabellen auswirken.

Ein Tutorium für die Sprache SQL ist auf der Webseite dieses Buches, www.sql-nosql.org, aufgeführt. Die hier skizzierte Einführung in SQL deckt nur einen kleinen Teil des Standards ab. Die heutige SQL-Sprache bietet viele Erweiterungen, unter anderem im Bereich Programmierung, Sicherheit, Objektorientierung und Auswertung.

3.3.2 QBE

Die Sprache Query-by-Example, abgekürzt QBE, ist eine Datenbanksprache, bei der die Benutzer ihre Auswertungsvorstellungen direkt in der Tabelle interaktiv durch Beispiele entwerfen und ausführen können.

Beispielsweise erhalten die Benutzer eine grafische Darstellung der Tabelle MITARBEITER mit den dazugehörenden Merkmalen:

```
MITARBEITER | M# | Name | Strasse | Ort | Unt
-------------------------------------------
            |    |      |         |     |
```

Dieses Skelett kann nun für Selektionen verwendet werden, indem die Benutzer Anzeigebefehle (abgekürzt durch «P.»), Variablen oder Konstanten einfügen. Möchten

sie beispielsweise die Namen der Mitarbeitenden und die Abteilungsnummern auflisten, so könnten sie das Skelett wie folgt ergänzen:

```
MITARBEITER | M# | Name | Strasse | Ort | Unt
------------------------------------------
            |    | P.   |         |     | P.
```

Der Befehl «P.» (für *print*) entspricht der Forderung nach Darstellung sämtlicher Datenwerte der entsprechenden Tabellenspalte. Damit können Projektionen auf einzelne Merkmale der Tabelle veranlasst werden.

Möchten die Anwendenden alle Merkmale aus der Tabelle selektiert haben, so können sie den Anzeigebefehl direkt unter den Tabellennamen schreiben:

```
MITARBEITER | M# | Name | Strasse | Ort | Unt
-----------------------------------------------
P.          |    |      |         |     |
```

Durch Selektionsbedingungen erweiterte Abfragen auf der Tabelle MITARBEITER können ebenfalls formuliert werden: Beispielsweise interessieren sich die Benutzer für alle Namen von Mitarbeitenden, die in Liestal wohnen und in der Abteilung A6 arbeiten. Sie fügen zu diesem Zweck in der Spalte Ort die Konstante Liestal und in der Spalte Unt die Abteilungsnummer A6 ein:

```
MITARBEITER | M# | Name | Strasse | Ort       | Unt
 ------------------------------------------------
            |    | P.   |         |'Liestal'|'A6'
```

Werden Selektionsbedingungen auf der gleichen Zeile eingegeben, so entsprechen diese immer einer AND-Verknüpfung. Für eine OR- Verknüpfung werden zwei Zeilen für die entsprechenden Bedingungen gebraucht wie in folgendem Beispiel:

```
MITARBEITER | M# | Name | Strasse | Ort       | Unt
------------------------------------------------
            |    | P.   |         |'Liestal'|
            |    | P.   |         |         |'A6'
```

Die Abfrage entspricht einer Selektion sämtlicher Namen von Mitarbeitenden, die entweder in Liestal wohnen oder in der Abteilung A6 arbeiten. Die Resultattabelle kann also mit dem folgenden äquivalenten Ausdruck der Relationenalgebra bestimmt werden:

$$\pi_{\text{Name}}(\sigma_{\text{Ort = Liestal OR Unt = A6}}(\text{MITARBEITER}))$$

Bei Abfragen mit QBE können neben Konstanten auch Variablen verwendet werden. Diese beginnen immer mit einem Underscore-Zeichen «_», gefolgt von einer beliebigen Zeichenkette. Variablen werden beispielsweise für Verbundoperatoren benötigt. Interessieren sich die Anwendenden danach für die Namen und Adressen der Mitarbeitenden, die in der Abteilung Informatik arbeiten, so lautet die Abfrage in QBE:

```
MITARBEITER | M# | Name | Strasse | Ort     | Unt
-----------------------------------------------
            |    | P.   | P.      | P.      |_A

ABTEILUNG   | A# | Bezeichnung
-----------------------------------------------
            | _A | 'Informatik'
```

Der Verbund der beiden Tabellen MITARBEITER und ABTEILUNG wird durch die Variable _A gebildet, die das bekannte Verbundprädikat Unt = A# darstellt.

Um einen neuen Mitarbeiter (M2, Kunz, Riesweg, Olten, A6) in die Tabelle MITARBEITER einzufügen, schreiben die Benutzer den Einfügebefehl «I.» (für insert) in die Spalte des Tabellennamens und füllen die Tabellenzeile mit den Eingabedaten aus:

```
MITARBEITER | M# | Name | Strasse | Ort     | Unt
-----------------------------------------------
I.          |'M2'|'Kunz'|'Riesweg'|'Olten'  |'A6'
```

Die Anwendenden können eine Menge von Tupeln verändern oder löschen, indem sie den Befehl «U.» (für update) oder «D.» (für delete) in die entsprechenden Spalten bzw. Tabellen schreiben.

Die in Liestal wohnenden Personen aus der Tabelle MITARBEITER können die Benutzer wie folgt entfernen:

```
MITARBEITER | M# | Name | Strasse | Ort     | Unt
-----------------------------------------------
D.          |    |      |         |'Liestal'|
```

Wie der Name «Query By Example» andeutet, können ganz im Gegensatz zu SQL mit QBE keine Tabellen definiert oder Berechtigungen für Benutzer vergeben und verwaltet werden. QBE beschränkt sich lediglich auf den Abfrage- und Manipulationsteil, ist aber diesbezüglich wie SQL relational vollständig.

Beim Einsatz von QBE in der Praxis hat es sich gezeigt, dass für kompliziertere Abfragen QBE-Formulierungen weniger gut lesbar sind als die entsprechenden Anweisungen in SQL. Deshalb ist es nicht verwunderlich, dass vor allem Analysespezialisten für schwierigere Abfragen die Sprache SQL bevorzugen.

Obwohl die hier vorgestellte Syntax von QBE eher ein historisches Beispiel darstellt, ist die Grundidee der Abfrage von Datenbanktabellen durch Angabe von Parametern direkt in der grafischen Darstellung der Tabelle auch heute noch relevant. Desktop-Datenbanken für Büroautomation (sowohl kommerziell als auch Open Source) und auch einige Datenbank-Clients beinhalten grafisch orientierte Benutzerschnittstellen, welche den Endanwendenden eine QBE-Oberfläche bieten, in der Daten mit dem in diesem Kapitel dargestellten Prinzip abgefragt werden können. Siehe dazu auch die Fallstudie Travelblitz mit OpenOffice Base auf der Website dieses Buches, www.sql-nosql.org.

3.4 Graphbasierte Sprachen

Die Entwicklung graphbasierter Datenbanksprachen begann Ende der Achtzigerjahre des letzten Jahrhunderts. Da mit der Entwicklung des Webs und den sozialen Medien immer mehr graphstrukturierte Daten produziert werden, gibt es ein Interesse an leistungsfähigen *Graphabfragesprachen* (engl. *graph query languages*).

Eine Graphdatenbank (siehe Abschn. 7.6) speichert Daten in Graphstrukturen und bietet Datenmanipulationen auf der Ebene von Graph-Transformationen an. Wie wir in Abschn. 1.4.1 gesehen haben, bestehen Graphdatenbanken aus Eigenschaftsgraphen mit Knoten und Kanten, welche jeweils eine Menge von Schlüssel-Wert-Paaren als Eigenschaften speichern. Graphbasierte Datenbanksprachen knüpfen dort an und ermöglichen die computersprachliche Interaktion und Programmierung der Verarbeitung von Graphstrukturen in Datenbanken.

Gleich wie relationale Sprachen sind graphbasierte Sprachen mengenorientiert. Sie arbeiten auf der Ebene von Graphen, welche als Mengen von Knoten sowie Kanten resp. Pfaden angesehen werden können. Wie in relationalen Sprachen lassen graphbasierte Sprachen die Filterung von Daten anhand von Prädikaten zu. Diese Filterung wird *konjunktive Abfrage* (engl. *conjunctive query*) genannt. Filter auf Graphen geben eine Untermenge von Knoten oder Kanten des Graphen zurück, was einen Teilgraph darstellt. Dieses Prinzip heisst auf Englisch *Subgraph Matching* (Untergraph- resp. Teilgraph-Abgleich). Ebenfalls wie in relationalen Sprachen auch bieten graphbasierte Sprachen die Möglichkeit, Mengen von Knoten im Graphen zu skalaren Werten zu aggregieren, beispielsweise als Anzahl, Summe oder Minimum.

Im Unterschied zu relationalen Sprachen bieten graphbasierte Sprachen zusätzlich Mechanismen zur Analyse von Pfaden in Graphen. Interessant ist die Suche nach Mustern, welche sich direkt auf Pfade im Graph beziehen; dies gelingt mit dedizierten Sprachelementen. Eine sogenannte reguläre *Pfadabfrage* (engl. *regular path query*), bietet die Möglichkeit, Muster von Pfaden im Graph mit regulären Ausdrücken zu beschreiben, um entsprechende Datensätze in der Datenbank zu finden (vgl. Tutorium für Cypher auf der Website www.sql-nosql.org).

Als Beispiel wählen wir das Entitäten-Beziehungsteilmodell aus Abb. 3.11 aus. Hier wird eine rekursive Beziehung dargestellt, wobei Teile (z. B. eines Produkts) optional

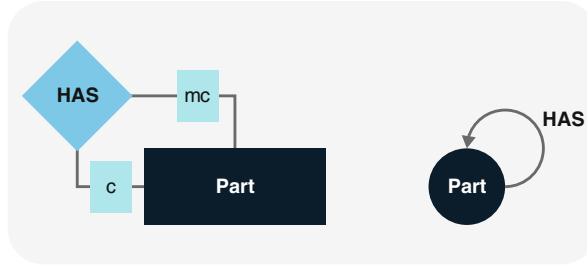

Abb. 3.11 Rekursive Beziehung als Entitäten-Beziehungsmodell resp. als Graph mit Knoten- und Kantentyp

mehrere Unterteile haben können, und selbst wiederum optional in einem Oberteil vorkommen. Wenn wir mit einer Datenbankabfrage alle Unterteile ausgeben möchten, welche ein Oberteil *direkt oder auch indirekt* enthält, reicht ein einfacher Verbund (Join) nicht aus. Wir müssen rekursiv alle Unterteile aller Unterteile etc. durchgehen, um die Liste zu vervollständigen.

In SQL war es lange Zeit nicht möglich, solche Abfragen zu definieren. Erst im SQL:1999-Standard kamen rekursive Abfragen über sogenannte *common table expressions* hinzu, welche allerdings kompliziert zu formulieren sind. Die Definition der Abfrage aller indirekten Unterteile ist mit einem (rekursiven) SQL-Befehl ziemlich umständlich:

```
with recursive
rpath (partID, hasPartId, length) -- CTE-Definition
as (
    select partID, hasPartId, 1 -- Initialisierung
    from part
    union all
    select r.partID, p.hasPartId, r.length+1
    from part p
    join rpath r -- rekursiver Join der CTE
    on ( r.hasPartId = p.partID )
)
select
distinct path.partID, path.hasPartId, path.length
from path -- Selektion via rekursiv definierter CTE
```

Diese Abfrage gibt pro Teil eine Liste aller Unterteile zurück, zusammen mit dem Grad der Verschachtelung, d. h. der Länge des Pfades im Baum vom Oberteil zum (möglicherweise indirekten) Unterteil.

Eine reguläre Pfadabfrage in einer graphbasierten Sprache ermöglicht die vereinfachte Filterung von Pfadmustern mit regulären Ausdrücken. Zum Beispiel definiert der reguläre Ausdruck HAS* mit der Verwendung des Kleene-Sterns (*) die Menge aller möglichen Konkatenationen von Verbindungen mit Kantentyp HAS (die sogenannte Kleene'sche Hülle). Somit wird die Definition einer Abfrage aller indirekt verbundenen Knoten in einer graphbasierten Sprache wesentlich vereinfacht. Das folgende Beispiel in der

graphbasierten Sprache Cypher kann die gleiche Abfrage aller direkten und indirekten Unterteile in einem Zweizeiler deklarieren:

```
MATCH path = (p:Part) <-[:HAS*]- (has:Part)
RETURN p.partID, has.partID, LENGTH(path)
```

Zusammengefasst ist der Vorteil graphbasierter Sprachen, dass die Sprachkonstrukte direkt auf Graphen ausgerichtet sind, und somit die sprachliche Definition der Verarbeitung von graphstrukturierten Daten viel direkter vor sich geht. Im folgenden Abschnitt werden wir nun als konkretes Beispiel die graphbasierte Sprache Cypher vertiefen.

3.4.1 Cypher

Die Graphdatenbank Neo4J[2] (siehe auch Tutorium für Cypher resp. Fallbeispiel Travelblitz mit Neo4J auf der Website www.sql-nosql.org) unterstützt mit der Sprache Cypher eine Sprachschnittstelle für die Verschriftlichung (engl. *scripting*) von Datenbankinteraktionen. Cypher basiert auf dem Mechanismus des *Musterabgleichs* (engl. *pattern matching*).

Cypher kennt wie SQL Sprachbefehle für Datenabfragen und Datenmanipulation (*data manipulation language*, DML). Im Unterschied zu SQL geschieht die Schemadefinition in Cypher aber implizit. Das heisst, dass Knoten- und Kantentypen definiert werden, indem Instanzen dieser Typen als konkrete Knoten und Kanten in die Datenbank eingefügt werden.

Die Datendefinitionssprache *(data definition language*, DDL) von Cypher kann nur Indexe, Unique-Constraints (siehe Abschn. 3.7) und Statistiken beschreiben. Was Cypher nicht kennt, sind direkte Sprachelemente für Sicherheitsmechanismen, welche in relationalen Sprachen mit Befehlen wie GRANT und REVOKE zum Einsatz kommen (vgl. Abschn. 3.8).

Im Folgenden werden wir die Sprache Cypher vertiefen. Die Beispiele beziehen sich auf das Northwind Data Set.[3]

Wie in Abschn. 1.4.2 beschrieben, ist das Grundgerüst von Cypher dreiteilig:

- MATCH zur Beschreibung von Suchmustern,
- WHERE für Bedingungen zur Filterung der Ergebnisse sowie
- RETURN zur Rückgabe von Eigenschaften, Knoten, Beziehungen oder Pfaden.

[2] http://neo4j.com.

[3] http://neo4j.com/developer/guide-importing-data-and-etl/.

Die RETURN-Klausel kann entweder Knoten oder Eigenschaftstabellen zurückeben. Folgendes Beispiel gibt den Knoten zurück, dessen Produktname „Chocolade“ ist:

```
MATCH (p:Product)
WHERE p.productName = 'Chocolade'
RETURN p
```

Diese Rückgabe von ganzen Knoten ist vergleichbar mit ‚SELECT *‘ in SQL. Es können in Cypher auch Eigenschaften als Attributwerte von Knoten und Kanten in Tabellenform zurückgegeben werden:

```
MATCH (p:Product)
WHERE p.unitPrice > 55
RETURN p.productName, p.unitPrice
ORDER BY p.unitPrice
```

Diese Abfrage beinhaltet eine Selektion, eine Projektion und eine Sortierung. In der MATCH-Klausel findet ein Musterabgleich statt, der den Graphen auf den Knoten des Typs ‚Product‘ filtert. Die WHERE-Klausel selektiert alle Produkte mit einem Preis größer als 55. Die RETURN-Klausel projiziert diese Knoten auf die Eigenschaften Produktname und Preis pro Einheit. Dabei sortiert die ORDER-BY-Klausel die Produkte nach dem Preis.

Das kartesische Produkt von zwei Knotentypen kann in Cypher mit folgender Syntax hergestellt werden:

```
MATCH (p:Product), (c:Category)
RETURN p.productName, c.categoryName
```

Diese Anweisung listet alle möglichen Kombinationen von Produktnamen und Kategorienamen auf. Der Verbund (Join) von Knoten, d. h. eine Selektion über dem kartesischen Produkt, geschieht aber graphbasiert über einen Abgleich von Pfadmustern nach Kantentypen:

```
MATCH (p:Product) -[:PART_OF]-> (c:Category)
RETURN p.productName, c.categoryName
```

Diese Abfrage listet zu jedem Produkt die Kategorie, zu der es gehört, indem nur die Produkt- und Kategorie-Knoten berücksichtigt werden, welche über den Kantentyp

PART_OF verbunden sind. Dies entspricht dem inneren Verbund (Inner Join) des Knotentyps ‚Product' mit dem Knotentyp ‚Category' über den Kantentyp PART_OF.

Es gibt Knotentypen, bei denen nur eine Teilmenge von Knoten eine Kante von einem bestimmten Kantentyp aufweist. Zum Beispiel hat nicht jeder Mitarbeiter weitere Mitarbeitende, die ihm unterstellt sind. Somit weist nur eine Teilmenge der Knoten vom Typ Employee eine eingehende Kante vom Typ REPORTS_TO auf.

Nehmen wir an, wir möchten eine Liste mit sämtlichen Mitarbeitenden generieren, welche zudem die Anzahl Unterstellte für die jeweilige Mitarbeiterin oder für den jeweiligen Mitarbeiter aufweist, auch wenn diese Anzahl gleich Null ist. Würde man nun nach dem Muster MATCH (e:Employee)<-[:REPORTS_TO]-(sub) abgleichen, würden in der Liste nur die Mitarbeitenden auftauchen, welche effektiv Unterstellte haben, d. h. bei denen die Anzahl Unterstellten grösser als Null ist:

```
MATCH (e:Employee) <-[:REPORTS_TO]-(sub)
RETURN e.employeeID, count(sub.employeeID)
```

Mit der Klausel OPTIONAL MATCH werden auch diejenigen Mitarbeitenden aufgelistet, welche keine Unterstellte haben:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO]-(sub)
RETURN e.employeeID, count(sub.employeeID)
```

Wie in SQL gibt es in Cypher Operatoren, eingebettete Funktionen und Aggregate. Das folgende Beispiel gibt für jeden Mitarbeitenden den Vornamen und den ersten Buchstaben des Nachnamens aus, zusammen mit der Anzahl der Unterstellten:

```
MATCH (e:Employee)
OPTIONAL MATCH (e)<-[:REPORTS_TO]-(sub)
RETURN
  e.firstName + " "
    + left(e.lastName, 1) + "." as name,
  collect(sub.employeeID)
```

Der Operator +, angewendet auf Datenwerte vom Typ ‚Text', reiht diese aneinander. Die eingebettete Funktion LEFT gibt die ersten n Zeichen eines Texts zurück. Die Aggregatsfunktion ‚Count' zählt die Anzahl Knoten, welche pro Datenwert-Kombination im RETURN-Statement in der Menge der MATCH-Anweisung vorhanden sind.

Im Unterschied zu SQL ist bei Aggregaten keine Angabe einer GROUP-BY-Klausel notwendig. Weitere Aggregate sind wie in SQL Summe (sum), Minimum (min) und Maximum (max). Ein interessantes nicht-atomares Aggregat ist ‚collect', welches ein Array aus den vorhandenen Datenwerten generiert. Somit listet der Ausdruck im vorhergehenden Beispiel alle Mitarbeitenden mit Vornamen und abgekürztem Nachnamen auf, zusammen mit einer Liste der Mitarbeiternummern ihrer Unterstellten (welche auch leer sein kann).

Im Unterschied zu SQL geschieht die *Schemadefinition in Cypher* implizit. Das heisst, dass die abstrakten Datenklassen (Metadaten) wie Knoten- und Kantentypen und Attribute dadurch erstellt werden, dass sie beim Einfügen von konkreten Datenwerten verwendet werden. Folgendes Beispiel fügt neue Daten in die Datenbank ein:

```
CREATE
   (p:Product {
      productName:'SQL- & NoSQL-Datenbanken',
      year:2016})
-[:PUBLISHER]->
   (o:Organization {
      name:'Springer'})
```

Diese Anweisung verdient besondere Betrachtung, weil dadurch vieles implizit geschieht. Es werden zwei neue Knoten kreiert und verbunden, einer für das Produkt „SQL- & NoSQL-Datenbanken", und einer für den Springer-Verlag. Dabei wird implizit ein neuer Knotentyp „Organization" generiert, den es vorher noch nicht gab. Für diese Knoten werden Datenwerte als Attribut-Wert-Paare mitgegeben, welche in die Knoten eingefügt werden. Sowohl das Attribut „year" als auch das Attribut „name" waren bisher nicht vorhanden und werden deshalb implizit im Schema hinzugefügt; in SQL wären dazu ein CREATE-TABLE- und ALTER-TABLE-Befehl nötig. Gleichzeitig wird eine Kante zwischen dem Buchknoten und dem Verlagsknoten mit Kantentyp „PUBLISHER" erstellt, wodurch nicht nur die Kante, sondern auch dieser neue Kantentyp ebenfalls implizit zum Datenbankschema hinzugefügt wird.

Mit dem Befehl SET können Datenwerte verändert werden, die ein bestimmtes Muster erfüllen. Das folgende Beispiel zeigt einen Ausdruck, der den Preis des Produkts „Chocolade" neu setzt:

```
MATCH (p:Product)
WHERE p.productName = 'Chocolade'
SET p.unitPrice = 13.75
```

Mit dem Befehl DELETE können entsprechende Knoten und Kanten gelöscht werden. Da die Graphdatenbank die referenzielle Integrität sicherstellt (siehe Abschn. 3.7), können Knoten erst gelöscht werden, wenn keine Kanten mehr mit ihnen verbunden sind. Daher müssen vor dem Entfernen eines Knotens erst sämtliche eingehenden und ausgehenden Verbindungen gelöscht werden.

Folgendes Beispiel zeigt einen Ausdruck, der für das Produkt ‚Tunnbröd' erst alle verbundenen Kanten als Muster erkennt, diese Kanten entfernt und anschliessend den entsprechenden Knoten löscht:

```
MATCH
  ()-[r1]->(p:Product),
  (p)-[r2]->()
WHERE p.productName = 'Tunnbröd'
DELETE r1, r2, p
```

Zusätzlich zur gewohnten Datenmanipulation unterstützt Cypher im Unterschied zu SQL auch Operationen auf Pfaden im Graphen. Im folgenden Beispiel wird für alle Produktpaare, die in der gleichen Bestellung vorkommen, eine Kante vom Typ „BASKET" (Einkaufskorb) generiert. Diese sagt aus, dass zwei Produkte einmal in der gleichen Bestellung vorgekommen sind. Anschließend kann für zwei beliebige Produkte mit der Funktion *shortestPath* der kürzeste Pfad von gemeinsamen Einkaufskörben eruiert werden:

```
MATCH
    (p1:Product)<--(o:Order)-->(p2:Product)
CREATE
   p1-[:BASKET{order:o.orderID}]->p2,
   p2-[:BASKET{order:o.orderID}]->p1;

MATCH path =
   shortestPath(
      (p1:Product)-[b:BASKET*]->(p2:Product))
RETURN
   p1.productName, p2.productName, LENGTH(path),
   EXTRACT( r in RELATIONSHIPS(path)| r.order )
```

Die RETURN-Klausel enthält neben dem Namen der beiden Produkte auch die Länge des kürzesten Pfades und eine Liste der Bestellnummern, über die zwei Produkte indirekt verbunden sind.

An dieser Stelle muss angemerkt werden, dass Cypher zwar einige Funktionalitäten zur Verarbeitung von Pfaden in Graphen aufweist (unter anderem auch die Kleene'sche Hülle bezogen auf Kantentypen), aber nicht die volle Funktionalität einer Kleene'schen Algebra auf Pfaden im Graphen abdecken kann wie in der Theorie der graphbasierten Sprachen gefordert wird. Cypher ist gleichwohl eine Sprache, die sich für den Einsatz in der Praxis gut eignet.

3.5 Eingebettete Sprachen

Die relationalen Abfrage-und Manipulationssprachen können nicht nur als selbstständige Sprachen interaktiv verwendet werden, sondern auch eingebettet in einer eigentlichen Programmiersprache (Wirtssprache). Für das Einbetten in eine Programmierumgebung müssen allerdings einige Vorkehrungen getroffen werden, auf die wir hier näher eingehen.

Das Konzept der eingebetteten Sprachen soll am Beispiel von SQL erläutert werden. Damit ein Programm durch ein SELECT-Statement eine Tabelle einlesen kann, muss es *von einem Tupel auf das nächste* zugreifen können, wozu ein Cursorkonzept benötigt wird.

3.5.1 Das Cursorkonzept

Ein CURSOR *ist ein Zeiger*, der in einer vom Datenbanksystem vorgegebenen Reihenfolge eine Menge von Tupeln durchlaufen kann. Da ein konventionelles Programm eine ganze Tabelle nicht auf einen Schlag verarbeiten kann, erlaubt das Cursorkonzept ein tabellenzeilenweises Vorgehen. Für die Selektion einer Tabelle muss ein CURSOR wie folgt im Programm definiert werden:

```
DECLARE cursor-name CURSOR FOR <SELECT-statement>
```

Auf diese Art wird eine Tabelle satzweise, d. h. Tupel um Tupel, abgearbeitet. Falls erforderlich, lassen sich gleichzeitig einige oder alle Datenwerte des jeweiligen Tupels verändern. Muss die Tabelle in einer bestimmten Sortierreihenfolge verarbeitet werden, so ist der obigen Spezifikation eine ORDER-BY-Klausel anzufügen.

In einem Programm können zu Navigierungszwecken mehrere CURSORs verwendet werden. Diese müssen deklariert und anschliessend durch OPEN- und CLOSE-Befehle aktiviert und wieder ausser Kraft gesetzt werden. Der eigentliche Zugriff auf eine Tabelle und die Übertragung der Datenwerte in die entsprechenden Programmvariablen erfolgt durch einen FETCH-Befehl, dabei müssen die Variablen, die in der Programmiersprache angesprochen werden, typenkonform zu den Feldformaten der Tabellen sein. Der FETCH- Befehl lautet:

```
FETCH cursor-name INTO host-variable {,host-variable}
```

Jeder FETCH-Befehl positioniert den CURSOR um einen Tupel weiter, entweder aufgrund der physischen Reihenfolge der zugrunde liegenden Tabelle oder anhand einer ORDER-BY-Klausel. Wird kein Tupel mehr gefunden, so wird dem Programm ein entsprechender Statuscode geliefert.

Das Cursorkonzept erlaubt es, *eine mengenorientierte Abfrage- und Manipulationssprache in eine prozedurale Wirtssprache einzubetten*. So können bei SQL dieselben

Sprachkonstrukte sowohl interaktiv (ad hoc) wie eingebettet (programmiersprachlich) verwendet werden. Beim Testen von eingebetteten Programmierteilen zeigen sich ebenfalls Vorteile, da die Testtabellen jederzeit mit der interaktiven SQL-Sprache ausgewertet und überprüft werden können.

3.5.2 Stored Procedures und Stored Functions

Seit SQL:1999 wurde im SQL-Standard die Möglichkeit vorgesehen, SQL in datenbankinternen Prozeduren und Funktionen einzubetten. Da diese auf dem Datenbankserver im Data Dictionary gespeichert werden, heissen sie *Stored Procedures* (gespeicherte Prozeduren), beziehungsweise, falls sie Werte zurückgeben, *Stored Functions* (gespeicherte Funktionen). Diese Sprachelemente erlauben die prozedurale Verarbeitung von Datensatzmengen mit CURSORs durch Fallunterscheidungen und Schleifen. Die prozeduralen Sprachelemente von SQL wurden erst spät standardisiert, weshalb sich bei vielen Herstellern unterschiedliche und proprietäre Formate etabliert haben. Deshalb ist die prozedurale Programmierung mit SQL produktspezifisch.

Als Beispiel berechnet die folgende Stored Function das erste Quartil[4] des Lohns aller Mitarbeitenden:

```
CREATE FUNCTION LohnQuartil()
RETURNS INTEGER DETERMINISTIC
BEGIN
   DECLARE cnt int;
   DECLARE i int;
   DECLARE tmpLohn int;
   DECLARE mitarbeiterCursor CURSOR FOR
      SELECT Lohn
      FROM Mitarbeiter
      ORDER BY Lohn ASC;
   SELECT COUNT(*)/4 INTO cnt FROM Mitarbeiter;
   SET i := 0;
   OPEN mitarbeiterCursor;
   mitarbeiterLoop: LOOP
      FETCH mitarbeiterCursor INTO tmpLohn;
      SET i := i + 1;
      IF i >= cnt THEN
         LEAVE mitarbeiterLoop;
      END IF;
   END LOOP;
   RETURN tmpLohn;
END
```

[4] Quartile teilen die Grundmenge in aufsteigend sortierte Viertel.

Im Wesentlichen öffnet diese Funktion einen CURSOR über die nach Lohn aufsteigend sortierte Mitarbeitertabelle, geht in einer Schleife Zeile für Zeile durch, und gibt den Wert der Spalte Lohn für diejenige Zeile zurück, für welche COUNT(*)/4 Iterationen der Schleife durchlaufen worden sind. Dies entspricht dann dem ersten Quartil, d. h. dem Wert, für den 25 % der Werte kleiner sind. Mit folgendem Ausdruck kann das Resultat der Funktion nun selektiert werden:

```
Select LohnQuartil();
```

3.5.3 JDBC

Die Sprache SQL kann auch in Java eingebettet werden. Ähnlich dem Cursorkonzept gibt es dort die Klasse *ResultSet*, welche als Zeiger auf Elemente einer Resultatmenge die iterative Verarbeitung von Datensätzen erlaubt. Mit *dem Java Database Connectivity (JDBC) Standard* ist es gelungen, eine einheitliche Schnittstelle von der Java-Sprache zu verschiedensten SQL-basierten Datenbanken herzustellen. Die meisten SQL-Datenbanken unterstützen JDBC und bieten eine entsprechende Treiberbibliothek an.

Das folgende Beispiel zeigt eine Java-Klasse, welche mit einem eingebetteten SQL-Ausdruck Daten aus einer SQL-Datenbank importiert und diese dann weiterverarbeitet:

```
public class HelloJDBC {

   public static void main(String[] args) {
      try {
         Class.forName("com.mysql.jdbc.Driver");
         Connection connection =
            DriverManager.getConnection(
               "jdbc:mysql://127.0.0.1:3306/ma",
               "root",
               "");
         Statement statement =
            connection.createStatement();
         ResultSet resultset =
            statement.executeQuery(
               "SELECT * FROM MITARBEITER");
         while (resultset.next()) {
            System.out.println(
               resultset.getString("Name"));
         }
      } catch (Exception e) {}
   }
}
```

Das simple Beispiel liefert die Namen aller Mitarbeitenden an die Konsole weiter. Vom Prinzip her kann hier jedoch eine beliebige weitere Verarbeitung der Daten mit SQL in der Wirtssprache Java stattfinden. Dafür relevant sind vor allem die vier Hauptklassen

Connection, DriverManager, Statement und ResultSet. Objekte der Klasse *Connection* stellen eine Verbindung zur Datenbank dar. Sie werden von der Klasse *DriverManager* über die statische Methode *getConnection* instanziert, wo auch die Zugangscodes mitgegeben werden. In Objekten der Klasse *Statement* können beliebige SQL-Ausrücke eingebettet werden. Die Methode *executeQuery* der Klasse Connection gibt ein ResultSet-Objekt zurück, welches eine Menge von Datensätzen als Resultat des ausgeführten SQL-Statements enthält. Gleich wie beim CURSOR kann mit Objekten der Klasse *ResultSet* eine iterative datensatzorientierte Verarbeitung der Resultattupel erfolgen.

3.5.4 Einbettung graphbasierter Sprachen

Die vorhergehenden Beispiele von eingebetteten Datenbanksprachen haben sich alle auf SQL bezogen. Auch die graphbasierten Sprachen lassen sich, da sie ebenfalls mengenorientiert sind, nach dem gleichen Prinzip mit der Verwendung des Cursorkonzepts in Wirtssprachen einbetten.

Das folgende Beispiel zeigt ein Java-Programm, welches mit eingebettetem Cypher-Code auf Daten in einer Neo4j-Datenbank zugreift und alle Produktnamen an den Bildschirm ausdruckt.

```
try (
   Transaction t = db.beginTx();
   Result result = db.execute(
      "MATCH p:Product RETURN p.productName" ))
{
   while (result.hasNext()) {
      Map<String,Object> node = result.next();
      for ( Entry<String,Object> property :
         node.entrySet()){
            System.out.println(property.getValue());
      }
   }
}
```

Ähnlich wie bei JDBC erhält man unter Ausführung eines eingebetteten Cypher-Statements eine Resultatmenge zurück, welche in einer Schleife beliebig verarbeitet werden kann.

3.6 Behandlung von Nullwerten

Beim Arbeiten mit Datenbanken kommt es immer wieder vor, dass einzelne Datenwerte für eine Tabelle nicht oder noch nicht bekannt sind. Beispielsweise möchte man einen Mitarbeitenden in die Tabelle MITARBEITER einfügen, dessen Adresse nicht vollständig

vorliegt. In solchen Fällen ist es sinnvoll, anstelle wenig aussagekräftiger oder sogar falscher Werte sogenannte Nullwerte zu verwenden.

Nullwerte
Ein Nullwert (engl. *null value*) steht für einen Datenwert einer Tabellenspalte, der (noch) nicht bekannt ist.

Ein Nullwert – symbolisch durch das Fragezeichen «?» ausgedrückt – ist nicht mit der Ziffer «Null» oder dem Wert «Blank» (Space) zu verwechseln. Diese beiden Werte drücken bei relationalen Datenbanken einen bestimmten Sachverhalt aus, wogegen der Nullwert einen Platzhalter (unbekannt / unknown) darstellt.

In Abb. 3.12 zeigen wir die Tabelle MITARBEITER mit Nullwerten bei den Merkmalen Straße und Ort. Natürlich dürfen nicht alle Merkmalskategorien Nullwerte enthalten, sonst sind Konflikte vorprogrammiert. Primärschlüssel dürfen definitionsgemäss keine Nullwerte aufweisen; im Beispiel gilt dies für die Mitarbeiternummer. Beim Fremdschlüssel «Unt» liegt die Wahl im Ermessen des Datenbankarchitekten. Maßgebend sind dessen Realitätsbeobachtungen.

Das Arbeiten mit Nullwerten ist nicht ganz unproblematisch. Der Nullwert bildet neben den Werten TRUE (1) und FALSE (0) den neuen Informationsgehalt UNKNOWN (?). Damit verlassen wir die Zweiwertlogik, dass jede Aussage entweder falsch oder wahr ist. Auch mit drei Wahrheitswerten lassen sich Wahrheitstabellen von logischen Operatoren wie AND, OR und NOT ableiten. Wie in Abb. 3.13 dargestellt, kann die Kombination von wahren oder falschen Aussagen als Resultat Nullwerte zurückgeben. Dies kann zu kontraintuitiven Resultaten führen, wie das Beispiel aus Abb. 3.12 zeigt.

Mit der Abfrage aus Abb. 3.12, die sämtliche Mitarbeitende aus der Tabelle MITARBEITER selektiert, die entweder in Liestal oder ausserhalb von Liestal wohnen, erhalten wir in der Resultattabelle nur eine Mitarbeiterteilmenge der ursprünglichen Tabelle, da einige Wohnorte von Mitarbeitenden *unbekannt* (und daher nicht *wahr*) sind. Dies widerspricht klar der herkömmlichen Logik, dass eine Vereinigung der Teilmenge «Mitarbeiter wohnhaft in Liestal» mit deren Komplement «Mitarbeiter NICHT wohnhaft in Liestal» die Menge aller Mitarbeitenden ergibt.

Die Aussagelogik mit den Werten TRUE, FALSE und UNKNOWN wird oft als Dreiwertlogik bezeichnet, da jede Aussage entweder «wahr» oder «falsch» oder «unbekannt» sein kann. Diese Logik ist weniger geläufig und stellt an die Anwendenden relationaler Datenbanken besondere Anforderungen, da Auswertungen von Tabellen mit Nullwerten schwieriger zu interpretieren sind. Deshalb wird in der Praxis oft entweder auf die Anwendung von Nullwerten verzichtet oder es werden anstelle von Nullwerten Default-Werte verwendet. In unserem Beispiel der Mitarbeitertabelle könnte die Geschäftsadresse als Default-Wert die noch unbekannte Privatadresse ersetzen. Mit der Funktion COALESCE(X, Y) wird für ein Attribut X immer dann der Wert Y eingesetzt, wenn der Wert dieses Attributs NULL ist. Müssen Nullwerte explizit zugelassen werden, können Attribute mit einem spezifischen Vergleichsoperator IS NULL bzw. IS NOT

MITARBEITER

M#	Name	Straße	Ort	Unt
M19	Schweizer	Hauptstraße	Frenkendorf	A6
M1	Meier	?	?	A3
M7	Huber	Mattenweg	Basel	A5
M4	Becker	?	?	A6

```
SELECT   Name
FROM     MITARBEITER
WHERE    Ort = ‚Liestal‘
   UNION
SELECT   *
FROM     MITARBEITER
WHERE    NOT Ort = ‚Liestal‘
```

RESULTATTABELLE

M#	Name	Straße	Ort	Unt
M19	Schweizer	Hauptstraße	Frenkendorf	A6
M7	Huber	Mattenweg	Basel	A5

Abb. 3.12 Überraschende Abfrageergebnisse beim Arbeiten mit Nullwerten

NULL auf den Wert ‚unknown' geprüft werden, um unvorhergesehene Seiteneffekte zu vermeiden.

Normalerweise sind Nullwerte bei Fremdschlüsseln unerwünscht. Als Ausnahme gelten jedoch Fremdschlüssel, die unter eine spezielle Regelung der referenziellen Integrität fallen. So kann man beispielsweise bei der Löschregel für die referenzierte Tabelle ABTEILUNG angeben, ob eventuelle Fremdschlüsselverweise auf Null gesetzt werden sollen oder nicht. Die referenzielle Integritätsregel «Nullsetzen» sagt aus, dass Fremdschlüsselwerte beim Löschen der referenzierten Tupel auf Null wechseln. Löschen wir das Tupel (A6, Finanz) in der Tabelle ABTEILUNG mit der Integritätsregel «Nullsetzen», so werden die beiden Fremdschlüssel der Mitarbeitenden Schweizer und Becker in Abb. 3.12 in der Tabelle MITARBEITER auf Null gesetzt. Diese Regel ergänzt die besprochenen Regeln der restriktiven und fortgesetzten Löschung aus Abschn. 2.3.3. Siehe dazu auch Abschn. 3.7.

OR	1	?	0
1	1	1	1
?	1	?	?
0	1	?	0

AND	1	?	0
1	1	?	0
?	?	?	0
0	0	0	0

NOT	
1	0
?	?
0	1

Abb. 3.13 Wahrheitstabellen der dreiwertigen Logik

In graphbasierten Sprachen gibt es ebenfalls Nullwerte mit den entsprechenden Funktionen. Cypher basiert auf der dreiwertigen Logik. Die Handhabung von Nullwerten ist mit IS NULL und COALESCE gleich wie bei SQL.

3.7 Integritätsbedingungen

Die Integrität einer Datenbank ist eine grundlegende Eigenschaft, die das Datenbankmanagementsystem unterstützen muss. Die Vorschriften, die bei Einfüge- oder Änderungsoperationen jederzeit gelten, werden *Integritätsregeln* (engl. *integrity constraints*) genannt. Solche Regeln werden sinnvollerweise nicht in jedem Programm einzeln, sondern einmal umfassend im Datenbankschema spezifiziert. Bei diesen Integritätsregeln unterscheidet man deklarative und prozedurale Regeln.

Die *deklarativen Integritätsregeln* (engl. *declarative integrity constraints*) werden mit Hilfe der Datendefinitionssprache bei der Erstellung einer Tabelle im CREATE TABLE Statement festgelegt. Im Beispiel der Abb. 3.14 wird bei der Tabelle ABTEILUNG der Primärschlüssel in Form einer Integritätsregel mit PRIMARY KEY spezifiziert. Analoges gilt für die Primär- und Fremdschlüsseldefinition in der Tabelle MITARBEITER.

Es gibt verschiedene Arten von deklarativen Integritätsregeln, und zwar:

- **Primärschlüsseldefinition**: Mit PRIMARY KEY wird ein eindeutiger Primärschlüssel für eine Tabelle definiert. Ein Primärschlüssel darf definitionsgemäss nie Nullwerte enthalten.
- **Fremdschlüsseldefinition**: Mit FOREIGN KEY kann ein Fremdschlüssel spezifiziert werden, der durch die Angabe REFERENCES auf die zugehörige Tabelle verweist.

- **Eindeutigkeit**: Die Eindeutigkeit eines Attributes kann durch die Angabe UNIQUE festgehalten werden. Im Gegensatz zu Primärschlüsseln können eindeutige Attributwerte auch Nullwerte enthalten.
- **Keine Nullwerte**: Mit der Angabe NOT NULL wird verhindert, dass ein Attribut Nullwerte besitzen kann. In Abb. 3.14 muss beispielsweise für jeden Mitarbeitenden ein Name mitgegeben werden, da der Name nicht leer sein darf.
- **Prüfregel***:* Eine solche Regel kann durch den CHECK-Befehl deklariert werden. Jedes Tupel in der Tabelle muss die Prüfregel erfüllen. Beispielsweise wird mit der Angabe CHECK Lohn $>$ 25000 in der Tabelle PERSONAL aus Abb. 3.14 garantiert, dass das Jahreseinkommen für jeden Angestellten mindestens 25‘000 Euro beträgt.
- **Ändern oder Löschen mit Nullsetzen**: Mit der Angabe ON UPDATE SET NULL beziehungsweise ON DELETE SET NULL wird bei einer abhängigen Tabelle deklariert, dass beim Ändern respektive Löschen des Tupels aus der Referenztabelle der Fremdschlüsselwert beim abhängigen Tupel auf Null gesetzt wird (vgl. Abschn. 2.3.3).
- **Restriktives Ändern oder Löschen**: Mit ON UPDATE RESTRICT bzw. ON DELETE RESTRICT können Referenztupel nicht geändert respektive gelöscht werden, solange sie noch abhängige Tupel besitzen (vgl. Abschn. 2.3.3).
- **Fortgesetztes Ändern oder Löschen**: Mit der Angabe ON UPDATE CASCADE bzw. ON DELETE CASCADE kann die Änderung respektive die Löschung eines Referenztupels auf die abhängigen Tupel ausgeweitet werden (vgl. Abschn. 2.3.3).

In Abb. 3.14 ist für die beiden Tabellen ABTEILUNG und MITARBEITER eine restriktive Löschregel spezifiziert. Diese Regel garantiert, dass eine bestimmte Abteilung nur dann gelöscht werden kann, wenn sie keine abhängigen Mitarbeitertupel mehr aufweist.

Der folgende Befehl

```
DELETE FROM Abteilung WHERE A# = 'A6'
```

würde somit eine Fehleranzeige ergeben, da die beiden Mitarbeitenden Schweizer und Becker in der Finanzabteilung registriert sind.

Die deklarativen Integritätsregeln können neben den Löschoperationen auch bei den Einfüge- und Änderungsoperationen zur Geltung kommen. So ergibt die Einfügeoperation

```
INSERT INTO MITARBEITER
VALUES ('M20','Kunz','Riesweg','Olten','A7')
```

eine Fehlermeldung aus. Die Abteilung A7 wird in der referenzierten Tabelle ABTEILUNG noch nicht geführt. Aufgrund der Fremdschlüsselbedingung hat das Datenbankverwaltungssystem geprüft, ob der Schlüssel A7 in der referenzierten Tabelle vorhanden ist.

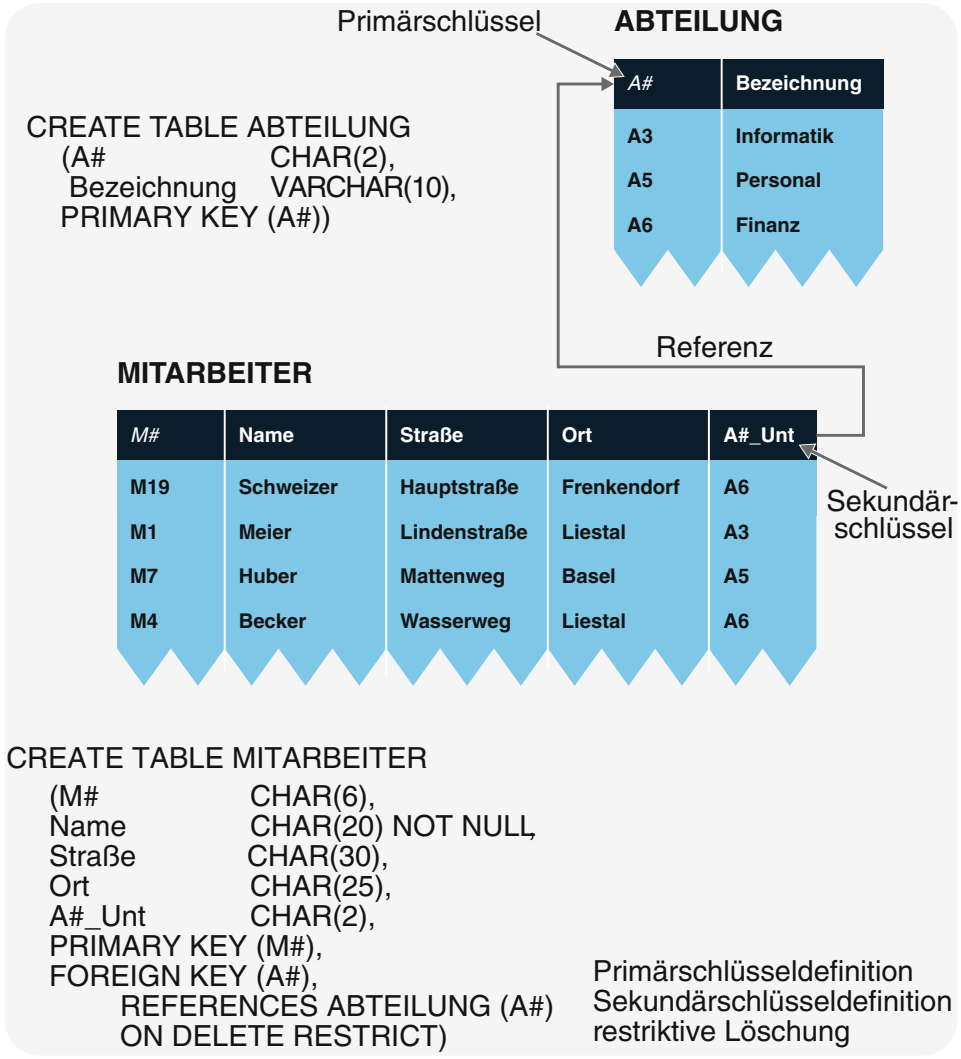

Abb. 3.14 Definition von referenziellen Integritätsregeln

Statische bzw. deklarative Integritätsbedingungen können bei der Erstellung der Tabelle (CREATE-TABLE-Befehl) angegeben werden. Bei dynamischen bzw. *prozeduralen Integritätsregeln* (engl. *procedural integrity constraints*) werden jedoch Datenbankzustände vor und nach einer Veränderung verglichen. Daher kann dies nur zur Laufzeit überprüft werden. Die Auslöser oder *Trigger* stellen eine Alternative zu den deklarativen Integritätsregeln dar, da sie die Ausführung einer Sequenz von prozeduralen

Fallunterscheidungen durch Anweisungen veranlassen. Ein Trigger ist im Wesentlichen durch die Angabe eines Auslösernamens, einer Datenbankoperation und einer Liste von Folgeaktionen definiert:

```
CREATE TRIGGER KeineKürzung    -- Auslösername
BEFORE UPDATE ON Mitarbeiter   -- Datenbankoperation
FOR EACH ROW BEGIN             -- Folgeaktion
    IF NEW.Lohn < OLD.Lohn
    THEN set NEW.Lohn = OLD.Lohn
    END IF;
END
```

Im obigen Beispiel nehmen wir an, dass die Löhne von Mitarbeitenden nicht gekürzt werden sollen. Deshalb wird mit einem Trigger vor dem Ändern der Tabelle MITARBEITER geprüft, ob der neue Lohn kleiner ist als der alte Lohn. Ist dies der Fall, ist eine Integritätsbedingung verletzt, und der neue Lohn wird auf den ursprünglichen Wert vor dem UPDATE gesetzt. In diesem Beispiel haben wir den Trigger bewusst einfach gehalten, um das Grundprinzip darzustellen. In der Praxis würde man auch eine Meldung für den Benutzer generieren.

Das Arbeiten mit Auslösern ist nicht ganz einfach, da im allgemeinen Fall einzelne Trigger weitere Trigger aufrufen können. Dabei stellt sich die Frage der Terminierung sämtlicher Folgeaktionen. Meistens ist in kommerziellen Datenbanksystemen das gleichzeitige Aktivieren von Auslösern unterbunden, damit eine eindeutige Aktionsreihenfolge garantiert bleibt und ein Trigger wohlgeordnet terminieren kann.

Die einzige *Integritätsbedingung in der graphbasierten Sprache Cypher,* welche explizit festgelegt werden kann, ist die (deklarative) Bedingung, dass der Wert eines Attributs für einen Knotentypen eindeutig sein muss. Beispielsweise kann mit folgendem Ausdruck die Regel festgelegt werden, dass der Produktname für Knoten des Typs Produkt eindeutig ist.

```
CREATE CONSTRAINT ON (p:Product)
ASSERT p.productName IS UNIQUE;
```

Die Graphdatenbank Neo4j, für welche Cypher die Sprachschnittstelle bietet, prüft jedoch alle Daten *implizit auf referenzielle Integrität*. Es müssen somit weder Primär- noch Fremdschlüssel explizit deklariert werden. Das Datenbankverwaltungssystem von Neo4J prüft in jedem Fall, dass sich Kanten immer auf existierende Knoten beziehen. Aus diesem Grund kann ein Knoten erst wieder gelöscht werden, wenn es keine Kanten gibt, welche mit ihm verbunden sind (siehe dazu auch Abschn. 3.4.1).

3.8 Datenschutzaspekte

Unter *Datenschutz* (engl. *data protection*) versteht man den Schutz der Daten vor unbefugtem Zugriff und Gebrauch. Schutzmaßnahmen sind Verfahren zur eindeutigen Identifizierung von Personen, zum Erteilen von Benutzerberechtigungen für bestimmte Datenzugriffe, aber auch kryptografische Methoden zur diskreten Speicherung oder Weitergabe von Informationen.

Im Gegensatz zum Datenschutz fallen unter den Begriff *Datensicherung* oder *Datensicherheit* (engl. *data security*) technische und softwaregestützte Maßnahmen zum Schutz der Daten vor Verfälschung, Zerstörung oder Verlust. Datensicherungsverfahren beim Sicherstellen oder Wiederherstellen von Datenbanken werden in Kap. 4 behandelt.

Das Relationenmodell vereinfacht die Implementation zuverlässiger Restriktionen für die Umsetzung des Datenschutzes. Ein wesentlicher Datenschutzmechanismus relationaler Datenbanksysteme besteht darin, den berechtigten Benutzern lediglich die für ihre Tätigkeit notwendigen Tabellen und Tabellenausschnitte zur Verfügung zu stellen. Dies wird durch *Sichten* (engl. *views*) auf Tabellen ermöglicht. Jede solche Sicht basiert entweder auf einer Tabelle oder auf mehreren physischen Tabellen und wird aufgrund einer SELECT-Anweisung definiert:

```
CREATE VIEW view-name AS <SELECT-statement>
```

In Abb. 3.15 werden anhand der Basistabelle PERSONAL zwei Beispiele von Sichten dargestellt. Die Sicht MITARBEITER zeigt alle Merkmale ausser den Lohnangaben. Bei der Sicht GRUPPE_A werden lediglich diejenigen Mitarbeitenden mit ihren Lohnangaben gezeigt, die zwischen 70‘000 und 90‘000 Euro im Jahr verdienen. Auf analoge Art können weitere Sichten festgelegt werden, um beispielsweise den Personalverantwortlichen pro Lohngruppe die vertraulichen Angaben zugänglich zu machen.

Die beiden Beispiele in Abb. 3.15 illustrieren einen bedeutenden Schutzmechanismus. Einerseits können Tabellen durch Projektionen nur ganz bestimmter Merkmalskategorien auf Benutzergruppen eingeschränkt werden. Andererseits ist eine wertabhängige Zugriffskontrolle beispielsweise auf Lohnbereiche möglich, indem die Sichten in der WHERE-Klausel entsprechend spezifiziert werden.

Auf Sichten können analog zu den Tabellen Abfragen formuliert werden. Manipulationen auf Sichten lassen sich nicht in jedem Fall auf eindeutige Art durchführen. Falls eine Sicht als Verbund aus mehreren Tabellen festgelegt wurde, wird eine Änderungsoperation vom Datenbanksystem in bestimmten Fällen abgewiesen.

Änderbare Sicht

Änderbare Sichten (engl. *updateable views*) erlauben Einfüge-, Lösch- und Aktualisierungsoperationen. Damit eine Sicht änderbar ist, müssen folgende Kriterien erfüllt sein:

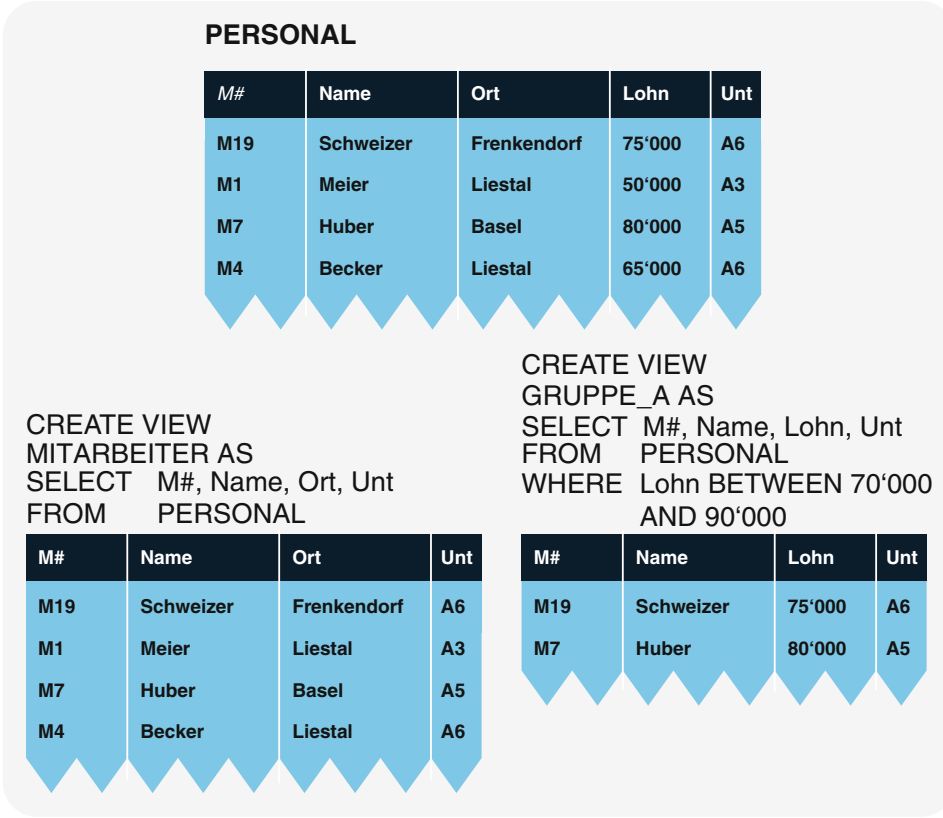

Abb. 3.15 Definition von Sichten im Umfeld des Datenschutzes

- Die Sicht bezieht sich auf eine einzige Tabelle (keine Joins erlaubt).
- Diese Basistabelle hat einen Primärschlüssel.
- Der definierende SQL-Ausdruck beinhaltet keine Operationen, welche die Anzahl Zeilen der Resultatmenge beeinflussen (z. B. *aggregate, group by, distinct*, etc.).

Wichtig ist, dass verschiedene Sichten einer bestimmten Tabelle nicht mit den jeweiligen Daten redundant zur Basistabelle verwaltet werden. Vielmehr werden nur die Definitionen der Sichten festgehalten. Erst bei einer Abfrage auf die Sicht durch eine SELECT-Anweisung werden die entsprechenden Resultattabellen aus den Basistabellen der Sicht mit den erlaubten Datenwerten aufgebaut.

Ein wirksamer Datenschutz verlangt mehr, als nur Tabellen durch Sichten einzuschränken. Auch die Funktionen für die Tabellen sollten benutzerspezifisch festgelegt werden können. Die SQL-Befehle GRANT und REVOKE dienen einer solchen Verwaltung von Benutzungsrechten (Privilegien).

Was mit GRANT vergeben wird, kann mit REVOKE zurückgenommen werden:

```
GRANT <Privileg> ON <Tabelle> TO <Benutzer>
REVOKE <Privileg> ON <Tabelle> FROM <Benutzer>
```

Der Befehl GRANT verändert die Privilegienliste derart, dass die begünstigte Benutzer Lese-, Einfüge- oder Löschoperationen auf bestimmten Tabellen oder Sichten durchführen dürfen. Entsprechend kann mit dem Befehl REVOKE ein vergebenes Recht wieder weggenommen werden.

Auf der Sicht MITARBEITER in Abb. 3.15 sollen beispielsweise nur Leserechte vergeben werden:

```
GRANT SELECT ON MITARBEITER TO PUBLIC
```

Anstelle einer Benutzerliste wird in diesem Beispiel das Leserecht mit PUBLIC bedingungslos allen Benutzern vergeben, so dass diese das Recht haben, die mit der Sicht MITARBEITER eingeschränkte Tabelle anzusehen.

Möchte man die Rechte selektiv vergeben, so könnte man zum Beispiel für die Sicht GRUPPE_A in Abb. 3.15 nur für einen bestimmten Personalverantwortlichen mit der spezifischen Benutzeridentifikation ID37289 eine Veränderungsoperation definieren:

```
GRANT UPDATE ON GRUPPE_A TO ID37289
WITH GRANT OPTION
```

Benutzer ID37289 kann die Sicht GRUPPE_A verändern und hat ausserdem aufgrund der GRANT OPTION die Möglichkeit, dieses Recht oder ein eingeschränkteres Leserecht in Eigenkompetenz weiterzugeben und später auch wieder zurückzunehmen. Mit diesem Konzept lassen sich Abhängigkeitsbeziehungen von Rechten festlegen und verwalten.

Im Unterschied zu SQL bietet *Cypher keine Sprachelemente zur Verwaltung von Zugriffsrechten* auf Ebene der abstrakten Datentypen (Knoten- oder Kantentypen). Neo4j unterstützt die Sicherung von Zugriffen auf Datenebene explizit nicht. Es gibt zwar eine Authentisierung von Benutzern mit Benutzernamen und Passwort, aber die Zugriffe erfolgen pro Benutzer jeweils auf die gesamte Datenbank. Sicherheitsmechanismen können nur auf Netzwerkebene für einen gesamten Datenbankserver festgelegt werden. Es gibt zudem die Möglichkeit, in der Programmiersprache Java spezifische Autorisationsregeln für einen Neo4j-Server zu programmieren; dies verlangt Software-Entwicklung und ist aus Sicht der Benutzerfreundlichkeit weit entfernt von den Sicherheitsmechanismen der SQL-Sprache.

Bei der Freigabe einer relationalen Abfrage- und Manipulationssprache für einen Endbenutzer darf der administrative Aufwand zur Vergabe und Rücknahme von Rechten nicht unterschätzt werden, auch wenn den Datenadministratoren die Befehle GRANT und REVOKE zur Verfügung stehen. In der Praxis zeigt sich, dass beim täglichen Änderungsdienst und bei der Überwachung der Benutzerrechte weitere Kontrollinstrumente benötigt werden. Zusätzlich verlangen interne und externe Kontroll- oder Aufsichtsorgane besondere Vorkehrungen, um den rechtmäßigen Umgang mit schützenswerten Daten jederzeit gewährleisten zu können (vgl. dazu die Pflichten der Datenschutzbeauftragten im Datenschutzgesetz).

SQL-Injection

Ein Sicherheitsaspekt, der im Zeitalter des Webs im Bereich der Datenbanken eine immer grössere Rolle spielt, ist die Verhinderung der sogenannten *SQL-Injections*. Da Webseiten oft serverseitig programmiert und an eine Datenbank angebunden werden, ist es oft so, dass diese Server-Skripts ihrerseits SQL-Code generieren, um die Schnittstelle zur Datenbank herzustellen (siehe Abschn. 3.5). Wenn dieser generierte SQL-Code Parameter enthält, die von den Benutzern eingegeben werden können (z. B. in Formularen oder als Teil der URL), kann es sein, dass schützenswerte Informationen in der Datenbank preisgegeben oder verändert werden können.

Als stark vereinfachtes Beispiel nehmen wir an, dass nach dem Einloggen in das Benutzerkonto eines Webshops die Zahlungsmethoden angezeigt werden. Die Webseite, welche die gespeicherten Zahlungsmethoden des Benutzers anzeigt, hat folgende URL:

```
http://example.net/payment?uid=117
```

Im Hintergrund gibt es ein Java-Servlet, welches die Kreditkartendaten (Kartennummer und Name), welche die Webseite mit HTML tabellarisch darstellt, aus der Datenbank holt.

```
ResultSet resultset =
   statement.executeQuery(
      "SELECT creditcardnumber, name FROM PAYMENT" +
      "WHERE uid = " + request.getParameter("uid"));
while (resultset.next()) {
   out.println("<tr><td>" +
      resultset.getString("creditcardnumber") +
      "</td> <td>" +
      resultset.getString("name") + "</td></tr>"
}
```

Zu diesem Zweck wird eine SQL-Anfrage auf die Tabelle PAYMENT dynamisch generiert, welche über die Identifikation des Benutzers (uid) parametrisiert wird. Diese Codegenerierung ist anfällig für eine SQL-Injection. Wird in der URL der Parameter uid folgendermassen ergänzt, werden sämtliche Kreditkartendaten aller Benutzer auf der Webseite angezeigt:

```
http://example.net/payment?uid=117%20OR%201=1
```

Der Grund dafür ist, dass das oben dargestellte Servlet aufgrund des GET-Parameters den folgenden SQL-Code generiert:

```
SELECT creditcardnumber, name
FROM  PAYMENT
WHERE uid = 117 OR 1=1;
```

Der zusätzlich eingefügte SQL-Code „OR 1 = 1“, die *SQL-Injection*, bewirkt also, dass der Suchfilter mit der Benutzeridentifizierung in der generierten Abfrage inaktiv wird, da 1 = 1 immer stimmt, und eine OR-Verknüpfung immer wahr ist, auch wenn nur eine der Bedingungen zutrifft. Deshalb gibt in diesem einfachen Beispiel die Webseite aufgrund dieser SQL-Injection schützenswerte Daten preis.

SQL-Injection ist eine große Sicherheitslücke, und Hackern gelingt es immer wieder, auch namhafte Webseiten über diesen Mechanismus anzugreifen. Um eine Webseite davor zu schützen, gibt es verschiedene Möglichkeiten: Erstens werden heute vermehrt NoSQL-Datenbanken wie MongoDB oder CouchDB für die Entwicklung von Webseiten verwendet, die aufgrund fehlender SQL-Schnittstelle naturgemäß keine Angriffsfläche für SQL-Injection bieten. Wird dennoch im Web-Umfeld mit SQL-Datenbanken gearbeitet, kann die Auslagerung von SQL-Codegenerierung in streng typisierte Stored Functions auf der Datenbank vorgenommen werden (siehe Abschn. 3.5.2). Im obigen Beispiel könnte eine Funktion als Input eine *Benutzer-ID als Zahl* akzeptieren, und dann als Output die Kreditkarteninformationen zurückgeben. Würde dieser Funktion der Text „OR 1 = 1“ als SQL-Injection mitgegeben, würde eine Fehlermeldung generiert.

Im Sicherheitsbereich kann also zusammenfassend gesagt werden, dass auf Ebene des Datenbanksystems die SQL-Datenbanken einen umfassenden Schutzmechanismus mit den Konstrukten CREATE VIEW, GRANT und REVOKE bieten, welche keine NoSQL-Datenbank in ähnlichem Ausmaß abdecken kann. Allerdings können diese Kontrollmechanismen im größeren Rahmen von webbasierten Informationssystemen mit SQL-Injection ausgehebelt werden. Dort bieten NoSQL-Datenbanken mit ihren direkteren API-Anbindungen einen besseren Schutz.

3.9 Literatur

Die frühen Arbeiten von Codd (1970) beschreiben sowohl das Relationenmodell als auch die Relationenalgebra. Weitere Erläuterungen über die Relationenalgebra und den Relationenkalkül finden sich in den Standardwerken von Date (2004); Elmasri und Navathe (2015) sowie Maier (1983). Ullman (1982) zeigt die Äquivalenz von Relationenalgebra und Relationenkalkül auf.

Die Sprache SQL ist aus den Forschungsarbeiten für das relationale Datenbanksystem «System R» von Astrahan et al. (1976) entstanden. QBE ist ebenfalls Mitte der Siebzigerjahre durch Zloof (1977) entwickelt worden.

Eine Übersicht über die verschiedenen Abfrage- und Manipulationssprachen findet sich im Datenbankhandbuch von Lockemann und Schmidt (1993). Sprachaspekte werden in den deutschsprachigen Werken von Saake et al. (2013); Kemper und Eickler (2013) sowie von Lang und Lockemann (1995) behandelt.

Über SQL gibt es mehrere Lehrbücher, z. B. Beaulieu (2006); Kuhlmann und Müllmerstadt (2004); Panny und Taudes (2000) oder Sauer (2002).

Darwen und Date (1997) widmen ihr Werk dem Standard von SQL. Pistor (1993) beschreibt in seinem Artikel die objektorientierten Konzepte des SQL-Standards. Das umfangreiche Werk von Melton und Simon (2002) beschreibt SQL:1999. Das Fachbuch Türker (2003) widmet sich dem Standard SQL:1999 und SQL:2003.

Im Bereich der graphbasierten Sprachen gibt es noch keine internationale Standardisierung. Eine Übersicht zu graphbasierten Sprachen bietet Wood (2012). He und Singh (2010) beschreiben GraphQL, eine Sprache, die auf einer Graphalgebra basiert, und die mindestens gleich ausdrucksstark ist wie die Relationenalgebra. Das bisher einzige Buch, welches die kommerzielle Sprache Cypher beschreibt, ist von Panzarino (2014). Weitere Quellen zu Cypher finden sich auf der Webseite des Herstellers Neo4j.

Konsistenzsicherung

4

4.1 Mehrbenutzerbetrieb

Unter dem Begriff Konsistenz oder Integrität einer Datenbank versteht man den Zustand widerspruchsfreier Daten. Integritätsbedingungen (Abschn. 2.4.3 und 3.7) sollen garantieren, dass bei Einfüge- oder Änderungsoperationen die Konsistenz der Daten jederzeit gewährleistet bleibt.

Eine Schwierigkeit ergibt sich aus der Tatsache, dass mehrere Benutzer gleichzeitig auf eine Datenbank zugreifen und gegebenenfalls Daten verändern. Dabei können Konfliktsituationen betreffend gegenseitiger Behinderung (Deadlocks) oder gar Konsistenzverletzungen entstehen. Je nach Anwendungsfall sind Verstöße gegen Konsistenzregeln nicht hinnehmbar. Das klassische Beispiel dazu sind Buchungstransaktionen aus dem Bankenumfeld, bei welchen die Regeln der doppelten Buchhaltung jederzeit gewährleistet sind und nicht verletzt werden dürfen (siehe Abschn. 4.2.2).

Transaktionsverwaltungen garantieren, dass konsistente Datenbankzustände immer in konsistente Datenbankzustände überführt werden. Ein Transaktionssystem arbeitet nach dem Alles-oder-Nichts-Prinzip. Es wird damit ausgeschlossen, dass Transaktionen nur teilweise Änderungen auf der Datenbank ausführen. Entweder werden alle gewünschten Änderungen ausgeführt oder keine Wirkung auf der Datenbank erzeugt. Mit der Hilfe von pessimistischen oder optimistischen Synchronisationsverfahren wird garantiert, dass die Datenbank jederzeit in einem konsistenten Zustand verbleibt.

Bei umfangreichen Webanwendungen hat man erkannt, dass die Konsistenzforderung nicht in jedem Fall anzustreben ist. Der Grund liegt darin, dass man aufgrund des sogenannten CAP-Theorems nicht alles gleichzeitig haben kann: Konsistenz (engl. *consistency*), Verfügbarkeit (engl. *availability*) und Ausfalltoleranz (engl. *partition tolerance*). Setzt man beispielsweise auf Verfügbarkeit und Ausfalltoleranz, so muss man zwischenzeitlich inkonsistente Datenbankzustände in Kauf nehmen.

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_4

Im Folgenden wird zuerst das klassische Transaktionskonzept in Abschn. 4.2 erläutert, das auf Atomarität (engl. *atomicity*), Konsistenz (engl. *consistency*), Isolation (engl. *isolation*) und Dauerhaftigkeit (engl. *durability*) setzt und als ACID-Prinzip bekannt ist. Danach wird in Abschn. 4.3 das bereits erwähnte CAP-Theorem und die leichtere Variante für Konsistenzgewährung unter dem Kürzel BASE (Basically Available, Soft State, Eventually Consistent) erläutert. Hier wird erlaubt, dass replizierte Rechnerknoten zwischenzeitlich unterschiedliche Datenversionen halten und erst zeitlich verzögert aktualisiert werden. Abschnitt 4.4 stellt die beiden Ansätze ACID und BASE einander gegenüber. Bibliographische Angaben folgen in Abschn. 4.5.

4.2 Transaktionskonzept

4.2.1 ACID

Die Integrität der Daten zu gewährleisten ist eine wichtige Forderungen aus der Sicht vieler Datenbankanwendungen. Die *Transaktionenverwaltung* eines Datenbanksystems dient dazu, *mehreren Benutzern ein konfliktfreies Arbeiten zu ermöglichen*. Dabei dürfen Änderungen in der Datenbank nach außen erst sichtbar werden, wenn die von den Benutzern definierten Integritätsbedingungen alle respektiert sind.

Unter dem Begriff der *Transaktion* (engl. *transaction*) versteht man an Integritätsregeln gebundene Datenbankoperationen, die Datenbankzustände konsistenzerhaltend nachführen. Präziser ausgedrückt: eine Transaktion ist eine Folge von Operationen, die atomar, konsistent, isoliert und dauerhaft sein muss.

- **Atomarität** (A = Atomicity): Eine Transaktion wird entweder komplett durchgeführt, oder sie hinterlässt keine Spuren ihrer Wirkung auf der Datenbank. Die von einzelnen Operationen erzeugten Zwischenzustände einer bestimmten Transaktion sind für die übrigen konkurrierenden Transaktionen nicht ersichtlich. In diesem Sinne bildet die Transaktion eine *Einheit für die Rücksetzbarkeit* nicht abgeschlossener Transaktionen (vgl. dazu Abschn. 4.2.5).
- **Konsistenz** (C = Consistency): Während der Transaktion mögen zwar einzelne Konsistenzbedingungen zeitweise verletzt werden, bei Transaktionsende müssen jedoch alle wieder erfüllt sein. Eine Transaktion bewirkt also die Überführung einer Datenbank von einem konsistenten Zustand in einen anderen und garantiert die Widerspruchsfreiheit der Daten. Sie wird als *Einheit zur Konsistenzerhaltung* aufgefasst.
- **Isolation** (I = Isolation): Das Prinzip der Isolation verlangt, dass gleichzeitig ablaufende Transaktionen dieselben Resultate wie im Falle einer Einbenutzerumgebung

erzeugen. Falls einzelne Transaktionen isoliert von parallel dazu ablaufenden Transaktionen sind, bleiben diese vor ungewollten Seiteneffekten geschützt. Die Transaktion gilt damit als *Einheit für die Serialisierbarkeit*.

- **Dauerhaftigkeit** (D = Durability): Datenbankzustände müssen so lange gültig und erhalten bleiben, bis sie von Transaktionen verändert werden. Bei Programmfehlern, Systemabbrüchen oder Fehlern auf externen Speichermedien garantiert die Dauerhaftigkeit die Wirkung einer korrekt abgeschlossenen Transaktion. Von den Wiederanlauf- und Wiederherstellungsverfahren von Datenbanken her gesehen kann jede Transaktion als *Recovery-Einheit* aufgefasst werden (vgl. Abschn. 4.2.5).

Die vier Begriffe *Atomarität* (A), *Konsistenz* (C), *Isolation* (I) und *Dauerhaftigkeit* (D) beschreiben das sogenannte *ACID-Prinzip einer Transaktion*. Dieses ist für einige Datenbanksysteme grundlegend und garantiert allen Anwendenden, konsistente Datenbankzustände in ebensolche überführen zu können. Zwischenzeitlich inkonsistente Zustände bleiben nach außen unsichtbar und werden im Fehlerfall rückgängig gemacht.

Um eine Folge von Operationen als Transaktion zu deklarieren, sollten die Datenbankoperationen durch ein BEGIN TRANSACTION und durch ein END_OF_TRANSACTION gekennzeichnet werden.[1] Start und Ende einer Transaktion signalisieren dem Datenbanksystem, welche Operationen eine Einheit bilden und durch das ACID-Prinzip geschützt werden müssen.

Mit dem SQL-Befehl COMMIT werden die in der Transaktion vorgenommenen Änderungen festgeschrieben. Sie bleiben dauerhaft bis zu einer nächsten, erfolgreich abschließenden Transaktion. Kommt es während der Transaktion zu einem Fehler, kann die Transaktion mit dem SQL-Befehl ROLLBACK vollständig widerrufen werden.

Nach dem SQL-Standard kann der Grad an vom Datenbanksystem durchgesetzter Konsistenz durch die Einstellung einer Isolationsstufe (Isolation) mit folgendem Befehl konfiguriert werden:

```
SET TRANSACTION ISOLATION LEVEL <Isolationsstufe>
```

Es gibt vier Isolationsstufen. Diese reichen von READ UNCOMMITTED (keine Konsistenzsicherung) über READ COMMITTED (nur festgeschriebene Änderungen werden von anderen Transaktionen gelesen) und REPEATABLE READ (Leseanfragen geben wiederholt dasselbe Resultat) bis hin zur Isolationsstufe SERIALIZABLE, welche die volle serialisierbare ACID-Konsistenz durchsetzt.

[1] Im SQL-Standard werden Transaktionen implizit durch SQL-Anweisungen begonnen und durch COMMIT abgeschlossen. Sie lassen sich auch explizit mit START TRANSACTION aufrufen

4.2.2 Serialisierbarkeit

Bei der Beschreibung von Betriebssystemen und Programmiersprachen kommt der Koordination (Synchronisation) aktiver Prozesse und dem wechselseitigen Ausschluss konkurrierender Prozesse eine große Bedeutung zu. Auch bei Datenbanksystemen müssen konkurrierende Zugriffe auf dieselben Datenobjekte serialisiert werden, da die einzelnen Datenbankanwendenden unabhängig voneinander arbeiten möchten.

Prinzip der Serialisierbarkeit
Ein System gleichzeitig ablaufender Transaktionen heißt *korrekt synchronisiert, wenn es eine serielle Ausführung gibt, die denselben Datenbankzustand erzeugt.*

Bei parallel ablaufenden Transaktionen garantiert das Prinzip der Serialisierbarkeit, dass die Resultate auf den Datenbanken identisch sind, gleichgültig ob die Transaktionen streng nacheinander ausgeführt worden sind oder nicht. Um Bedingungen zur Serialisierbarkeit festlegen zu können, gilt bei den einzelnen Transaktionen unser Augenmerk den READ- und WRITE-Operationen, die das *Lesen und Schreiben von Datensätzen auf der Datenbank* bewerkstelligen.

Das klassische Beispiel zur Illustration konkurrierender Transaktionen stammt aus dem Bankbereich. Bei Buchungstransaktionen lautet die grundlegende Integritätsbedingung, dass Kontobelastungen und -gutschriften sich gegenseitig ausgleichen müssen. Abbildung 4.1 zeigt zwei parallel ablaufende Buchungstransaktionen mit ihren READ- und WRITE-Operationen in zeitlicher Abfolge. Jede Buchungstransaktion verändert für sich betrachtet die Gesamtsumme der Bestände der Konten a, b und c nicht. So schreibt die Transaktion TRX_1 dem Konto a 100 Währungseinheiten gut und belastet gleichzeitig das Gegenkonto b mit 100 Währungseinheiten. Entsprechendes gilt für die Buchungstransaktion TRX_2 mit dem Konto b und dem Gegenkonto c für den Betrag von 200 Währungseinheiten. Beide Buchungstransaktionen erfüllen somit die Integritätsbedingung der Buchführung, da sich die Salden zu Null aufheben.

Bei der gleichzeitigen Ausführung der beiden Buchungstransaktionen hingegen entsteht ein *Konflikt*: Die Transaktion TRX_1 übersieht die von TRX_2 vorgenommene Gutschrift b:=b + 200,[2] da diese Wertveränderung nicht sofort zurückgeschrieben wird, und liest im Konto b einen «falschen» Wert. Nach erfolgreichem Abschluss der beiden Buchungstransaktionen enthält das Konto a den ursprünglichen Wert plus 100 Einheiten (a + 100), b hat sich um 100 Einheiten verringert (b-100) und c ist um 200 Einheiten gekürzt worden (c-200). Die Summe von Belastungen und Gutschriften ist nicht konstant geblieben, und die Integritätsbedingung ist verletzt, da im Konto b der Wert b + 20 von der Transaktion TRX_1 übersehen statt verrechnet worden ist.

[2] Unter der Zuweisung b:= b + 200 versteht man, dass der aktuelle Bestand des Kontos b um 200 Währungseinheiten erhöht wird.

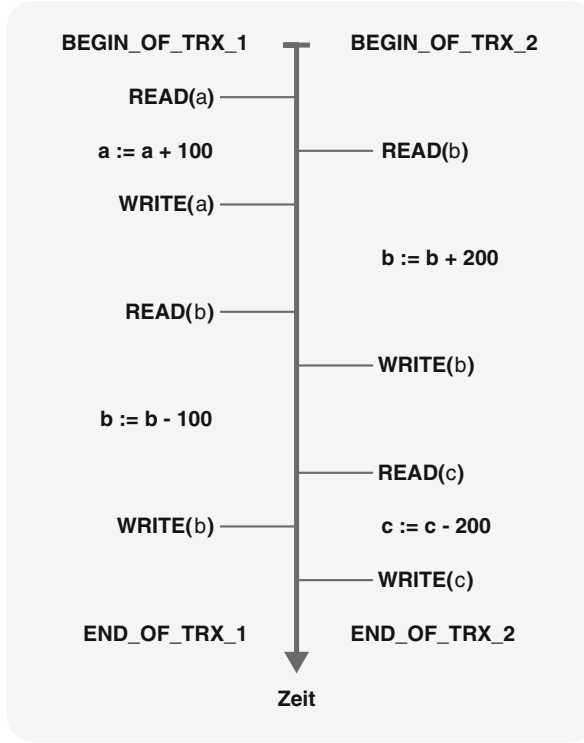

Abb. 4.1 Konfliktträchtige Buchungstransaktionen

Wie sind nun Konfliktsituationen zu erkennen? Aus der Menge aller Transaktionen führt der Weg dieser Untersuchung jeweils über diejenigen READ- und WRITE-Operationen, die sich auf ein bestimmtes Objekt, d. h. einen einzelnen Datenwert, einen Datensatz, eine Tabelle oder im Extremfall sogar eine ganze Datenbank beziehen. Von der *Granularität* (der relativen Größe) dieser Objekte hängt es ab, wie gut die herausgepflückten Transaktionen parallelisiert werden können. Je größer die Granularität des Objektes gewählt wird, desto kleiner wird der Grad der Parallelisierung von Transaktionen und umgekehrt. Die objektwirksamen READ- und WRITE-Operationen aus unterschiedlichen Transaktionen werden deshalb im sogenannten *Logbuch* (engl. *log*) des Objektes x, im LOG(x), festgehalten. Das Logbuch LOG(x) eines bestimmten Objektes x listet in zeitlicher Abfolge alle READ- und WRITE-Operationen auf, die auf das Objekt x zugreifen.

Als Beispiel für die parallelen Buchungstransaktionen TRX_1 und TRX_2 wählen wir die einzelnen Konten a, b und c als Objektgrößen. Wie in Abb. 4.2 dargestellt, erhält das Logbuch für das Objekt b beispielsweise vier Einträge (vgl. dazu auch Abb. 4.1). Zuerst liest Transaktion TRX_2 den Datenwert b, anschließend liest TRX_1 denselben Wert, noch bevor die Transaktion TRX_2 den veränderten Datenwert b zurückschreibt. Den letzten Eintrag ins Logbuch verursacht die Transaktion TRX_1, die mit ihrem veränderten Wert b jenen der Transaktion TRX_2 in der Datenbank überschreibt. Eine Auswertung der

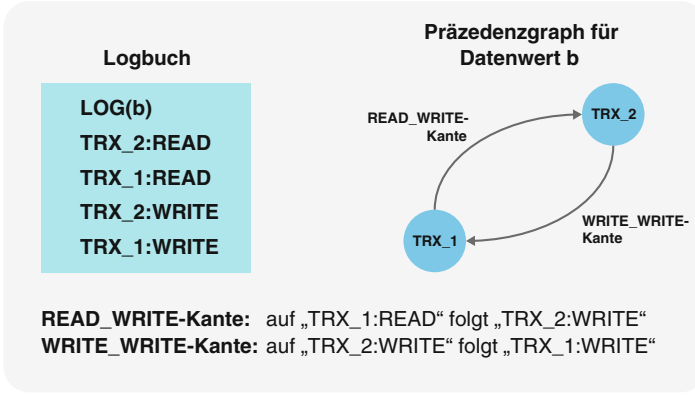

Abb. 4.2 Auswertung des Logbuches anhand des Präzedenzgraphen

Logbücher erlaubt uns nun auf einfache Weise, Konflikte bei konkurrierenden Transaktionen zu analysieren.

Der sogenannte *Präzedenzgraph* (engl. *precedence graph*) stellt die Transaktionen als Knoten und die möglichen READ_WRITE- oder WRITE_WRITE-Konflikte durch gerichtete Kanten (gebogene Pfeile) dar. Bezogen auf ein bestimmtes Objekt kann ein auf ein READ oder WRITE folgendes WRITE zu einem Konflikt führen. Hingegen gilt allgemein, dass mehrmaliges Lesen nicht konfliktträchtig ist. Aus diesem Grund hält der Präzedenzgraph keine READ_ READ-Kanten fest.

Abbildung 4.2 zeigt für die beiden Buchungstransaktionen TRX_1 und TRX_2 neben dem Logbuch des Objektes b auch den zugehörigen Präzedenzgraphen. Gehen wir vom Knoten TRX_1 aus, so folgt auf ein READ des Objektes b ein WRITE desselben durch Transaktion TRX_2, dargestellt durch eine gerichtete Kante vom Knoten TRX_1 zum Knoten TRX_2. Vom Knoten TRX_2 aus erhalten wir gemäß Logbuch eine WRITE_WRITE-Kante zum Knoten TRX_1, da auf ein WRITE von TRX_2 ein weiteres WRITE desselben Objektes b von TRX_1 folgt. Der Präzedenzgraph ist also zyklisch oder kreisförmig, da von einem beliebigen Knoten ausgehend ein gerichteter Weg existiert, der zum Ursprung zurückführt. Diese zyklische Abhängigkeit zwischen den beiden Transaktionen TRX_1 und TRX_2 zeigt klar, dass sie nicht serialisierbar sind.

Serialisierbarkeitskriterium
Eine Menge von Transaktionen ist *serialisierbar*, wenn die zugehörigen Präzedenzgraphen *keine Zyklen* aufweisen.

Das Serialisierbarkeitskriterium besagt, dass die verschiedenen Transaktionen in einer Mehrbenutzerumgebung dieselben Resultate liefern wie in einer Einbenutzerumgebung. Zur Gewährleistung der Serialisierbarkeit verhindern *pessimistische Verfahren* von vornherein, dass überhaupt Konflikte bei parallel ablaufenden Transaktionen entstehen

können. *Optimistische Verfahren* nehmen Konflikte in Kauf, beheben diese jedoch durch Zurücksetzen der konfliktträchtigen Transaktionen im Nachhinein.

4.2.3 Pessimistische Verfahren

Eine Transaktion kann sich gegenüber anderen absichern, indem sie durch Sperren die zu lesenden oder zu verändernden Objekte vor weiteren Zugriffen schützt. *Exklusive Sperren* (engl. *exclusive locks*) sind solche, die ein bestimmtes Objekt ausschließlich von einer Transaktion bearbeiten und die übrigen konkurrierenden Transaktionen abweisen oder warten lassen. Sind solche Sperren gesetzt, müssen die übrigen Transaktionen warten, bis die entsprechenden Objekte wieder freigegeben sind.

In einem *Sperrprotokoll* (engl. *locking protocol*) wird festgehalten, auf welche Art und Weise Sperren verhängt bzw. aufgehoben werden. Falls Sperren zu früh oder leichtsinnig zurückgegeben werden, können nichtserialisierbare Abläufe entstehen. Auch muss verhindert werden, dass mehrere Transaktionen sich gegenseitig blockieren und eine sogenannte Verklemmung oder Blockierung (engl. *deadlock*) heraufbeschwören.

Für das exklusive Sperren von Objekten sind die beiden Operationen LOCK und UNLOCK notwendig. Grundsätzlich muss jedes Objekt gesperrt werden, bevor eine Transaktion darauf zugreift. Falls ein Objekt x durch eine Sperre LOCK(x) geschützt ist, kann dieses von keiner anderen Transaktion gelesen oder verändert werden. Erst nach Aufheben der Sperre für Objekt x durch UNLOCK(x) kann eine andere Transaktion erneut eine Sperre erwirken.

Normalerweise unterliegen Sperren einem wohldefinierten Protokoll und können nicht beliebig angefordert und zurückgegeben werden:

Zweiphasen-Sperrprotokoll
Das Zweiphasen-Sperrprotokoll (engl. *two-phase locking protocol*) untersagt einer Transaktion, *nach dem ersten UNLOCK (Entsperren) ein weiteres LOCK (Sperren) zu verlangen*.

Mit Hilfe dieses Sperrprotokolls läuft eine Transaktion immer in zwei Phasen ab: In der *Wachstumsphase* werden sämtliche Sperren angefordert und errichtet, in der *Schrumpfungsphase* werden die Sperren sukzessive wieder freigegeben. Bei einer Transaktion mit Zweiphasen-Sperrprotokoll dürfen also innerhalb der Wachstumsphase nur LOCKs nach und nach oder alle auf einmal gesetzt, jedoch nie freigegeben werden. Erst in der Schrumpfungsphase können UNLOCKs stufenweise oder insgesamt am Ende der Transaktion wieder ausgegeben werden. Das Zweiphasen-Sperrprotokoll verbietet somit ein Durchmischen von Errichten und Freigeben von Sperren.

Abbildung 4.3 illustriert für die Buchungstransaktion TRX_1 ein mögliches Zweiphasen-Sperrprotokoll. In der Wachstumsphase wird nacheinander das Konto a wie das Gegenkonto b gesperrt, bevor beide Konten sukzessive wieder freigegeben werden. Bei

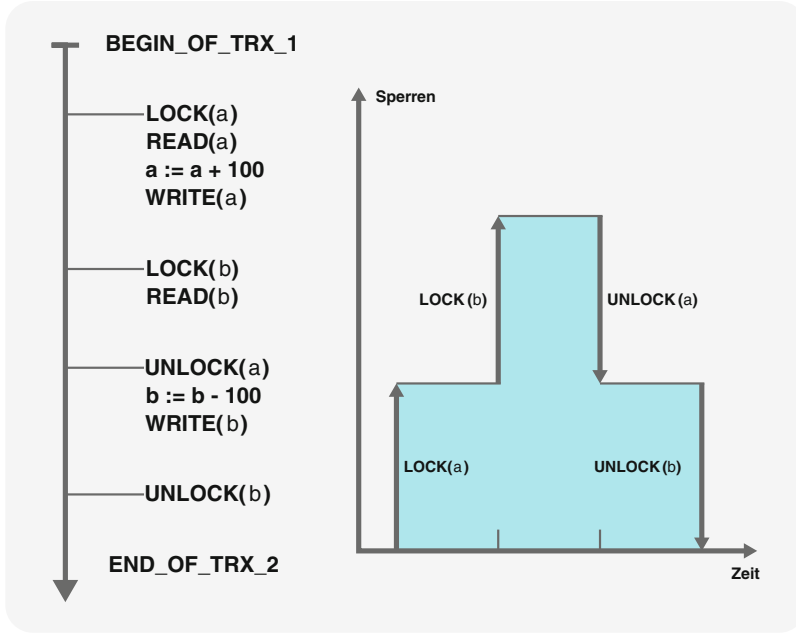

Abb. 4.3 Beispiel für ein Zweiphasen-Sperrprotokoll der Transaktion TRX_1

diesem Beispiel wäre auch möglich, beide Sperren gleich zu Beginn der Transaktion zu verfügen, anstatt sie im zeitlichen Ablauf nacheinander errichten zu lassen. Analog könnten die beiden Sperren auch am Ende der Transaktion TRX_1 nicht gestaffelt, sondern insgesamt aufgehoben werden.

Da die Wachstumsphase schrittweise für die Objekte a und b Sperren erwirkt und die Schrumpfungsphase diese schrittweise wieder freigibt, wird der Grad der Parallelisierung der Transaktion TRX_1 erhöht. Würden die beiden Sperren zu Beginn gesetzt und erst am Ende der Transaktion wieder zurückgegeben, müssten konkurrierende Transaktionen während der gesamten Verarbeitungszeit von TRX_1 auf die Freigabe der Objekte a und b warten.

Allgemein gilt, dass das Zweiphasen-Sperrprotokoll die Serialisierbarkeit parallel ablaufender Transaktionen garantiert.

Pessimistische Synchronisation (engl. *pessimistic concurrency control*)
Jede Menge konkurrienter Transaktionen ist dank Anwendung des Zweiphasen-Sperrprotokolls serialisierbar.

Aufgrund der strikten Trennung der Wachstums- von der Schrumpfungsphase lässt sich zeigen, dass das Zweiphasen-Sperrprotokoll zyklische Abhängigkeiten in sämtlichen Präzedenzgraphen von vornherein verhindert; die konkurrierenden Transaktionen bleiben konfliktfrei. Für die beiden Buchungstransaktionen TRX_1 und TRX_2 bedeutet dies,

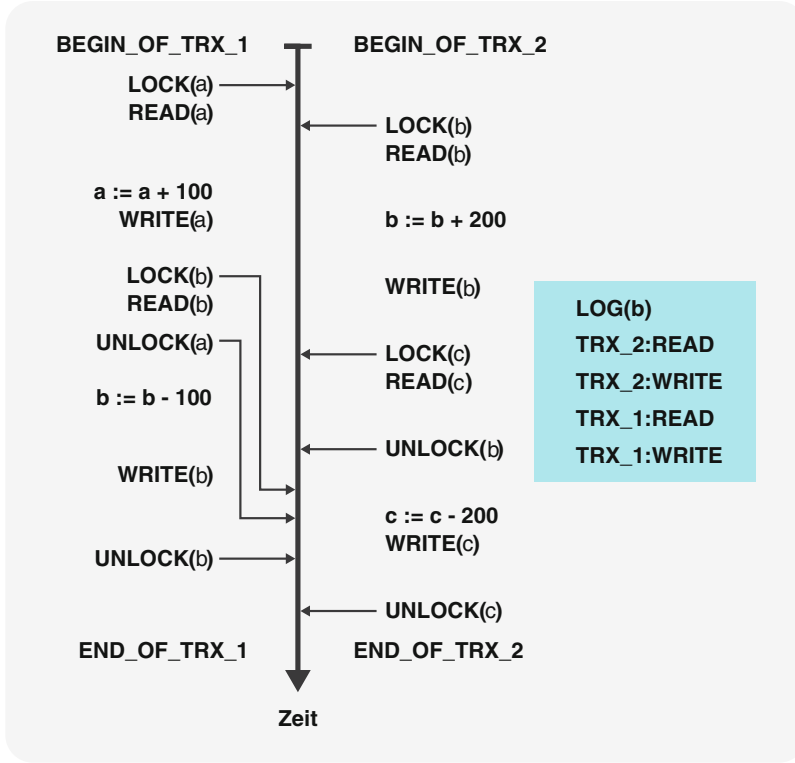

Abb. 4.4 Konfliktfreie Buchungstransaktionen

dass sie bei optimaler Organisation von Sperren und Entsperren parallelisiert werden können, ohne dass die Integritätsbedingung verletzt wird.

Abbildung 4.4 untermauert unsere Behauptung, dass die beiden Transaktionen TRX_1 und TRX_2 konfliktfrei ablaufen können. Dazu werden LOCKs und UNLOCKs nach den Regeln des Zweiphasen-Sperrprotokolls gesetzt. Damit kann beispielsweise das von TRX_2 gesperrte Konto b erst in der Schrumpfungsphase wieder freigegeben werden, und TRX_1 muss beim Anfordern der Sperre für b warten. Sobald TRX_2 das Konto b durch UNLOCK(b) entsperrt, fordert TRX_1 das Konto b an. Diesmal liest die Transaktion TRX_1 den «richtigen» Wert von b, nämlich b + 200. Die beiden Transaktionen TRX_1 und TRX_2 können somit parallel ausgeführt werden.

Das Zweiphasen-Sperrprotokoll bewirkt im zeitlichen Ablauf von TRX_1 zwar eine Verzögerung, aber nach Ablauf der beiden Transaktionen bleibt die Integrität erhalten. Das Konto a hat sich um 100 Einheiten erhöht (a + 100), das Konto b ebenfalls (b + 100), und das Konto c wurde um 200 Einheiten reduziert (c-200). Die Summe der Bestände der einzelnen Konten hat sich somit nicht verändert.

Der Vergleich des in Abb. 4.4 gegebenen Logbuches LOG(b) des Kontos b mit dem früher diskutierten Logbuch aus Abb. 4.2 zeigt einen wesentlichen Unterschied: Je ein Lesen (TRX_2: READ) und ein Schreiben (TRX_2: WRITE) wird jetzt strikt zuerst durch TRX_2 durchgeführt, bevor TRX_1 die Kontrolle über das Konto b erhält und ebenfalls Lesen (TRX_1: READ) und Schreiben (TRX_1:WRITE) darf. Der zugehörige Präzedenzgraph enthält weder READ_WRITE- noch WRITE_WRITE-Kanten zwischen den Knoten TRX_2 und TRX_1, er bleibt also zyklenfrei. Die beiden Buchungstransaktionen erfüllen damit die Integritätsbedingung.

Bei vielen Datenbankanwendungen verbietet die Forderung nach hoher Parallelität, gleich ganze Datenbanken oder Tabellen als Sperreinheiten zu verwenden. Man definiert deshalb kleinere Sperrgrößen, wie beispielsweise einen Datenbankausschnitt, einen Tabellenteil, ein Tupel oder sogar einen Datenwert. *Sperrgrößen* werden vorteilhaft so festgelegt, dass sie bei der Sperrverwaltung *hierarchische Abhängigkeiten* zulassen. Sperren wir beispielsweise eine Menge von Tupeln für eine bestimmte Transaktion, so dürfen während der Sperrzeit die übergeordneten Sperreinheiten wie Tabelle oder zugehörige Datenbank von keiner anderen Transaktion in vollem Umfang blockiert werden. Falls ein Objekt mit einer exklusiven Sperre versehen wird, können mit Hilfe einer Sperrhierarchie die übergeordneten Objekte automatisch evaluiert und entsprechend gekennzeichnet werden.

Neben Sperrhierarchien sind verschiedene Sperrmodi von Bedeutung. Die einfachste Klassifizierung von Sperren beruht auf der Unterscheidung von Lese- und Schreibsperren. Eine Lesesperre (engl. *shared lock*) erlaubt einer Transaktion nur den lesenden Zugriff auf das Objekt. Fordert eine Transaktion hingegen eine Schreibsperre (engl. *exclusive lock*) an, dann darf sie lesend und schreibend auf das Objekt zugreifen.

Ein weiteres pessimistisches Verfahren, das Serialisierbarkeit gewährleistet, ist die Vergabe von Zeitstempeln, um aufgrund des Alters von Transaktionen streng geordnet die Objektzugriffe durchführen zu können. Solche Zeiterfassungsverfahren erlauben, die zeitliche Reihenfolge der einzelnen Operationen der Transaktionen einzuhalten und damit Konflikte zu vermeiden.

4.2.4 Optimistische Verfahren

Bei *optimistischen Verfahren* geht man davon aus, dass Konflikte konkurrierender Transaktionen selten vorkommen. Man verzichtet von vornherein auf das Setzen von Sperren, um den Grad der Parallelität zu erhöhen und die Wartezeiten zu verkürzen. Bevor Transaktionen erfolgreich abschließen, werden rückwirkend Validierungen durchgeführt.

Transaktionen mit *optimistischer Synchronisation* durchlaufen drei Phasen, und zwar *eine Lese-, eine Validierungs- und eine Schreibphase*. Ohne irgendwelche präventive Sperren zu setzen, werden in der Lesephase alle benötigten Objekte gelesen und in einem transaktionseigenen Arbeitsbereich gespeichert und verarbeitet. Nach Abschluss der Verarbeitung werden in der Validierungsphase die Objekte dahingehend geprüft, ob die Veränderungen nicht in Konflikt mit anderen Transaktionen stehen. Ziel dabei ist, die momentan aktiven Transaktionen auf Konfliktfreiheit zu überprüfen. Behindern sich zwei

Transaktionen gegenseitig, so wird die in der Validierungsphase stehende Transaktion zurückgestellt. Im Falle einer erfolgreichen Validierung werden durch die Schreibphase die Änderungen aus dem Arbeitsbereich der Transaktion in die Datenbank eingebracht.

Mit Hilfe transaktionseigener Arbeitsbereiche wird bei den optimistischen Verfahren die Parallelität erhöht. Lesende Transaktionen behindern sich gegenseitig nicht. Erst wenn sie Werte zurückschreiben wollen, ist Vorsicht geboten. Die Lesephasen verschiedener Transaktionen können deshalb parallel ablaufen, ohne dass Objekte durch irgendwelche Sperren blockiert sind. Dafür muss in der Validierungsphase geprüft werden, ob die im Arbeitsbereich eingelesenen Objekte überhaupt gültig sind, also mit der Wirklichkeit in der Datenbank noch übereinstimmen.

Der Einfachheit halber wird vorausgesetzt, dass sich die Validierungsphasen verschiedener Transaktionen keinesfalls überlappen. Hierfür wird der Zeitpunkt hervorgehoben, zu welchem die Transaktion in die Validierungsphase tritt. Dadurch lassen sich sowohl die Startzeiten der Validierungsphasen als auch die Transaktionen selbst zeitlich ordnen. Sobald eine Transaktion in die Validierungsphase tritt, wird die Serialisierbarkeit geprüft.

Bei optimistischer Synchronisation wird betreffend Serialisierbarkeit wie folgt vorgegangen: Mit TRX_t sei die zu überprüfende Transaktion, mit TRX_1 bis TRX_k seien alle parallel zu TRX_t laufenden Transaktionen bezeichnet, die während der Lesephase von TRX_t bereits validiert haben. Alle übrigen fallen außer Betracht, da sämtliche Transaktionen streng nach der Eintrittszeit in die Validierungsphase geordnet sind. Hingegen sind die von TRX_t gelesenen Objekte zu überprüfen, sie könnten ja in der Zwischenzeit von den kritischen Transaktionen TRX_1 bis TRX_k bereits verändert worden sein. Wir bezeichnen die von TRX_t gelesene Objektmenge mit dem Ausdruck READ_SET (TRX_t) und die von den übrigen Transaktionen geschriebene Objektmenge mit WRITE_SET(TRX_1,. . .,TRX_k) und erhalten das folgende Serialisierbarkeitskriterium:

Optimistische Synchronisation (engl. optimistic concurrency control)
Bei optimistischer Synchronisation müssen die Mengen READ_ SET(TRX_t) und WRITE_SET(TRX_1,. . .,TRX_k) *disjunkt* sein, damit die Transaktion TRX_t serialisierbar bleibt.

Als Beispiel können wir wiederum die beiden ursprünglichen Buchungstransaktionen TRX_1 und TRX_2 von Abb. 4.1 heranziehen und dabei voraussetzen, dass TRX_2 vor TRX_1 validiert hat. Ist in diesem Fall nun TRX_1 serialisierbar oder nicht? Um diese Frage zu beantworten, bemerken wir (Abb. 4.5), dass das von TRX_1 gelesene Objekt b von der Transaktion TRX_2 bereits zurückgeschrieben worden ist und in der Schreibmenge WRITE_SET(TRX_2) liegt. Die Lesemenge READ_SET(TRX_1) und die Schreibmenge WRITE_SET(TRX_2) überlappen sich, was das Prüfkriterium zur Serialisierbarkeit verletzt. Die Buchungstransaktion TRX_1 muss nochmals gestartet werden.

Eine Verbesserung des optimistischen Verfahrens bringt die präventive Garantie von Disjunktheit der Mengen READ_SET und WRITE_SET. Dabei wird in der Validierungsphase der Transaktion TRX_t geprüft, ob diese eventuell Objekte verändert, die bereits

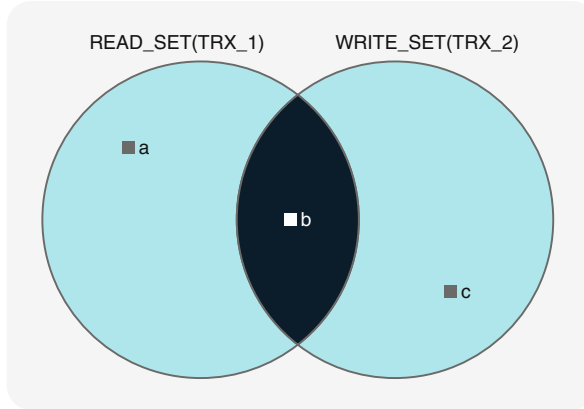

Abb. 4.5 Serialisierbarkeitsbedingung für Transaktion TRX_1 nicht erfüllt

von anderen Transaktionen gelesen worden sind. Bei dieser Prüfvariante bleibt der Validierungsaufwand auf Änderungstransaktionen beschränkt.

4.2.5 Fehlerbehandlung

Beim Betrieb einer Datenbank können verschiedene Fehler auftreten, die normalerweise durch das Datenbanksystem selbst entschärft oder behoben werden können. Einige Fehlersituationen wie Integritätsverletzungen oder Zusammenbrüche im Transaktionenverkehr sind bei der Behandlung parallel ablaufender Transaktionen bereits zur Sprache gekommen. Andere Fehler können durch das Betriebssystem oder durch die Hardware verursacht werden. Beispielsweise kann es vorkommen, dass die Daten nach einem Speicherfehler auf einem externen Medium unlesbar bleiben.

Das *Wiederherstellen eines korrekten Datenbankzustandes nach einem Fehlerfall* steht unter dem Begriff Recovery. Beim Recovery ist es wesentlich zu wissen, wo ein Fehler aufgetreten ist: im Anwendungsprogramm, in der Datenbanksoftware oder bei der Hardware. Bei Integritätsverletzungen oder nach einem «Absturz» eines Anwendungsprogrammes genügt es, eine oder auch mehrere Transaktionen rückgängig zu machen und anschließend zu wiederholen. Bei schwerwiegenden Fehlern müssen im Extremfall frühere Datenbestände aus Archiven (Archivdatei, „Backup“) geholt und durch eine teilweise wiederholte Transaktionsverarbeitung rekonstruiert werden.

Um Transaktionen rückgängig zu machen, benötigt das Datenbanksystem gewisse Angaben. Normalerweise wird vor der Veränderung eines Objektes eine Kopie (engl. *before image*) desselben in eine sogenannte *Logdatei*[3] (engl. *log file*) geschrieben. Außer den alten Werten des Objektes werden auch Marken in die Logdatei gesetzt, die den Beginn und das Ende einer Transaktion signalisieren. Damit die Logdatei im Fehler-

[3] Die Logdatei ist nicht zu verwechseln mit dem Logbuch aus Abschn. 4.2.2.

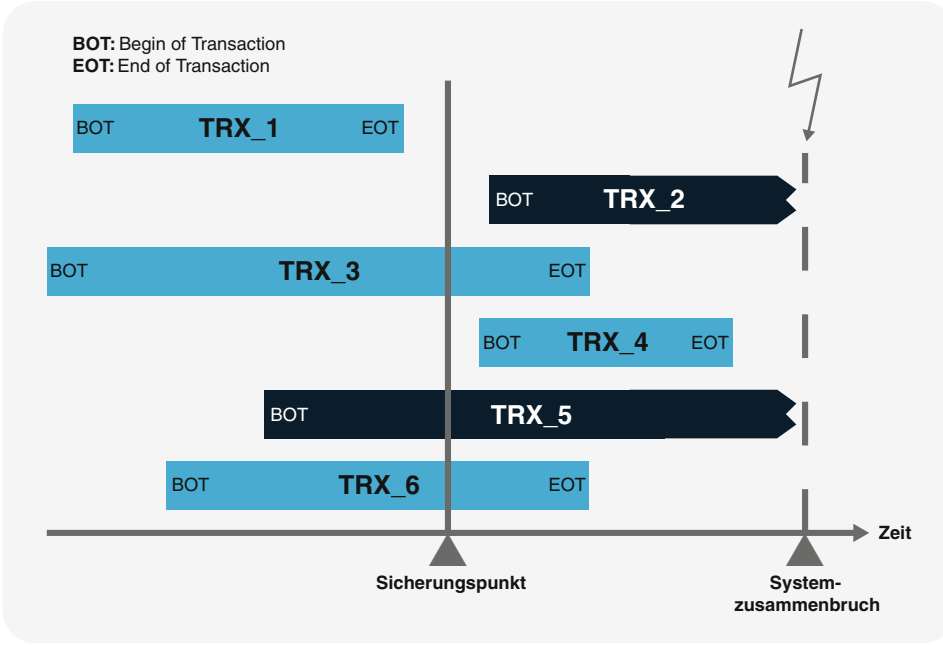

Abb. 4.6 Neustart eines Datenbanksystems nach einem Fehlerfall

fall effizient benutzt werden kann, werden entweder auf Grund von Anweisungen im Anwendungsprogramm oder bei bestimmten Systemereignissen *Sicherungspunkte* (engl. *checkpoints*) gesetzt. Ein systemweiter Sicherungspunkt enthält eine Liste der bis zu diesem Zeitpunkt aktiven Transaktionen. Bei einem *Neustart* (engl. *restart*) muss das Datenbanksystem nur den letzten Sicherungspunkt suchen und die noch nicht abgeschlossenen Transaktionen rückgängig machen (z. B. mit dem SQL-Befehl ROLLBACK).

Ein solcher Vorgang ist in Abb. 4.6 dargestellt: Nach dem Systemzusammenbruch muss die Logdatei rückwärts bis zum jüngsten Sicherungspunkt gelesen werden. Von Interesse sind dabei diejenigen Transaktionen, die noch nicht mit einer sogenannten EOT-Marke (EOT = End Of Transaction) ihre korrekte Beendigung signalisieren konnten, wie die beiden Transaktionen TRX_2 und TRX_5. Für diese muss nun mit Hilfe der Logdatei der alte Datenbankzustand hergestellt werden (engl. *undo*). Dabei muss im Falle der Transaktion TRX_5 vom Sicherungspunkt weg rückwärts bis zur BOT-Marke (BOT = Begin Of Transaction) gelesen werden, um das Before-Image der Transaktion TRX_5 zu erhalten. Unabhängig von der Art des Sicherungspunktes muss auch der neueste Zustand (engl. *after image*) für mindestens TRX_4 hergestellt werden (engl. *redo*).

Zur Rekonstruktion einer Datenbank nach einem Defekt auf einem externen Speicher benötigt man eine Archivkopie der Datenbank und eine Sammlung sämtlicher in der Zwischenzeit erfolgter Änderungen. Archivkopien werden normalerweise vor und nach der Tagesendverarbeitung gezogen, da dies zeitintensiv ist. Tagsüber behilft man sich mit

der Protokollierung der Änderungen auf der Logdatei, wobei pro Objekt jeweils die neusten Zustände festgehalten werden.

Das Sicherstellen von Datenbanken erfordert von den Datenbankspezialisten ein klares Vorgehenskonzept zur *Katastrophenvorsorge*. Normalerweise werden die Sicherheitskopien in Generationen teilweise redundant und physisch getrennt archiviert. Das Bereitstellen solcher Archivkopien und das Löschen alter Bestände muss laufend protokolliert werden. Im Fehlerfall oder bei Katastrophenübungen gilt es, aus alten Archivbeständen und sichergestellten Datenbankänderungen aktuelle Datenbestände innerhalb einer nützlichen Frist zu reproduzieren.

4.3 Konsistenz bei massiv verteilten Daten

4.3.1 BASE und CAP-Theorem

Bei umfangreichen und verteilten Datenhaltungssystemen hat man erkannt, dass die Konsistenzforderung nicht in jedem Fall anzustreben ist, vor allem wenn man auf Verfügbarkeit und Ausfalltoleranz setzen möchte.

Bei relationalen Datenbanksystemen sind die Transaktionen in der höchsten Isolationsstufe immer atomar, konsistenzerhaltend, isoliert und dauerhaft (vgl. ACID, Abschn. 4.2.1). Bei webbasierten Anwendungen hingegen strebt man hohe Verfügbarkeit an und möchte, dass bei einem Ausfall eines Rechnerknotens resp. einer Netzverbindung die Anwendenden ohne Einschränkungen weiterarbeiten können. Solche ausfalltoleranten Systeme verwenden replizierte Rechnerknoten und begnügen sich mit einer weicheren Konsistenzforderung, genannt BASE (Basically Available, Soft State, Eventually Consistent): Es wird erlaubt, dass replizierte Knoten zwischenzeitlich unterschiedliche Datenversionen halten und erst zeitlich verzögert aktualisiert werden.

Eric Brewer der Universität Berkeley stellte an einem Symposium im Jahre 2000 die Vermutung auf, dass die drei Eigenschaften der Konsistenz (engl. *consistency*), der Verfügbarkeit (engl. *availability*) und der Ausfalltoleranz (engl. *partition tolerance*) nicht gleichzeitig in einem massiv verteilten Rechnersystem gelten können:

- **Konsistenz** (Consistency oder abgekürzt C): Wenn eine Transaktion auf einer verteilten Datenbank mit replizierten Knoten Daten verändert, erhalten alle lesenden Transaktionen den aktuellen Zustand, egal über welchen der Knoten sie zugreifen.
- **Verfügbarkeit** (Availability oder A): Unter Verfügbarkeit versteht man einen ununterbrochenen Betrieb der laufenden Anwendung und akzeptable Antwortzeiten.
- **Ausfalltoleranz** (Partition Tolerance oder P): Fällt ein Knoten in einem replizierten Rechnernetzwerk oder eine Verbindung zwischen einzelnen Knoten aus, so hat das keinen Einfluss auf das Gesamtsystem. Zudem lassen sich jederzeit Knoten ohne Unterbruch des Betriebs einfügen oder wegnehmen.

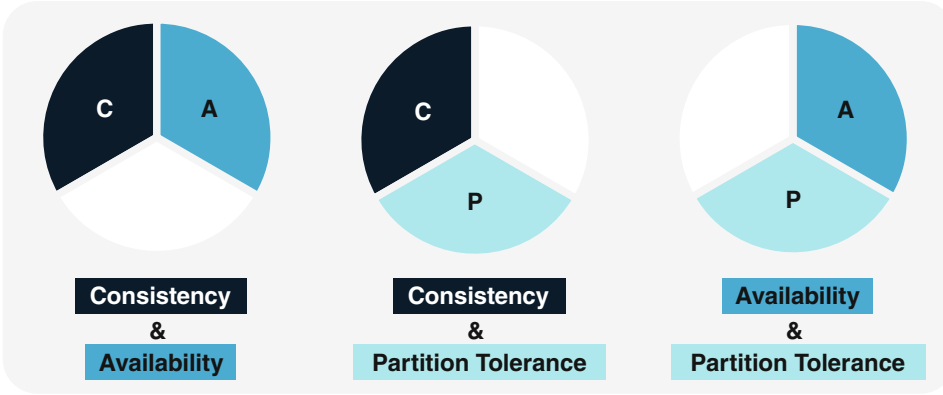

Abb. 4.7 Die möglichen drei Optionen des CAP-Theorems

Später wurde die obige Vermutung von Wissenschaftlern des MIT in Boston bewiesen und als CAP-Theorem etabliert:

CAP-Theorem
Das CAP-Theorem (engl. *CAP theorem*) sagt aus, dass in einem massiv verteilten Datenhaltungssystem jeweils nur zwei Eigenschaften aus den drei der Konsistenz (C), Verfügbarkeit (A) und Ausfalltoleranz (P) garantiert werden können.

Mit anderen Worten, es lassen sich in einem massiv verteilten System entweder Konsistenz mit Verfügbarkeit (CA) oder Konsistenz mit Ausfalltoleranz (CP) oder Verfügbarkeit mit Ausfalltoleranz (AP) kombinieren, aber alle drei sind nicht gleichzeitig zu haben (siehe Abb. 4.7).

Beispiele für die Anwendung des CAP-Theorems sind:

- An einem Börsenplatz wird auf Konsistenz und Verfügbarkeit gesetzt, d. h. CA wird hochgehalten. Dies erfolgt, indem man relationale Datenbanksysteme einsetzt, die dem ACID-Prinzip nachleben.
- Unterhält ein Bankinstitut über das Land verbreitet Geldautomaten, so muss nach wie vor Konsistenz gelten. Daneben ist erwünscht, dass das Netz der Geldautomaten ausfalltolerant ist. Gewisse Verzögerungen in den Antwortzeiten werden hingegen akzeptiert. Ein Netz von Geldautomaten wird demnach so ausgelegt, dass Konsistenz und Ausfalltoleranz gelten. Hier kommen verteilte und replizierte relationale oder NoSQL-Systeme zum Einsatz, die CP unterstützen.
- Der Internetdienst Domain Name System oder DNS muss jederzeit verfügbar und ausfalltolerant sein, da er die Namen von Webseiten zu numerischen IP-Adressen in der TCP/IP-Kommunikation auflösen muss (TCP = Transmission Control Protocol, IP = Internet Protocol). Dieser Dienst setzt auf AP und verlangt den Einsatz von NoSQL-Datenhaltungssystemen, da weltumspannende Verfügbarkeit und Ausfalltoleranz durch ein relationales Datenbanksystem nicht zu haben sind.

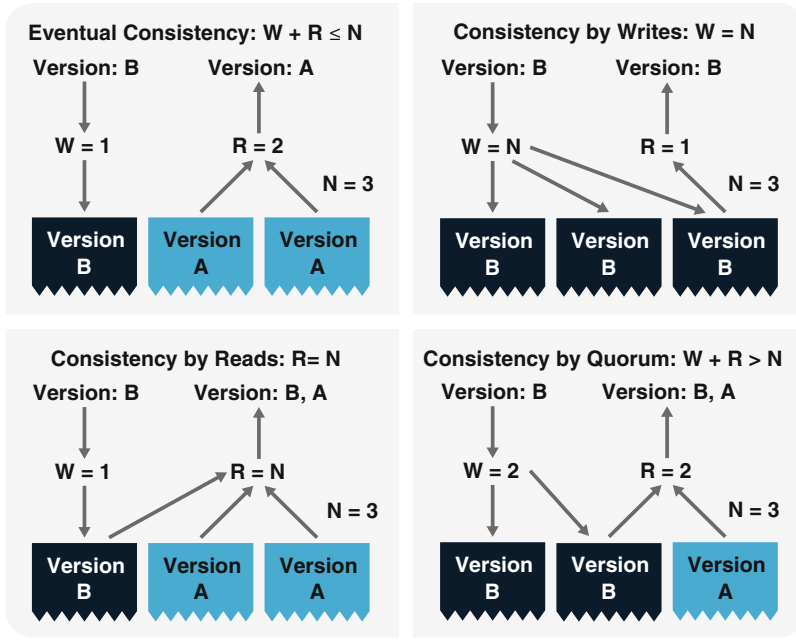

Abb. 4.8 Konsistenzgewährung bei replizierten Systemen

4.3.2 Differenzierte Konsistenzeinstellungen

Im Idealfall gibt es nur einen Ansatz zur Konsistenzgewährung in einem verteilten System: Wenn eine Änderung erfolgt, sehen alle lesenden Transaktionen diese Änderung und sind sicher, dass sie dem aktuellen Zustand entspricht. Offeriert beispielsweise eine Hotelkette ein Reservationssystem auf ihrer Website, dann wird die Buchung eines Hotelzimmers sofort von allen lesenden Buchungstransaktionen erkannt und Doppelbuchungen bleiben ausgeschlossen.

Aufgrund des CAP-Theorems ist bekannt, dass bei einem replizierten Netzwerk von Rechnerknoten nicht alle drei Eigenschaften gleichzeitig zu haben sind. International tätige Hotelketten setzen meistens auf AP, d. h. sie halten die Verfügbarkeit hoch und verlangen Ausfalltoleranz. Damit nehmen sie in Kauf, dass Buchungen nach dem BASE-Prinzip durchgeführt werden. Daneben sind weitere Differenzierungen möglich, die sich anhand der folgenden drei Parameter justieren lassen:

- N = Anzahl replizierter Knoten resp. Anzahl Kopien im Cluster
- R = Anzahl von Kopien, die gelesen werden sollen (successful read)
- W = Anzahl von Kopien, die geschrieben werden müssen (successful write)

Mit der Hilfe der drei Parameter N, R und W lassen sich vier grundlegende Differenzierungsoptionen zur Konsistenzgewährung zusammenstellen. In der Abb. 4.8 sind diese

Varianten zusammengestellt, wobei drei replizierte Knoten betrachtet werden ($N = 3$). Zudem wird angenommen, dass zur Ausgangslage alle drei Knoten die Objektversionen A halten, bevor einzelne Knoten mit der neuen Version B überschrieben werden. Die Frage lautet nun: Wie lassen sich aktuelle Versionen von lesenden Programmen erkennen, falls schreibende Programme Aktualisierungen durchführen?

In einem ersten Fall gilt $W + R \leq N$. Im Beispiel der Abb. 4.8 (oben links) werden die Parameter wie folgt gesetzt: $N = 3$, $W = 1$ und $R = 2$. Die Wahl von $W = 1$ bedeutet, dass mindestens ein Knoten erfolgreich geschrieben werden muss. Die Wahl $R = 2$ verlangt, dass mindestens zwei Knoten erfolgreich gelesen werden sollten. Als Resultat wird im Knoten des schreibenden Programms die alte Version A durch die neue Version B ersetzt. Beim Lesen der beiden Nachbarknoten wird die veraltete Version A zurückgegeben, d. h. der Vorgang veranschaulicht ‚Eventual Consistency'.

Im zweiten Fall ‚Consistency by Writes' muss W der Anzahl der replizierten Knoten entsprechen, d. h. $W = N$ (Abb. 4.8, oben rechts). Erfolgreiche Schreibvorgänge ersetzen die Version A in den drei Knoten durch die nun aktuelle Version B. Falls hier ein Leseprogramm einen Knoten konsultiert, erhält es die aktuelle Version B.

Der dritte Fall wird mit ‚Consistency by Reads' betitelt, da die Anzahl der Knoten mit dem Parameter R übereinstimmt (siehe Abb. 4.8, unten links, Fall $R = N$). Hier wird lediglich ein Knoten mit der neuen Version B geschrieben. Die Konsultation von drei Knoten im Lesevorgang gibt demnach sowohl die aktuelle Version B als auch die veraltete Version A zurück. Beim Erhalt von zwei Versionen, hier A und B, muss noch herausgefunden werden, in welchem zeitlichen Zusammenhang die beiden Versionen A und B zueinander stehen. Gilt A vor B ($A < B$) oder B vor A ($B < A$)? Mit anderen Worten: Welche der beiden Versionen ist die aktuelle? Die Auflösung dieser Kausalitäten wird mit der Hilfe von sogenannten Vektoruhren vorgenommen (siehe Abschn. 4.3.3).

Der vierte und letzte Fall betrifft ‚Consistency by Quorum' und verlangt die Einhaltung der Formel $W + R > N$ (Abb. 4.8, unten rechts). Als Beispiel werden die beiden Parameter W wie R auf Zwei gesetzt, d. h. $W = 2$ und $R = 2$. Demnach müssen je zwei Knoten erfolgreich geschrieben und je zwei erfolgreich gelesen werden. Als Resultat gibt der Lesevorgang wiederum die beiden Versionen B und A zurück und die Kausalität muss mit der Hilfe entsprechender Vektoruhren festgestellt werden.

4.3.3 Vektoruhren zur Serialisierung verteilter Ereignisse

In verteilten Systemen fallen aufgrund konkurrierender Prozesse zu unterschiedlichen Zeiten unterschiedliche Ereignisse an. Um diese Ereignisse partiell zu ordnen, können Vektoruhren (engl. *vector clocks*) verwendet werden. Vektoruhren sind keine eigentlichen Zeituhren, sondern Zählsysteme, die eine Halbordnung unter der zeitlichen Abfolge von Ereignissen ermöglichen.

Im Folgenden betrachten wir konkurrierende Prozesse in einem verteilten System. Eine Vektoruhr ist ein Vektor V mit k Komponenten resp. Zählern Z_i und $i = 1..k$, wobei k der

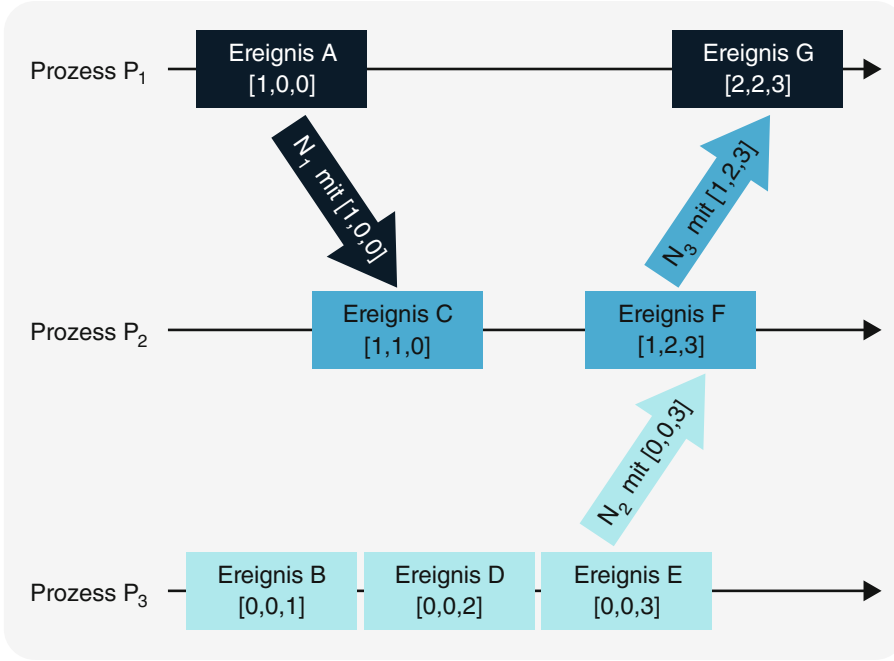

Abb. 4.9 Vektoruhren zeigen kausale Zusammenhänge

Anzahl der Prozesse entspricht. Jeder Prozess P_i besitzt demnach eine Vektoruhr $V_i = [Z_1,\ldots,Z_k]$ mit k Zählern.

Eine Vektoruhr funktioniert wie folgt:

- Initial ist jede Vektoruhr auf Null gestellt, d. h. $V_i = [0,0,\ldots,0]$ für alle Prozesse P_i und Zähler Z_k.
- Bei jedem Nachrichtenaustausch schickt der Sender dem Empfänger seine eigene Vektoruhr mit.
- Falls ein Empfänger eine Nachricht erhält, so erhöht er in seinem Vektor seinen eigenen Zähler Z_i um eins, d. h. $Z_i = Z_i + 1$. Zudem verschmelzt er seinen angepassten Vektor V_i komponentenweise mit dem zugeschickten Vektor W, indem er jeweils den maximalen Wert der korrespondierenden Zählerkomponenten übernimmt, d. h. $V_i[j] = \max(V_i[j],W[j])$ für alle $j = 1..k$.

In Abb. 4.9 ist ein mögliches Szenario von drei konkurrierenden Prozessen P_1, P_2 und P_3 gegeben.

Der Prozess P3 weist die drei Ereignisse B, D und E auf, die zeitlich hintereinander erfolgen. Demnach erhöht er seinen eigenen Zähler Z_3 in seiner Vektoruhr jeweils um eins, womit die drei Vektoruhren [0,0,1] für Ereignis B, [0,0,2] für Ereignis D und [0,0,3] für Ereignis E resultieren.

Beim Prozess P_1 erfolgt zuerst das Ereignis A und die eigene Vektoruhr V_1 wird in der ersten Komponente Z_1 um eins erhöht; die entsprechende Vektoruhr lautet [1,0,0]. Nun schickt der Prozess P_1 eine Nachricht N_1 an den Prozess P_2, wobei die entsprechende Vektoruhr [1,0,0] mitgeliefert wird. Das Ereignis C im Prozess P_2 passt zuerst seine eigene Vektorkomponente V_2 zu [0,1,0] an, bevor die Vektoruhr $V_2 = [0,1,0]$ mit der zugestellten Vektoruhr $V_1 = [1,0,0]$ zu [1,1,0] verschmolzen wird.

Analoge Verschmelzungsoperationen werden bei den Nachrichten N_2 und N_3 vorgenommen: Zuerst werden jeweils die prozessbezogenen Vektoruhren V_2 resp. V_1 in ihren eigenen Komponenten um eins erhöht, bevor die Maximumsbildung der entsprechenden Vektoruhren erfolgt. Es resultieren die beiden Vektoruhren $V_2 = [1,2,3]$, da $[1,2,3] = \max([1,2,0],[0,0,3])$ für Ereignis F resp. $V_1 = [2,2,3]$ für Ereignis G.

Die Kausalität lässt sich für je zwei Ereignisse im verteilten System beantworten: Ein Ereignis X hat vor dem Ereignis Y stattgefunden, falls die Vektoruhr $V(X) = [X_1,X_2,\ldots,X_k]$ von X kleiner oder gleich der Vektoruhr $V(Y) = [Y_1,Y_2,\ldots,Y_k]$ von Y ist. Mit anderen Worten gilt:

Kausalitätsprinzip basierend auf Vektoruhren
Das *Ereignis X erfolgte vor dem Ereignis Y* (oder $X < Y$) genau dann, wenn für alle Komponenten $i = 1,..k$ gilt: $X_i \leq Y_i$ und falls mindestens ein j existiert mit $X_j < Y_j$.

In Abb. 4.9 liegt es auf der Hand, dass das Ereignis B vor dem Ereignis D stattfand, denn die entsprechenden Vektoruhren [0,0,1] und [0,0,2] erfüllen obiges Vergleichskriterium.

Werden die beiden Ereignisse F und G verglichen, so folgert man gemäß den entsprechenden Vektoruhren [1,2,3] und [2,2,3], dass F vor G stattfand: Die erste Komponente der Vektoruhr V(F) ist echt kleiner als die erste Komponente von V(G) und die übrigen Komponenten sind identisch. Demnach signalisiert der Vergleich der Vektoruhren $[1,2,3] \leq [2,2,3]$ die Kausalität $F < G$.

Nun werden zwei fiktive Vektoruhren $V(S) = [3,1,1]$ für das Ereignis S und $V(T) = [1,1,2]$ für das Ereignis T betrachtet: Die beiden Vektoruhren sind nicht vergleichbar, denn es gilt weder $S < T$ noch $T < S$. Die beiden Ereignisse sind nebenläufig und es lässt sich bei solchen Konstellationen keine Kausalität feststellen.

Bei massiv verteilten und replizierten Rechnerstrukturen zeigen Vektoruhren ihre Stärke. Da echte Zeituhren in weltumspannenden Netzen sich schlecht synchronisieren lassen, weicht man auf Vektoruhren aus. Hier kriegen die entsprechenden Vektoruhren so viele Komponenten, wie es Replikate gibt.

Bei der Verteilung von Replikaten kann dank den Vektoruhren festgestellt werden, welche Version die jüngere resp. aktuellere ist. Bei den beiden Fällen ‚Consistency by Reads' resp. ‚Consistency by Quorum' aus Abschn. 4.3.2 resultierten die beiden Versionen B und A beim Lesevorgang. Sind die beiden Versionen mit Vektoruhren bestückt, so kann gemäß obigem Kriterium der Kausalität herausgefunden werden, dass $A < B$, d. h. dass B die aktuelle Version darstellt.

ACID	BASE
Konsistenz hat oberste Priorität (strong consistency)	Konsistenz wird verzögert etabliert (weak consistency)
meistens pessimistische Synchronisationsverfahren mit Sperrprotokollen	meistens optimistische Synchronisationsverfahren mit Differenzierungsoptionen
Verfügbarkeit bei überschaubaren Datenmengen gewährleistet	hohe Verfügbarkeit resp. Ausfalltoleranz bei massiv verteilter Datenhaltung
einige Integritätsregeln sind im Datenbankschema gewährleistet (z. B. referenzielle Integrität)	einige Integritätsregeln sind im Datenbankschema gewährleistet (z. B. referenzielle Integrität)

Abb. 4.10 Vergleich zwischen ACID und BASE

4.4 Vergleich zwischen ACID und BASE

Zwischen den beiden Ansätzen ACID (Atomicity, Consistency, Isolation, Durability) und BASE (Basically Available, Soft State, Eventually Consistent) gibt es gewichtige Unterschiede, die in Abb. 4.10 zusammengefasst sind.

Relationale Datenbanksysteme erfüllen strikt das ACID-Prinzip. Dies bedeutet, dass sowohl im zentralen wie in einem verteilten Fall jederzeit Konsistenz gewährleistet ist. Bei einem verteilten relationalen Datenbanksystem wird ein Koordinationsprogramm benötigt, das bei einer Änderung von Tabelleninhalten diese vollständig durchführt und einen konsistenten Zustand erzeugt. Im Fehlerfall garantiert das Koordinationsprogramm, dass keine Wirkung in der verteilten Datenbank erzielt wird und die Transaktion nochmals gestartet werden kann.

Die Gewährung der Konsistenz wird bei NoSQL-Systemen auf unterschiedliche Art und Weise unterstützt. Im Normalfall wird bei einem massiv verteilten Datenhaltungssystem eine Änderung vorgenommen und den Replikaten mitgeteilt. Allerdings kann es vorkommen, dass einige Knoten bei Benutzeranfragen nicht den aktuellen Zustand zeigen können, da sie zeitlich verzögert Nachführungen mitkriegen. Ein einzelner Knoten im Rechnernetz ist meistens verfügbar (Basically Available) und manchmal noch nicht konsistent nachgeführt (Eventually Consistent), d. h. er kann sich in einem weichen Zustand (Soft State) befinden.

Bei der Wahl der Synchronisationsverfahren verwenden die meisten relationalen Datenbanksysteme pessimistische Ansätze. Dazu müssen für die Operationen einer Transaktion Sperren nach dem Zweiphasensperrprotokoll (vgl. Abschn. 4.2.3) gesetzt und wieder freigegeben werden. Falls die Datenbankanwendungen wenige Änderungen im Vergleich zu den Abfragen durchführen, werden eventuell optimistische Verfahren angewendet (vgl. Abschn. 4.2.4). Im Konfliktfall müssen die entsprechenden Transaktionen nochmals gestartet werden.

Massiv verteilte Datenhaltungssysteme, die verfügbar und ausfalltolerant betrieben werden, können laut dem CAP-Theorem nur verzögert konsistente Zustände garantieren. Zudem wäre das Setzen und Freigeben von Sperren auf replizierten Knoten mit zu grossem Aufwand verbunden. Aus diesem Grund verwenden die meisten NoSQL-Systeme optimistische Synchronisationsverfahren.

Was die Verfügbarkeit betrifft, so können die relationalen Datenbanksysteme abhängig von der Größe des Datenbestandes und der Komplexität der Verteilung mithalten. Bei Big Data Anwendungen allerdings gelangen NoSQL-Systeme zum Einsatz, die hohe Verfügbarkeit neben Ausfalltoleranz oder Konsistenz garantieren.

Jedes relationale Datenbanksystem verlangt die explizite Spezifikation von Tabellen, Attributen, Wertebereichen, Schlüsseln und weiteren Konsistenzbedingungen und legt diese Definitionen im Systemkatalog ab. Zudem müssen die Regeln der referenziellen Integrität im Schema festgelegt werden (vgl. Abschn. 3.7). Abfragen und Änderungen mit SQL sind auf diese Angaben angewiesen und könnten sonst nicht durchgeführt werden. Bei den meisten NoSQL-Systemen liegt kein explizites Datenschema vor, da jederzeit mit Änderungen bei den semi-strukturierten oder unstrukturierten Daten zu rechnen ist.

Einige NoSQL-Systeme erlauben es, die Konsistenzgewährung differenziert einzustellen. Dies führt zu fließenden Übergängen zwischen ACID und BASE, was im Abschn. 4.3.2 veranschaulicht wurde.

4.5 Literatur

Gray und Reuter (1993); Weikum (1988) sowie Weikum und Vossen (2002) erläutern Transaktionskonzepte im Detail. Das ACID-Prinzip geht auf Härder und Reuter (1983) zurück. Das beschriebene Zweiphasen-Sperrprotokoll wurde von den Forschern Eswaran et al. (1976) am IBM Research Lab in San Jose definiert. Bernstein et al. (1987) beschreiben Mehrbenutzeraspekte und Recovery-Maßnahmen. Die Dissertation von Schaarschmidt (2001) stellt Konzepte und Sprachen für die Archivierung von Datenbanken zusammen. Eine weitere Dissertation von Störl (2001) widmet sich dem Backup und dem Recovery in Datenbanksystemen. Reuter (1981) zeigt Verfahren zur Fehlerbehandlung bei Datenbanksystemen. Castano et al. (1994) sowie Basta und Zgola (2011) erklären unterschiedliche Verfahren zur Datenbanksicherung.

Eric Brewer gab am Symposium über Principles of Distributed Computing im Jahr 2000 eine Keynote, die als Geburtsstunde des CAP-Theorems bezeichnet wird (Brewer 2000).

Seth Gilbert und Nancy Lynch vom MIT haben dieses Theorem zwei Jahre später bewiesen (Gilbert und Lynch 2002). Werner Vogels von Amazon.com hat in der Zeitschrift CACM (Communications of the ACM) unterschiedliche Facetten der Konsistenz beschrieben und den Begriff Eventually Consistent geprägt (Vogels 2009). Die differenzierte Konsistenzbetrachtung aus Abschn. 4.3.2 basiert auf dem Key/Value Store Riak und ist dem Fachbuch über NoSQL-Datenbanken von Redmond und Wilson (2012) resp. der Literatur über Riak (2014) entnommen. Der Ansatz für Quorenbildung in verteilten Systemen geht u. a. auf Gifford (1979) zurück. Das Aufdecken von Kausalitäten mittels Vektoruhren geht u.a. auf Schwarz und Mattern (1994) zurück.

Systemarchitektur

5

5.1 Verarbeitung homogener und heterogener Daten

In den Fünfziger- und Sechzigerjahren des letzten Jahrhunderts wurden Dateisysteme auf Sekundärspeichern (Band, Magnettrommel, Magnetplatte) gehalten, bevor ab den Siebzigerjahren Datenbanksysteme auf den Markt kamen. Das Merkmal solcher Dateisysteme war der wahlfreie (engl. *random access*) oder direkte Zugriff (engl. *direct access*) auf das externe Speichermedium. Mit der Hilfe einer Adresse konnte ein bestimmter Datensatz selektiert werden, ohne dass alle Datensätze konsultiert werden mussten. Zur Ermittlung der Zugriffsadresse diente ein Index oder eine Hash-Funktion (vgl. Abschn. 5.2.2).

Die Großrechner mit ihren Dateisystemen wurden vorwiegend für technisch-wissenschaftliche Anwendungen genutzt (Computer = Zahlenkalkulator). Mit dem Aufkommen von Datenbanksystemen eroberten die Rechner die Wirtschaft (Zahlen- & Wortkalkulator). Sie entwickelten sich zum Rückgrad administrativer und kommerzieller Anwendungen, da das Datenbanksystem den Mehrbenutzerbetrieb auf konsistente Art und Weise unterstützte (vgl. ACID, Abschn. 4.2.1). Nach wie vor basieren viele Informationssysteme auf der relationalen Datenbanktechnik, welche die früher eingesetzten hierarchischen oder netzwerkartigen Datenbanksysteme weitgehend ablöste.

Zur Aufbewahrung und Verarbeitung von Daten verwenden relationale Datenbanksysteme ein einziges Konstrukt, die Tabelle. Eine Tabelle ist eine Menge von Datensätzen, die strukturierte Daten flexibel verarbeiten lässt.

Strukturierte Daten (engl. *structured data*) unterliegen einer fest vorgegebenen Datenstruktur, wobei folgende Eigenschaften im Vordergrund stehen:

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_5

- **Schema**: Die Struktur der Daten muss dem Datenbanksystem durch die Spezifikation eines Schemas mitgeteilt werden (vgl. den CREATE-TABLE-Befehl von SQL in Kap. 3). Neben der Spezifikation der Tabellen werden Integritätsbedingungen ebenfalls im Schema abgelegt (vgl. z. B. die Definition der referenziellen Integrität und die Festlegung entsprechender Verarbeitungsregeln, Abschn. 3.7).
- **Datentypen**: Das relationale Datenbankschema garantiert bei der Benutzung der Datenbank, dass die Datenausprägungen jederzeit den vereinbarten Datentypen (z. B. CHARACTER, INTEGER, DATE, TIMESTAMP etc.; vgl. Tutorium für SQL auf der Website www.sql-nosql.org) entsprechen. Dazu konsultiert das Datenbanksystem bei jedem SQL-Aufruf die Systemtabellen (Schemainformation). Insbesondere werden Autorisierungs- und Datenschutzbestimmungen mit der Hilfe des Systemkatalogs geprüft (vgl. das VIEW-Konzept resp. die Vergabe von Privilegien mit den GRANT- und REVOKE-Befehlen, Abschn. 3.8 resp. SQL-Tutorium auf www.sql-nosql.org).

Relationale Datenbanksysteme verarbeiten demnach vorwiegend strukturierte und formatierte Daten. Aufgrund spezifischer Anforderungen aus Büroautomation, Technik oder Webnutzung ist SQL um Datentypen und Funktionen für Buchstabenfolgen (CHARACTER VARYING), Bitfolgen (BIT VARYING, BINARY LARGE OBJECT) oder Textstücke (CHARACTER LARGE OBJECT) erweitert worden (siehe SQL-Tutorium). Zudem wird die Einbindung von XML (eXtensible Markup Language) unterstützt. Diese Erweiterungen führen zur Definition von semi-strukturierten und unstrukturierten Daten.

Semi-strukturierte Daten (engl. *semi-structured data*) sind wie folgt charakterisiert:

- Sie bestehen aus einer Menge von Datenobjekten, deren Struktur und Inhalt laufenden Änderungen unterworfen sind.
- Die Datenobjekte sind entweder atomar oder aus weiteren Datenobjekten zusammengesetzt (komplexe Objekte).
- Die atomaren Datenobjekte enthalten Datenwerte eines vorgegebenen Datentyps.

Datenhaltungssysteme für semi-strukturierte Daten verzichten auf ein fixes Datenbankschema, da Struktur und Inhalt der Daten dauernd ändern. Ein Beispiel dazu wäre ein Content Management System für den Unterhalt einer Website, das Webseiten und Multimedia-Objekte flexibel speichern und verarbeiten kann. Ein solches System verlangt nach erweiterter relationaler Datenbanktechnik (vgl. Kap. 6), XML- oder NoSQL-Datenbanken (vgl. Kap. 7).

Ein *Datenstrom* (engl. *data stream*) ist ein kontinuierlicher Fluss von digitalen Daten, wobei die Datenrate (Datensätze pro Zeiteinheit) variieren kann. Die Daten eines Datenstroms sind zeitlich geordnet und werden oft mit einem Zeitstempel versehen. Neben Audio- und Video-Datenströmen kann es sich um Messreihen handeln, die mit Auswertungssprachen oder spezifischen Algorithmen (Sprachanalyse, Textanalyse, Mustererkennung u. a.) analysiert werden. Im Gegensatz zu strukturierten oder semi-strukturierten Daten lassen sich Datenströme nur sequenziell auswerten.

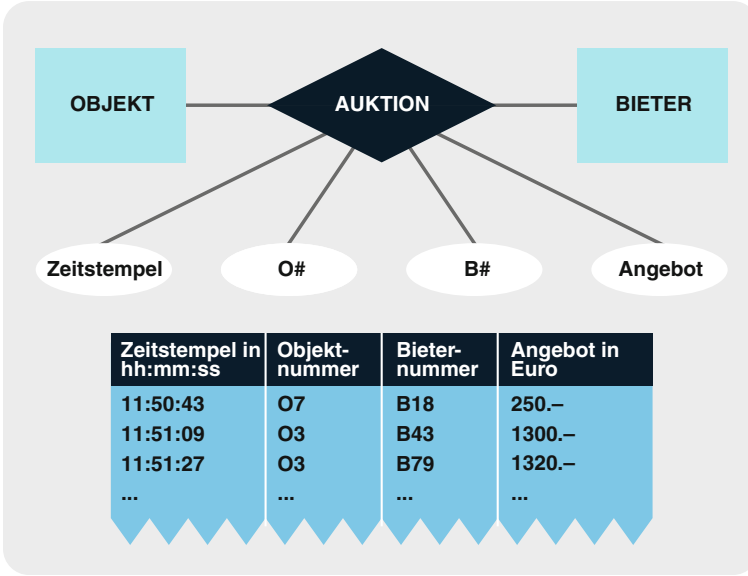

Abb. 5.1 Verarbeitung eines Datenstroms

In Abb. 5.1 ist ein einfaches Anwendungsbeispiel eines Datenstroms aufgezeigt. Auf einer elektronischen Plattform soll eine Englische Auktion für mehrere Gegenstände (engl. *multi-item auction*) durchgeführt werden. Bei der Englischen Auktion beginnt der Prozess des Bietens immer mit einem Mindestpreis. Jeder Teilnehmer kann mehrfach bieten, falls er das aktuelle Angebot übertrifft. Da bei elektronischen Auktionen ein physischer Handelsort entfällt, werden im Vorfeld Zeitpunkt und Dauer der Auktion festgelegt. Der Gewinner einer Englischen Auktion ist derjenige Bieter, der das höchste Angebot im Laufe der Auktion unterbreitet.

Eine AUKTION kann als Beziehungsmenge zwischen den beiden Entitätsmengen OBJEKT und BIETER aufgefasst werden. Die beiden Fremdschlüssel O# und B# werden um einen Zeitstempel und das eigentliche Angebot (z. B. in Euro) pro Bietvorgang ergänzt. Der Datenstrom wird während der Auktion dazu genutzt, den einzelnen Bietenden das aktuelle Angebot aufzuzeigen. Nach Abschluss der Auktion werden die Höchstangebote publiziert und die Gewinner für die einzelnen Gegenstände über ihren Erfolg informiert. Zudem wird der Datenstrom nach Abschluss der Auktion für weitere Auswertungen verwendet, beispielsweise zur Analyse des Bieteverhaltens oder für die Offenlegung bei rechtlichen Anfechtungen.

Unstrukturierte Daten (engl. *unstructured data*) sind digitalisierte Daten, die keiner Struktur unterliegen. Dazu zählen Multimedia-Daten wie Fliesstext, Musikaufnahmen, Satellitenbilder, Audio- oder Videoaufnahmen. Oft werden unstrukturierte Daten über digitale

Sensoren an einen Rechner übermittelt, beispielsweise in Form der oben erwähnten Datenströme, welche strukturierte oder auch unstrukturierte Daten sequenziell übermitteln können.

Die Verarbeitung von unstrukturierten Daten oder von Datenströmen muss mit speziell angepassten Softwarepaketen vorgenommen werden. NoSQL-Datenbanken oder spezifische Data Stream Management Systems werden genutzt, um die Anforderungen an Big Data zu erfüllen.

Im Folgenden werden einige Architekturaspekte für SQL- und NoSQL-Datenbanken behandelt. In Abschn. 5.2 diskutieren wir Speicher- und Zugriffsstrukturen, die sowohl von SQL- wie NoSQL-Datenbanken in unterschiedlicher Form genutzt werden. Drei Ansätze werden beschrieben: Baumstrukturen, Adressberechnungen (Hashing, Consistent Hashing) und mehrdimensionale Datenstrukturen. In Abschn. 5.3 wird aufgezeigt, wie eine mengenorientierte Abfrage in SQL verarbeitet und optimiert werden kann. Abschn. 5.4 widmet sich einem bedeutungsvollen Verfahren zur Parallelisierung unter dem Begriff Map/Reduce. Hier wird aufgezeigt, wie bei massiv verteilten NoSQL-Datenbanken Abfragen effizient durchgeführt werden. Abschn. 5.5 zeigt exemplarisch eine Schichtarchitektur mit klar definierten Software-Ebenen. In Abschn. 5.6 wird argumentiert, warum viele webbasierte Anwendungssysteme sowohl relationale wie nicht-relationale Datenspeicher gleichzeitig nutzen. Literaturangaben werden in Abschn. 5.7 aufgeführt.

5.2 Speicher- und Zugriffsstrukturen

Speicher- und Zugriffsstrukturen für relationale und nicht-relationale Datenbanksysteme müssen darauf ausgelegt sein, Daten in Sekundärspeichern effizient zu verwalten. Sind die Datenbestände umfangreich, so lassen sich Strukturen für Daten im Hauptspeicher nicht ohne Weiteres auf Hintergrundspeicher übertragen. Vielmehr müssen die Speicher- und Zugriffsstrukturen optimiert werden, um ein Schreiben und Lesen von Inhalten *auf externen Speichermedien mit möglichst wenigen Zugriffen* zu bewerkstelligen.

5.2.1 Indexe und Baumstrukturen

Unter einem *Index* (engl. *index*) eines Merkmals verstehen wir eine Zugriffsstruktur, die in einer bestimmten Reihenfolge für jeden Merkmalswert effizient die internen Adressen der Datensätze liefert, die diesen Merkmalswert enthalten. Ein Index entspricht dem Stichwortverzeichnis eines Buches: Auf jedes Stichwort – in alphabetischer Reihenfolge aufgeführt – folgen die Zahlen der Seiten, auf denen es im Text vorkommt.

Für die Tabelle MITARBEITER wünschen wir beispielsweise einen Index über das Merkmal Name. Mit folgendem SQL-Befehl kann ein solcher Index aufgebaut werden, der dem gewöhnlichen Benutzer allerdings verborgen bleibt:

```
CREATE INDEX IX1 ON MITARBEITER(NAME);
```

Zu jedem Namen in der Tabelle MITARBEITER wird in alphabetischer Reihenfolge in der Indexstruktur entweder der Identifikationsschlüssel M# oder die interne Adresse der Mitarbeitertupel festgehalten. Das Datenbanksystem nutzt bei einer entsprechenden Abfrage oder bei einem Verbund diesen Index der Mitarbeiternamen. Das Merkmal Name wird in diesem Fall als *Zugriffsschlüssel* bezeichnet.

Zur Speicherung von Datensätzen oder Zugriffsschlüsseln sowie zur *Indexierung* von Merkmalen können Baumstrukturen verwendet werden, um die Zugriffseffizienz zu steigern. Liegen umfangreiche Datenbestände vor, so ordnet man den Knoten (Wurzel- und Stammknoten) und den Blättern des Baumes nicht einzelne Schlüssel oder Datensätze, sondern ganze *Datenseiten* (engl. *data pages)* zu. Zum Aufsuchen eines bestimmten Datensatzes muss dann der Baum durchsucht werden.

Bei der Hauptspeicherverwaltung verwendet das Datenbanksystem im Hintergrund normalerweise *Binärbäume*, bei denen *der Wurzelknoten sowie jeder Stammknoten zwei Teilbäume aufweist*. Solche Bäume können nicht unkontrolliert für die Speicherung von Zugriffsschlüsseln oder von Datensätzen bei umfangreichen Datenbanken verwendet werden, denn sie wachsen stark in die Tiefe, wenn größere Datenbestände abgespeichert werden müssen. Umfangreiche Bäume sind aber für das Suchen und Lesen von Dateninhalten auf externen Speichermedien unerwünscht, da zu viele Seitenzugriffe notwendig sind.

Die *Höhe eines Baumes* – der Abstand zwischen Wurzelknoten und Blättern – ist ein *Gradmesser für die Anzahl der Zugriffe* auf externe Speichermedien. Um die Zahl der externen Zugriffe möglichst gering zu halten, versucht man bei Datenbanksystemen, die baumartigen Speicherstrukturen nicht so sehr in die Tiefe, sondern eher in die Breite wachsen zu lassen. Ein wichtiger Vertreter solcher Baumstrukturen ist der *Mehrwegbaum* (vgl. Abb. 5.2).

Ein Mehrwegbaum ist ein Baum, dessen *Wurzel- und Stammknoten im Allgemeinen mehr als zwei Teilbäume* aufweisen. Dabei sollten die durch die einzelnen Stammknoten oder Blätter repräsentierten Datenseiten nicht leer bleiben, sondern möglichst mit Schlüsselwerten oder ganzen Datensätzen gefüllt sein. Meistens wird deshalb verlangt, dass die Seiten mindestens zur Hälfte mit Datensätzen oder Schlüsseln besetzt sind (mit Ausnahme der zum Wurzelknoten gehörenden Seite).

***Mehrwegbaum* (engl. *B-tree*)**
Ein Baum ist ein *Mehrwegbaum oder B-Baum der Ordnung n*, falls

- er vollständig ausbalanciert ist (jeder Weg von der Wurzel zu einem beliebigen Blatt hat eine feste gleiche Länge), und

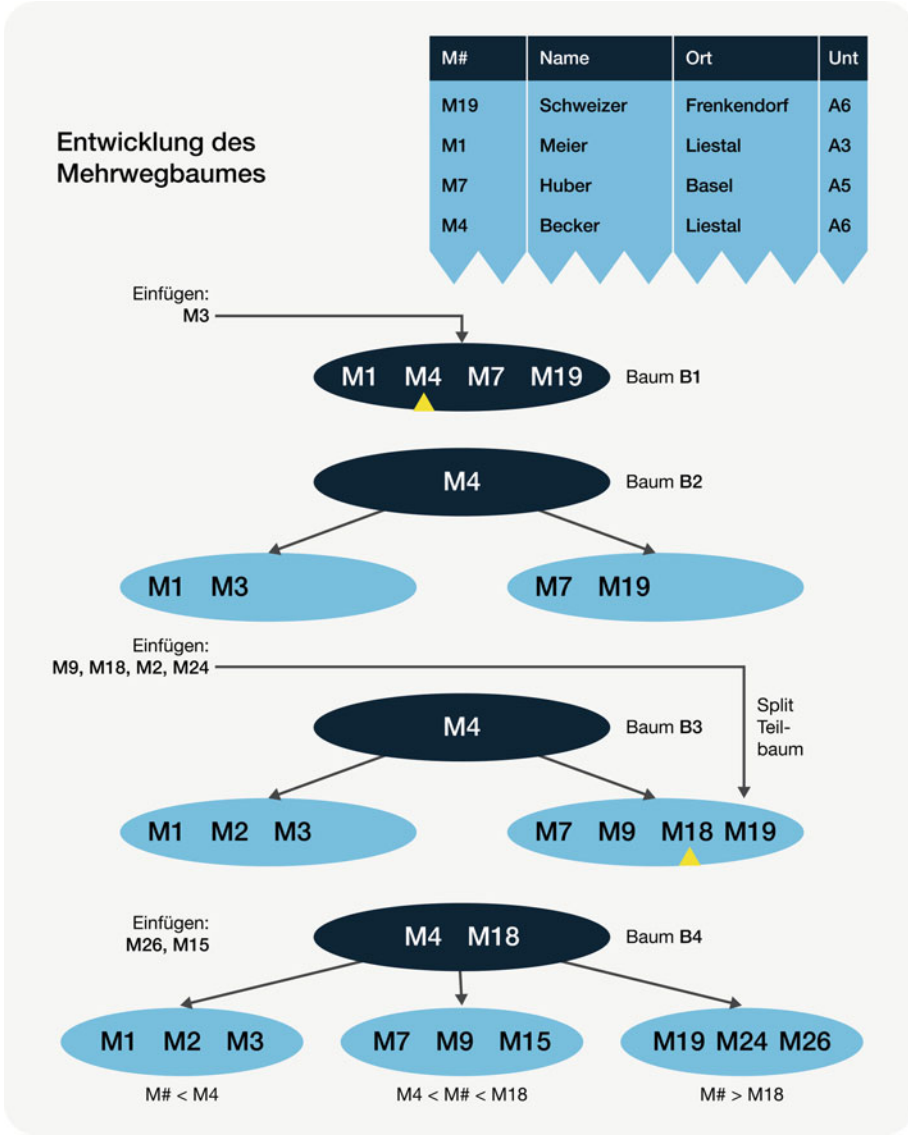

Abb. 5.2 Mehrwegbaum in dynamischer Veränderung

- jeder Knoten (außer dem Wurzelknoten) und jedes Blatt des Baumes mindestens n und höchstens 2*n Einträge in der entsprechenden Datenseite hat.

Die zweite Bedingung eines Mehrwegbaumes lässt sich auch anders interpretieren: Da jeder Knoten außer der Wurzel mindestens n Einträge aufweist, besitzt jeder Knoten

mindestens n Teilbäume. Umgekehrt enthält jeder Knoten höchstens 2*n Einträge, d. h., jeder Knoten eines Mehrwegbaumes kann höchstens auf 2*n Teilbäume verweisen.

Betrachten wir dazu unsere Tabelle MITARBEITER mit dem zugehörigen Schlüssel Mitarbeiternummer. Wollen wir den Schlüssel M# in einem Mehrwegbaum der Ordnung $n = 2$ als Zugriffsstruktur ablegen, so erhalten wir das in Abb. 5.2 gezeigte Bild.

Knoten und Blätter des Baumes können somit nicht mehr als vier Einträge enthalten. Neben den eigentlichen Schlüsseln nehmen wir stillschweigend an, dass die zu den Knoten und Blättern zählenden Seiten nicht nur Schlüsselwerte, sondern auch Zeiger auf Datenseiten aufweisen, die die eigentlichen Datensätze enthalten. Der Baum in Abb. 5.2 repräsentiert somit einen Zugriffsbaum und nicht die Datenverwaltung für die in der Tabelle MITARBEITER gezeigten Datensätze.

Der Wurzelknoten des Mehrwegbaumes B1 enthält in unserem Beispiel die vier Schlüssel M1, M4, M7 und M19 in Sortierreihenfolge. Beim Einfügen eines zusätzlichen Schlüssels M3 muss der Wurzelknoten geteilt werden, da er keinen Eintrag mehr erlaubt. Die Teilung geschieht so, dass ein ausbalancierter Baum entsteht. Der Schlüssel M4 wird zum Wurzelknoten erklärt, da er die restliche Schlüsselmenge in zwei gleichgroße Hälften zerlegt. Der linke Teilbaum entspricht Schlüsselwerten mit der Bedingung «M# kleiner als M4» (d. h. in unserem Fall M1 und M3), der rechte entspricht «M# größer als M4» (d. h. M7 und M19). Auf analoge Art werden weitere Schlüssel eingefügt, unter Beibehaltung einer festen Baumhöhe.

Beim Suchen eines bestimmten Schlüssels geht das Datenbanksystem wie folgt vor: Wird der Schlüsselkandidat M15 im Mehrwegbaum B4 der Abb. 5.2 nachgefragt, so vergleicht es ihn mit den Einträgen des Wurzelknotens. M15 liegt zwischen den Schlüsseln M4 und M18, also wählt es den entsprechenden Teilbaum (hier ein Blatt) aus und setzt seine Suche fort. Schließlich findet es den Eintrag im Blatt. Der Aufwand für die Suche des Schlüssels M15 beträgt in diesem vereinfachten Beispiel lediglich zwei Seitenzugriffe, einen für den Wurzelknoten und einen für das Blatt.

Die Höhe des Mehrwegbaumes bestimmt die Zugriffszeit der Schlüssel und entsprechend auch der zu einem (Such-)Schlüssel gehörenden Daten. Eine Verbesserung der Zugriffszeiten wird erreicht, indem man beim Mehrwegbaum den Verzweigungsgrad weiter erhöht.

Eine andere Möglichkeit besteht beim sogenannten *blattorientierten Mehrwegbaum* (bekannt unter dem Namen B*-Baum). Bei diesem werden die eigentlichen Datensätze nie in inneren Knoten, sondern immer in den Blättern des Baumes gespeichert. Die Knoten weisen allesamt nur Schlüsseleinträge auf, um den Baum möglichst niedrig zu halten.

5.2.2 Hash-Verfahren

Schlüsseltransformations- oder Adressberechnungsverfahren (engl. *key hashing* oder einfach *hashing*) bilden die Grundlage von gestreuten Speicher- und Zugriffsstrukturen. Eine

Schlüsseltransformation (engl. *hash function*) ist eine Abbildung einer Menge von Schlüsseln in eine Menge von Adressen, die einen zusammenhängenden Adressraum bilden.

Eine einfache Schlüsseltransformation ordnet jedem Schlüssel eines Datensatzes eine natürliche Zahl von 1 bis n als Adresse zu. Diese Adresse wird als relative Seitennummer interpretiert, wobei die Seite eine fixe Anzahl von Schlüsselwerten aufnimmt, inklusive oder exklusive dazugehörende Datensätze.

An Schlüsseltransformationen werden die folgenden Anforderungen gestellt:

- Die Transformationsvorschrift muss mit einfacher Berechnung und ohne große Kosten eingehalten werden können.
- Die belegten Adressen müssen gleichmäßig über den Adressraum verteilt sein.
- Die Wahrscheinlichkeit für Mehrfachbelegungen, d. h. die Verwendung gleicher Adressen für mehrere Schlüssel, sollte für alle Schlüsselwerte gleich groß sein.

Es besteht eine beachtliche Anzahl von Hash-Funktionen, die alle ihre Vor- und Nachteile haben. Als bekanntestes und einfachstes Verfahren gilt die Restklassenbildung, das Hashing mit Divisionsrest.

Hashing mit Divisionsrest
Jeder Schlüssel wird als natürliche Zahl interpretiert, indem die Bitdarstellung verwendet wird. Die Schlüsseltransformation oder *Hash-Funktion H für einen Schlüssel k und eine Primzahl p* ist durch die Formel

$$H(k) := k \bmod p$$

gegeben. Der ganzzahlige Rest «k mod p» – der Division des Schlüsselwertes k durch die Primzahl p – bildet eine relative Adresse oder Seitennummer. Bei diesem Divisionsrestverfahren bestimmt die Wahl der Primzahl p die Speicherausnutzung und den Grad der Gleichverteilung.

In Abb. 5.3 zeigen wir die Tabelle MITARBEITER, die wir mit der obigen Restklassenbildung auf verschiedenen Seiten abbilden.

Dabei nehmen wir für unser Beispiel an, dass jede Seite vier Schlüsselwerte aufnehmen kann. Als Primzahl wählen wir die Ziffer 5. Jeder Schlüsselwert wird nun durch 5 dividiert, wobei der ganzzahlige Rest die Seitennummer bestimmt.

Beim Einfügen des Schlüssels M14 kommt es zu einer Kollision, da die entsprechende Seite bereits gefüllt ist. Der Schlüssel M14 wird in einen *Überlaufbereich* gestellt. Ein Verweis von der Seite 4 auf den Überlaufbereich garantiert die Zugehörigkeit des Schlüssels M14 zur Restklasse 4.

Es existieren unterschiedliche Verfahren zur Behandlung von Überläufern. Anstelle eines Überlaufbereichs können für die Überläufer selbst wieder Schlüsseltransformationen angewendet werden. Bei stark anwachsenden Schlüsselbereichen oder bei größeren

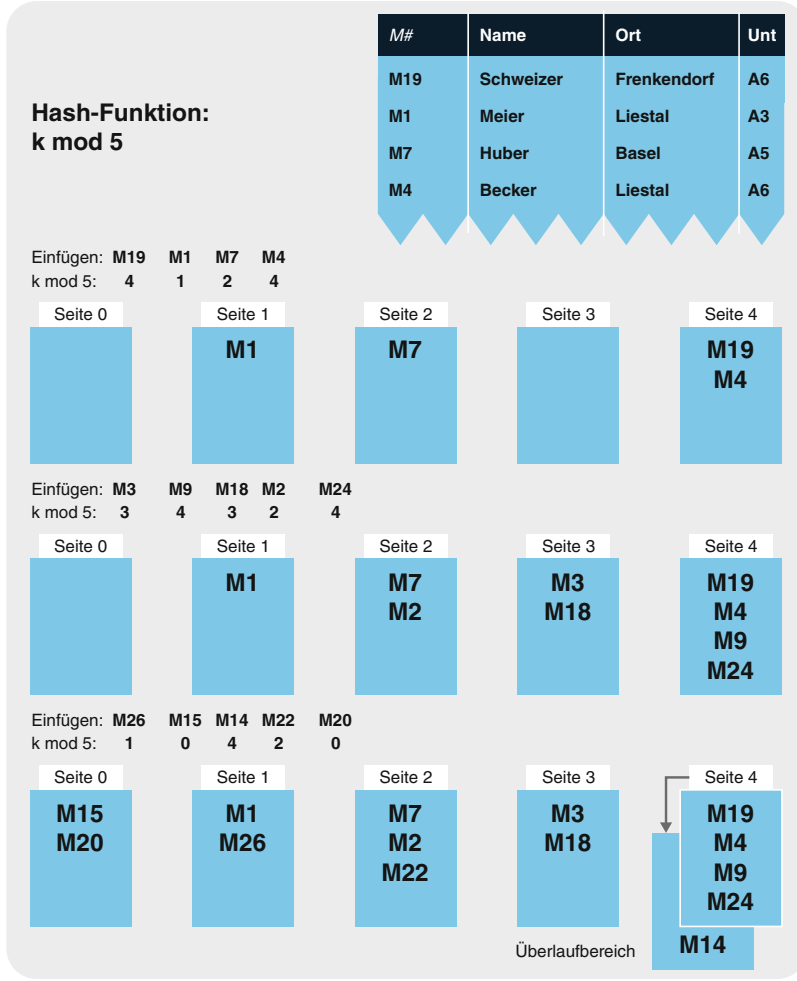

Abb. 5.3 Schlüsseltransformation mit Restklassenbildung

Löschoperationen treten bei der Überlaufbehandlung oft Schwierigkeiten auf. Um diese Probleme zu entschärfen, sind dynamische Verfahren zur Schlüsseltransformation entwickelt worden.

Bei *dynamischen Hash-Verfahren* wird versucht, die Belegung des Speicherplatzes unabhängig vom Wachstum der Schlüssel zu halten. Überlaufbereiche oder umfassende Neuverteilungen von Adressen werden weitgehend vermieden. Bei dynamischen Hash-Verfahren kann ein bestehender Adressraum entweder durch eine geschickte Wahl der Schlüsseltransformation oder durch Verwendung einer hauptspeicherresidenten Seitenzuordnungstabelle erweitert werden, ohne dass alle bereits gespeicherten Schlüssel oder Datensätze neu geladen werden müssen.

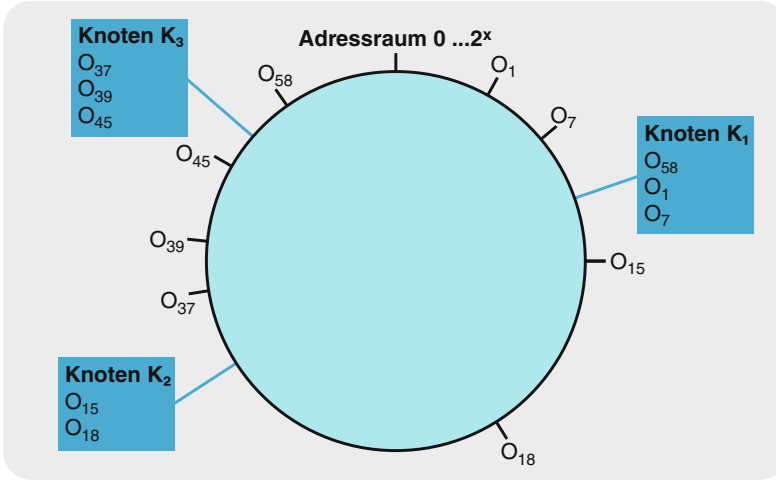

Abb. 5.4 Ring mit Zuordnung von Objekten zu Knoten

5.2.3 Consistent Hashing

Konsistente Hash-Funktionen (engl. *consistent hashing*) zählen zur Familie gestreuter Adressberechnungen (vgl. Hash-Verfahren im vorigen Abschnitt). Aus einer Menge von Schlüsseln wird eine Speicheradresse oder ein Hash-Wert berechnet, um den entsprechenden Datensatz abzulegen.

Im Fall von Big Data Anwendungen werden die Schlüssel-Wert-Paare unterschiedlichen Knoten im Rechnernetz zugeordnet. Aufgrund des Schlüssels (z. B. Begriff oder Tag) werden deren Werte (z. B. Häufigkeiten) im entsprechenden Knoten abgelegt. Wichtig dabei ist folgende Tatsache: Beim Consistent Hashing wird die Adressberechnung sowohl für die Knotenadressen als auch für die Speicheradressen der Objekte (Key/Value) verwendet.

In Abb. 5.4 wird das Consistent Hashing schematisch dargestellt. Der Adressraum von 0 bis 2^x Schlüsselwerten wird als Ring zusammengefasst, danach wird eine Hash-Funktion gewählt, die folgende Berechnungen vornimmt:

- **Adressberechnung der Knoten**: Die Netzwerkadressen der Knoten werden mit der Hilfe der gewählten Hash-Funktion auf Speicheradressen abgebildet und im Ring eingetragen.
- **Adressberechnung der Objekte**: Die Schlüssel der Key/Value-Paare werden mit dem Hashing-Algorithmus zu Adressen transformiert und die Objekte werden auf dem Ring eingetragen.

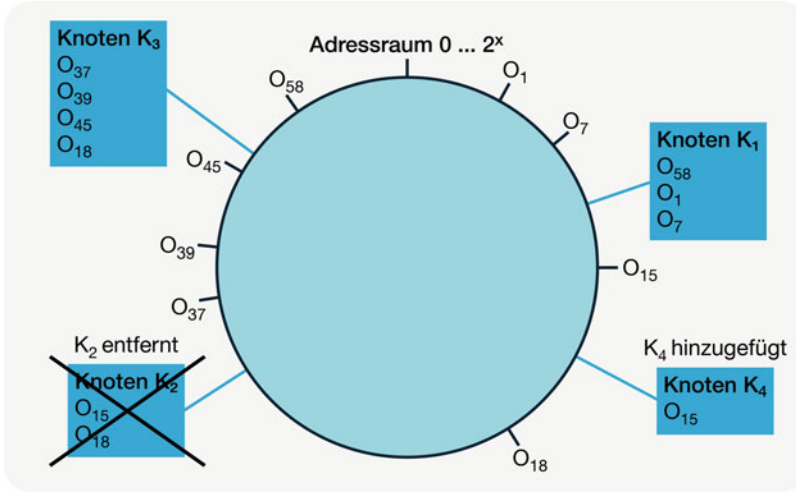

Abb. 5.5 Dynamische Veränderung des Rechnernetzes

Die Speicherung der Schlüssel-Wert-Paare zu den entsprechenden Speicherknoten erfolgt nach einer einfachen Zuordnungsregel: Die Objekte werden im Uhrzeigersinn dem nächsten verfügbaren Knoten zugeordnet und dort verwaltet.

Die Abb. 5.4 zeigt einen Adressraum mit drei Knoten und acht Objekten (Key/Value-Paaren). Die Lokalität der Knoten und der Objekte ergibt sich aufgrund der berechneten Adressen. Nach der Zuordnungsregel werden die Objekte O_{58}, O_1 und O_7 im Knoten K_1 gespeichert, entsprechend fallen die beiden Objekte O_{15} und O_{18} in den Knoten K_2 und die restlichen drei Objekte in den Knoten K_3.

Die Stärke von Consistent Hashing zeigt sich bei flexiblen Rechnerstrukturen, bei welchen jederzeit Knoten hinzugefügt resp. Knoten entfernt werden. Solche Änderungen haben nur Auswirkungen auf die Objekte in unmittelbarer Nähe zu den veränderten Knoten im Ring. Dadurch wird vermieden, dass bei Umstellungen im Rechnernetz die Adressen vieler Key/Value-Paare neu berechnet und zugeordnet werden müssen.

In Abb. 5.5 werden zwei Umstellungen illustriert: Hier wird der Knoten K_2 entfernt und ein neuer Knoten K_4 hinzugefügt. Nach den lokalen Anpassungen liegt das ursprünglich im Knoten K_2 gespeicherte Objekt O_{18} nun im Knoten K_3. Das restliche Objekt O_{15} wird gemäss der Zuordnungsregel in den neu eingefügten Knoten K_4 überwiesen.

Consistent Hashing kann für replizierte Rechnernetzwerke verwendet werden. Dazu werden die gewünschten Kopien der Objekte mit einer Versionennummer versehen und auf dem Ring eingetragen. Damit wird die Ausfallsicherheit wie die Verfügbarkeit des Gesamtsystems erhöht.

Eine weitere Option besteht darin, virtuelle Knoten einzuführen, um die Objekte gleichmäßiger auf die Knoten verteilen zu können. Auch hier erhalten die Netzadressen der Knoten Versionennummern, damit sie auf dem Ring abgebildet werden können.

Konsistente Hash-Funktionen werden für viele NoSQL-Systeme verwendet, vor allem bei der Implementierung von Key/Value-Speichersystemen.

5.2.4 Mehrdimensionale Datenstrukturen

Mehrdimensionale Datenstrukturen unterstützen den Zugriff auf Datensätze mit mehreren Zugriffsschlüsselwerten. Die Gesamtheit dieser Zugriffsschlüssel wird *mehrdimensionaler Schlüssel* genannt. Ein solcher ist immer eindeutig, braucht aber nicht in jedem Fall minimal zu sein.

Unter einer *mehrdimensionalen Datenstruktur* (engl. *multi-dimensional data structure*) versteht man nun eine Datenstruktur, die einen mehrdimensionalen Schlüssel unterstützt. Beispielsweise lässt sich eine Tabelle MITARBEITER mit den beiden Schlüsselteilen Mitarbeiternummer und Jahrgang als zweidimensionale Datenstruktur auffassen. Die Mitarbeiternummer bildet einen Teil des zweidimensionalen Schlüssels und bleibt nach wie vor eindeutig. Das Merkmal Jahr bildet den zweiten Teil und dient als zusätzlicher Zugriffsschlüssel, wobei diese Angabe nicht eindeutig zu sein braucht.

Bei den mehrdimensionalen Datenstrukturen strebt man – im Gegensatz zu den Baumstrukturen – an, dass kein Schlüsselteil bei der Speicherung der physischen Datensätze die Reihenfolge bestimmt. Eine mehrdimensionale Datenstruktur ist *symmetrisch*, wenn sie den Zugriff über mehrere Zugriffsschlüssel ermöglicht, ohne einen bestimmten Schlüssel oder eine Schlüsselkombination zu bevorzugen. Für unsere Beispieltabelle MITARBEITER ist es wünschenswert, dass beide Schlüsselteile, Mitarbeiternummer und Jahrgang, gleichberechtigt sind und bei einer konkreten Abfrage den Zugriff effizient unterstützen.

Als bedeutende mehrdimensionale Datenstruktur ist die sogenannte Gitterdatei, bekannt unter dem englischen Namen *Grid File*, zu erwähnen:

Gitterdatei
Eine Gitterdatei ist eine mehrdimensionale Datenstruktur mit folgenden Eigenschaften:

- Sie unterstützt den Zugriff bezüglich eines *mehrdimensionalen Zugriffsschlüssels* auf symmetrische Art, d. h. keine Dimension des Schlüssels ist dominant.
- Sie erlaubt das Lesen eines beliebigen Datensatzes mit *zwei Seitenzugriffen*, der erste auf das Gitterverzeichnis und der zweite auf die Datenseite selbst.

Eine Gitterdatei besteht aus einem Gitterverzeichnis und einer Datei mit den Datenseiten. Das Gitterverzeichnis stellt einen mehrdimensionalen Raum dar, wobei jede Dimension einem Teil des mehrdimensionalen Zugriffsschlüssels entspricht. Beim Einfügen von Datensätzen wird das Verzeichnis alternierend in den Dimensionen in Zellen unterteilt. Im Beispiel in Abb. 5.6 wechseln wir für den zweidimensionalen Zugriffsschlüssel alternierend nach Mitarbeiternummer und Jahrgang. Die entsprechenden Unterteilungsgrenzen werden als Skalen des Gitterverzeichnisses bezeichnet.

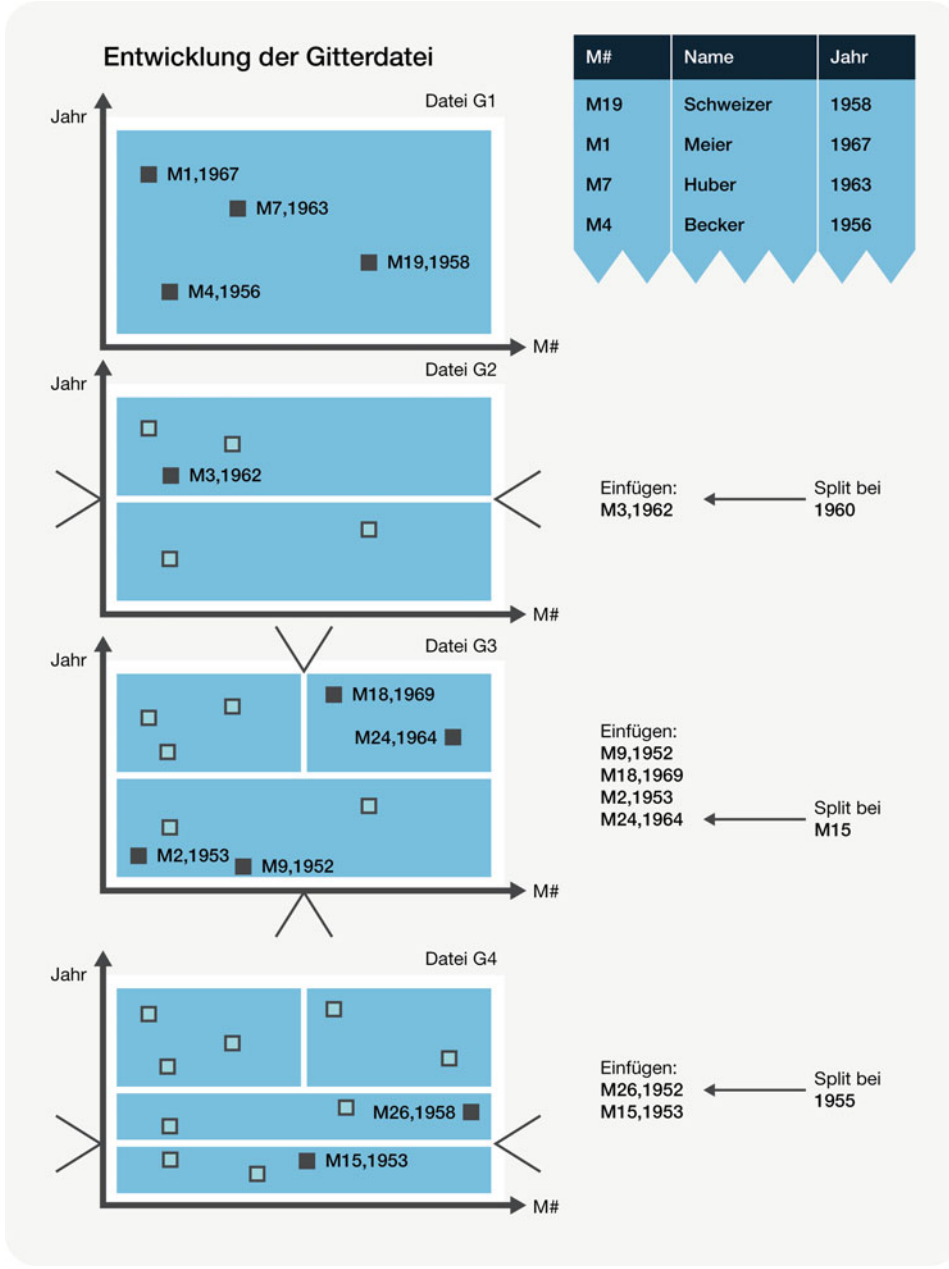

Abb. 5.6 Dynamisches Unterteilen des Gitterverzeichnisses

Eine *Zelle des Gitterverzeichnisses* entspricht einer Datenseite und enthält mindestens n Einträge und maximal 2*n Einträge. Leere Zellen im Gitterverzeichnis müssen zu größeren Zellen zusammengefasst werden, damit die zugehörigen Datenseiten die Minimalzahl der Einträge aufnehmen können. In unserem Beispiel nehmen wir wiederum an, dass höchstens vier Einträge in den Datenseiten vorkommen können (n = 2).

Da das Gitterverzeichnis im Allgemeinen groß ist, muss es wie die Datensätze im Sekundärspeicher gehalten werden. Die Menge der Skalen hingegen ist klein und kann resident im Hauptspeicher liegen. Somit geschieht ein Zugriff auf einen spezifischen Datensatz wie folgt: Mit den k Schlüsselwerten einer k-dimensionalen Gitterdatei durchsucht das System die Skalen und stellt fest, in welchem Intervall der jeweilige Teil des Suchschlüssels liegt. Die so bestimmten Intervalle erlauben einen direkten Zugriff auf den entsprechenden Abschnitt des Gitterverzeichnisses. Jede Zelle des Verzeichnisses enthält die Nummer der Datenseite, in der die zugehörigen Datensätze abgespeichert sind. Mit einem weiteren Zugriff auf die Datenseite der Zelle kann schließlich festgestellt werden, ob sie den gesuchten Datensatz enthält oder nicht.

Beim Suchen eines beliebigen Datensatzes in einer Gitterdatei ist das *Zwei-Seiten-Zugriffsprinzip* immer gewährleistet, das heißt, dass höchstens zwei Seitenzugriffe auf den Sekundärspeicher notwendig sind; der erste führt in den richtigen Abschnitt des Gitterverzeichnisses, der zweite zur richtigen Datenseite. Beispielsweise wird der Mitarbeitende mit der Nummer M18 und dem Jahrgang 1969 in der Gitterdatei G4 der Abb. 5.6 wie folgt gesucht: Die Mitarbeiternummer M18 liegt im Skalenbereich M15 bis M30, d. h. in der rechten Hälfte der Gitterdatei. Der Jahrgang 1969 liegt zwischen den Skalen 1960 und 1970 und somit in der oberen Hälfte. Das Datenbanksystem findet also anhand der Skalen mit einem ersten Zugriff die Adresse der Datenseite im Gitterverzeichnis. Ein zweiter Zugriff auf die entsprechende Datenseite führt zu den gesuchten Datensätzen mit den Zugriffsschlüsseln (M18,1969) und (M24,1964).

Eine k-dimensionale Gitterdatei unterstützt die Anfrage nach einem einzelnen Datensatz oder nach einem Bereich von Datensätzen: Durch eine *Punktfrage* (engl. *point query*) kann anhand von k Zugriffsschlüsseln der entsprechende Datensatz gesucht werden. Es ist auch möglich, mit einer Teilpunktfrage nur einen Teil des Schlüssels zu spezifizieren. Mittels einer *Bereichsfrage* (engl. *range query*) kann für jeden der k Schlüsselteile ein Bereich untersucht werden. Sämtliche Datensätze werden bereitgestellt, für die ihre Schlüsselteile in den jeweiligen Bereichen liegen. Es kann auch hier nur für einen Teil der Schlüssel ein Bereich angegeben und ausgewertet werden (sogenannte Teilbereichsfrage).

Eine Punktfrage entspricht beispielsweise dem bereits dargelegten Aufsuchen des Datensatzes (M18,1969). Wissen wir lediglich den Jahrgang des Mitarbeitenden, so spezifizieren wir zur Suche das Jahr 1969 als Teilpunktfrage. Wir können auch (Teil-) Bereichsfragen stellen, indem wir sämtliche Mitarbeitende mit Jahrgang 1960 bis 1969 abfragen. Auf unser Beispiel der Abb. 5.6 bezogen, liegen wir damit in der oberen Hälfte des Gitterverzeichnisses G4 und müssen deshalb nur die beiden entsprechenden Datenseiten durchsuchen. Bei einer mehrdimensionalen Gitterdatei können auf diese Weise

Bereichs- und Teilbereichsfragen beantwortet werden, ohne dass die gesamte Datei durchkämmt werden muss.

In den letzten Jahren sind verschiedene mehrdimensionale Datenstrukturen untersucht und beschrieben worden, die auf vorteilhafte Weise mehrere Zugriffsmerkmale symmetrisch unterstützen. Das Angebot mehrdimensionaler Datenstrukturen für SQL- oder NoSQL-Datenbanken ist noch bescheiden, doch verlangen webbasierte Suchvorgänge vermehrt nach solchen Speicherstrukturen. Insbesondere müssen geografische Informationssysteme sowohl topologische wie geometrische Anfragen effizient unterstützen können.

5.3 Übersetzung und Optimierung relationaler Abfragen

5.3.1 Erstellen eines Anfragebaums

Die Benutzerschnittstelle eines relationalen Datenbanksystems ist mengenorientiert, da den Benutzern ganze Tabellen oder Sichten zur Verfügung gestellt werden. Beim Einsatz einer relationalen Abfrage- und Manipulationssprache muss vom Datenbanksystem deshalb eine Übersetzung und Optimierung der entsprechenden Anweisungen vorgenommen werden. Dabei ist wesentlich, dass sowohl das Berechnen als auch das Optimieren des sogenannten Anfragebaumes ohne Benutzerintervention erfolgt.

Anfragebaum
Ein Anfragebaum (engl. *query tree)* visualisiert grafisch eine relationale Abfrage durch den *äquivalenten Ausdruck der Relationenalgebra*. Die Blätter des Anfragebaumes entsprechen den für die Abfrage verwendeten Tabellen. Der Wurzel- und die Stammknoten bezeichnen die algebraischen Operatoren.

Als Beispiel zur Illustration eines Anfragebaumes verwenden wir SQL und die bereits bekannten Tabellen MITARBEITER und ABTEILUNG (siehe Abb. 5.7). Wir interessieren uns für eine Liste derjenigen Ortschaften, in denen die Mitarbeitenden der Abteilung Informatik ihren Wohnsitz haben:

```
SELECT  Ort
FROM    MITARBEITER, ABTEILUNG
WHERE   Unt=A# AND Bezeichnung=‚Informatik‘
```

Algebraisch können wir dieselbe Abfrage durch eine Sequenz von Operatoren ausdrücken:

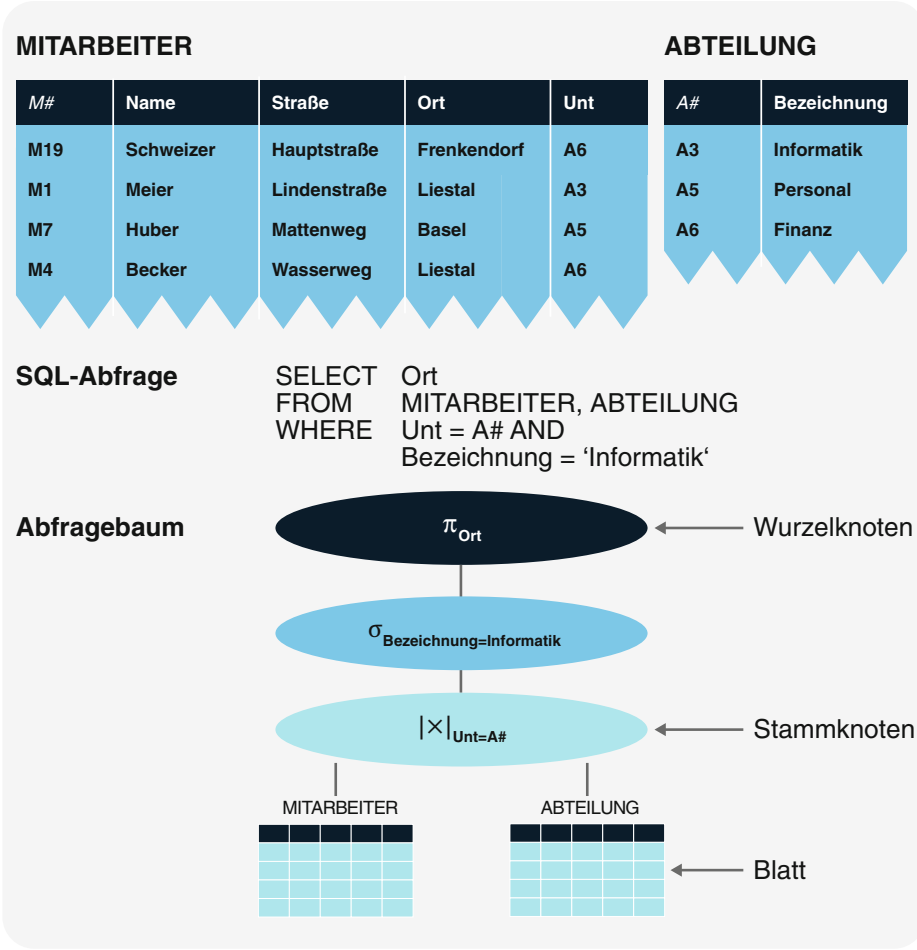

Abb. 5.7 Anfragebaum einer qualifizierten Abfrage über zwei Tabellen

$$\text{TABELLE} := \pi_{\text{Ort}} \left(\sigma_{\text{Bezeichnung=Informatik}} \left(\text{MITARBEITER} \left| \text{X} \right|_{\text{Unt = A\#}} \text{ABTEILUNG} \right) \right)$$

Der Ausdruck berechnet zuerst einen Verbund der Tabelle MITARBEITER mit der Tabelle ABTEILUNG über die gemeinsame Abteilungsnummer. Anschließend werden im Zwischenresultat diejenigen Mitarbeitenden selektiert, die in der Abteilung mit der Bezeichnung Informatik arbeiten. Zuletzt werden durch eine Projektion die gesuchten Ortschaften zusammengestellt. Abb. 5.7 zeigt den Ausdruck dieser algebraischen Operatoren in Form des zugehörigen Anfragebaumes.

Der Anfragebaum in Abb. 5.7 kann wie folgt interpretiert werden: Die Blattknoten bilden die beiden Tabellen MITARBEITER und ABTEILUNG der entsprechenden Abfrage. Diese werden in einem ersten Stammknoten (Verbundoperator) zusammengehalten und anschließend durch einen weiteren Stammknoten (Selektionsoperator) auf diejenigen Einträge reduziert, die die Bezeichnung Informatik tragen. Der Wurzelknoten entspricht der Projektion, die die Resultattabelle mit den gesuchten Ortschaften erzeugt.

Wurzel- und Stammknoten eines Anfragebaumes verweisen entweder auf einen oder auf zwei Teilbäume. Je nachdem, ob die Operatoren, welche die Knoten bilden auf eine oder auf zwei Zwischentabellen (Teilbäume) einwirken, spricht man von unären oder binären Operatoren. *Unäre Operatoren*, die nur auf eine Tabelle wirken, sind der Projektions- und der Selektionsoperator (vgl. die früheren Abb. 3.2 und 3.3). *Binäre Operatoren* mit zwei Tabellen als Operanden sind die Vereinigung, der Durchschnitt, die Subtraktion, das kartesische Produkt, der Verbund und die Division.

Der Aufbau eines Anfragebaumes ist der erste Schritt bei der Übersetzung und Ausführung einer relationalen Datenbankabfrage. Die vom Benutzer angegebenen Tabellen- und Merkmalsnamen müssen in den Systemtabellen auffindbar sein, bevor weitere Verarbeitungsschritte stattfinden. Der Anfragebaum dient also der Überprüfung der Syntax der Abfrage sowie der Zugriffsberechtigung des Benutzers. Weitere Sicherheitsanforderungen können erst zur Laufzeit überprüft werden, wie beispielsweise wertabhängiger Datenschutz.

Nach dieser Zugriffs- und Integritätskontrolle erfolgt in einem nächsten Schritt die Wahl der Zugriffspfade und deren Optimierung, bevor in einem dritten Schritt die eigentliche Codegenerierung oder eine interpretative Ausführung der Abfrage stattfindet. Bei der Codegenerierung wird ein Zugriffsmodul zur späteren Verwendung in einer Modulbibliothek abgelegt; alternativ dazu übernimmt der Interpreter die dynamische Kontrolle zur Ausführung der Anweisung.

5.3.2 Optimierung durch algebraische Umformung

Wie wir in Kap. 3 gesehen haben, können die Operatoren der Relationenalgebra zusammengesetzt werden. Man spricht von *äquivalenten Ausdrücken*, wenn die algebraischen Ausdrücke trotz unterschiedlicher Operatorenreihenfolge dasselbe Resultat erzeugen. Äquivalente Ausdrücke sind insofern interessant, als die Datenbankabfragen durch algebraische Umformungen optimiert werden können, ohne dass das Resultat verändert wird. Sie erlauben also, den Berechnungsaufwand zu reduzieren, und bilden damit einen wichtigen Teil der Optimierungskomponente eines relationalen Datenbanksystems.

Welch großen Einfluss die Reihenfolge von Operatoren auf den Berechnungsaufwand haben kann, soll anhand der früher diskutierten Beispielabfrage illustriert werden: Den Ausdruck

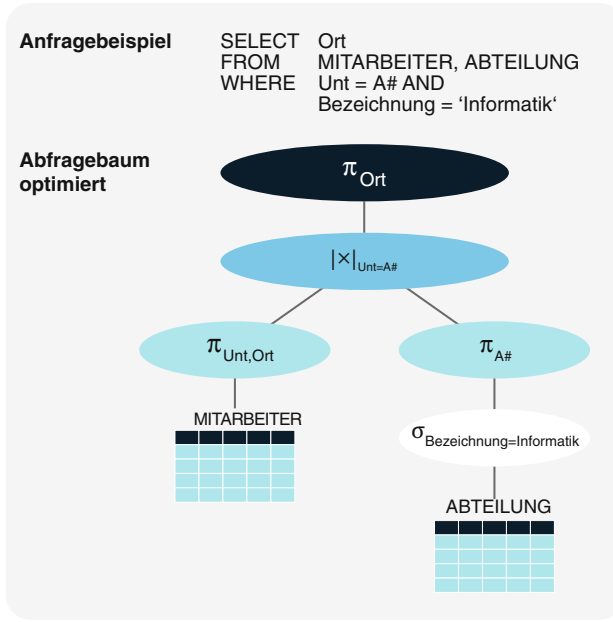

Abb. 5.8 Algebraisch optimierter Anfragebaum

$$\text{TABELLE} := \pi_{\text{Ort}} \left(\sigma_{\text{Bezeichnung=Informatik}} \left(\text{MITARBEITER} \left|\text{X}\right|_{\text{Unt=A\#}} \text{ABTEILUNG}\right)\right)$$

können wir – wie in Abb. 5.8 dargestellt – durch folgenden äquivalenten Ausdruck ersetzen:

$$\text{TABELLE} := \pi_{\text{Ort}} \left(\pi_{\text{Unt, Ort}} \left(\text{MITARBEITER}\right) \left|\text{X}\right|_{\text{Unt=A\#}} \pi_{\text{A\#}} \left(\sigma_{\text{Bezeichnung=Informatik}} \left(\text{ABTEILUNG}\right)\right)\right)$$

Dabei wird zuerst die Selektion ($\sigma_{\text{Bezeichnung=Informatik}}$) auf die Tabelle ABTEILUNG durchgeführt, da nur die Abteilung mit der Bezeichnung Informatik relevant ist. Anschließend werden zwei Projektionsoperationen berechnet, eine auf die Mitarbeitertabelle ($\pi_{\text{Unt,Ort}}$) und eine auf die bereits erzeugte Zwischentabelle ($\pi_{\text{A\#}}$) der Abteilung Informatik. Jetzt erst wird die Verbundoperation ($|\text{X}|_{\text{Unt=A\#}}$) über die Abteilungsnummer gebildet und anschließend auf die Ortschaften projiziert (π_{Ort}). Obwohl wir dasselbe Resultat erhalten, ist der Berechnungsaufwand auf diese Weise wesentlich geringer.

Allgemein lohnt es sich immer, *Projektions- und Selektionsoperatoren im Anfragebaum möglichst in die Nähe der «Blätter»* zu bringen. Dadurch können Zwischenresultate klein gehalten werden, bevor die zeitaufwendigen und deshalb teuren Verbundoperatoren kalkuliert werden. Gelingt eine Umformung eines Anfragebaums dank einer solchen Berechnungsstrategie, so spricht man von einer *algebraischen Optimierung*; für sie gelten die folgenden Prinzipien:

- Mehrere Selektionen auf ein und dieselbe Tabelle lassen sich zu einer einzigen verschmelzen, so dass das Selektionsprädikat nur einmal geprüft werden muss.
- Selektionen sind so früh wie möglich auszuführen, damit die Zwischenresultattabellen klein bleiben. Dies wird erreicht, wenn die Selektionsoperatoren so nahe wie möglich beim «Blattwerk» (bzw. bei den Ursprungstabellen) des Anfragebaumes angesiedelt werden.
- Projektionen sind ebenfalls so früh wie möglich durchzuführen, jedoch nie vor Selektionen. Sie reduzieren die Anzahl der Spalten und meistens auch die Anzahl der Tupel.
- Verbundoperatoren sind möglichst im Wurzelbereich des Anfragebaumes zu berechnen, da sie kostenaufwendig sind.

Neben der algebraischen Optimierung können durch den Einsatz effizienter Speicher- und Zugriffsstrukturen (vgl. Abschn. 5.2) bei der Bearbeitung einer relationalen Abfrage wesentliche Gewinne erzielt werden. So optimiert ein Datenbanksystem die einzelnen Selektions- und Verbundoperatoren aufgrund der Größe der Tabellen, der Sortierreihenfolgen, der Indexstrukturen etc. Gleichzeitig ist ein vernünftiges Modell zur Berechnung der Zugriffskosten unabdingbar, da oft mehrere Abarbeitungsvarianten in Frage kommen.

Man benötigt Kostenformeln, um den Berechnungsaufwand einer Datenbankabfrage kalkulieren zu können, beispielsweise für das sequenzielle Suchen innerhalb einer Tabelle, das Suchen über Indexstrukturen, das Sortieren von Tabellen oder Teiltabellen, das Ausnutzen von Indexstrukturen bezüglich der Verbundmerkmale oder das Berechnen von Gleichheitsverbundoperatoren über mehrere Tabellen hinweg. Eine solche Kostenformel berücksichtigt die Anzahl der Zugriffe auf die *physischen Seiten* (engl. *physical pages*) und bildet ein gewichtetes Maß für die Ein- und Ausgabeoperationen sowie für die CPU-Belastung (CPU = Central Processing Unit). Je nach Rechnerkonfiguration wird die Kostenformel wesentlich beeinflusst von der Zugriffszeit von externen Speichermedien und von Puffer- oder Hauptspeichern sowie der internen Rechenleistung.

5.3.3 Berechnen des Verbundoperators

Ein relationales Datenbanksystem muss über verschiedene Algorithmen verfügen, die die Operationen der Relationenalgebra bzw. des Relationenkalküls ausführen können. Gegenüber der Selektion von Tupeln aus einer einzigen Tabelle ist die Selektion aus mehreren Tabellen kostenaufwendig. Deshalb gehen wir in diesem Abschnitt näher auf

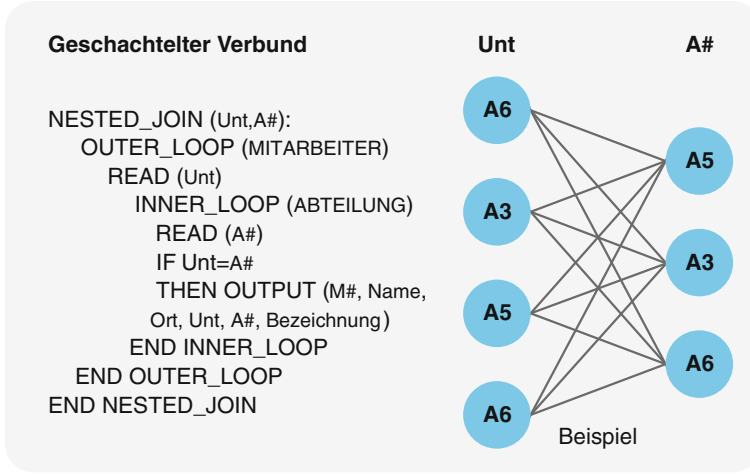

Abb. 5.9 Verbundberechnung durch Schachtelung

die verschiedenen Verbundstrategien ein, auch wenn gelegentliche Benutzer die Berechnungsvarianten kaum beeinflussen können.

Die Implementierung der Verbundoperation auf zwei Tabellen zielt darauf ab, jedes Tupel der einen Tabelle bezüglich des Verbundprädikats mit jedem Tupel der anderen Tabelle zu vergleichen und gegebenenfalls als zusammengesetztes Tupel in die Resultattabelle zu stellen. Konzentrieren wir uns auf die Berechnung eines Gleichheitsverbundes, so lassen sich im Wesentlichen zwei Verbundstrategien unterscheiden, nämlich der geschachtelte Verbund und der Sortier-Verschmelzungsverbund.

Geschachtelter Verbund

Bei einem geschachtelten Verbund (engl. *nested join*) zwischen der Tabelle R mit Merkmal A und der Tabelle S mit Merkmal B vergleichen wir *jedes Tupel in R mit jedem Tupel in S* darauf hin, ob das Verbundprädikat «R.A = S.B» erfüllt ist oder nicht. Weisen die beiden Tabellen R und S je n und m Tupel auf, so sind n mal m Vergleiche aufzuwenden.

Der Algorithmus des geschachtelten Verbundes berechnet das kartesische Produkt und prüft dabei die Erfüllung des Verbundprädikats. Da wir in einer äußeren Schleife OUTER_LOOP alle Tupel aus R mit allen Tupeln aus S der inneren Schleife INNER_ LOOP vergleichen, ist der Aufwand quadratisch. Falls für das Merkmal A oder B ein sogenannter Index existiert (siehe Abschn. 5.2.1), können wir den Aufwand für den geschachtelten Verbund reduzieren.

Abbildung 5.9 zeigt anhand der Abfrage über Mitarbeiter- und Abteilungsinformationen einen stark vereinfachten Algorithmus des geschachtelten Verbundes. Klar erscheinen die beiden Schleifen OUTER_LOOP und INNER_LOOP und lassen erkennen, dass dieser Algorithmus alle Tupel aus der Tabelle MITARBEITER mit allen Tupeln aus der Tabelle ABTEILUNG vergleicht.

Bei der Verbundberechnung in Abb. 5.9 besteht für das Merkmal A# ein Index, da diese Nummer der Primärschlüssel[1] der Tabelle ABTEILUNG ist. Das Datenbanksystem nutzt die Indexstruktur der Abteilungsnummer aus, indem bei der inneren Schleife nicht jedes Mal sequenziell von Tupel zu Tupel die gesamte Tabelle ABTEILUNG abgesucht, sondern direkt über den Index gezielt zugegriffen wird. Im besten Fall besteht auch für das Merkmal «Unt» (Unterstellung) der Tabelle MITARBEITER ein Index, den das Datenbanksystem zu Optimierungszwecken verwenden kann. Dieses Beispiel zeigt, welche wichtige Rolle der geeigneten Auswahl von Indexstrukturen durch die Datenbankadministratoren zukommt.

Ein effizienterer Algorithmus als derjenige des geschachtelten Verbundes ist dann möglich, wenn die Tupel aus den Tabellen R und S bezüglich der beiden Merkmale A und B des Verbundprädikats physisch aufsteigend oder absteigend sortiert vorliegen. Dazu muss vor der Berechnung des eigentlichen Verbundes eventuell eine interne Sortierung vorgenommen werden, um die Tabelle R oder die Tabelle S oder beide zusammen zu ordnen. Zur Berechnung des Verbundes genügt es dann, die Tabellen nach auf- oder absteigenden Merkmalswerten des Verbundprädikats zu durchlaufen und gleichzeitig Wertvergleiche zwischen A und B vorzunehmen. Diese Strategie wird wie folgt charakterisiert:

Sortier-Verschmelzungsverbund
Der Sortier-Verschmelzungsverbund (engl. *sort-merge join*) setzt voraus, dass die beiden Tabellen R und S mit dem Verbundprädikat R.A = S.B bezüglich der Merkmalswerte A aus R und B aus S sortiert vorliegen. Der Algorithmus berechnet den Verbund, indem er die *Vergleiche in der sortierten Reihenfolge* vornimmt. Sind die Merkmale A und B eindeutig definiert (z. B. als Primär- und als Fremdschlüssel), so ist der Aufwand linear.

Die Abb. 5.10 zeigt einen allgemeinen Algorithmus zum Sortier-Verschmelzungsverbund. Zuerst werden die beiden Tabellen anhand der im Verbundprädikat vorkommenden Merkmale sortiert. Danach werden sie in der Sortierreihenfolge durchlaufen und die Vergleiche R.A = S.B ausgeführt. Im allgemeinen Fall können die beiden Merkmale A und B in komplexer Beziehung zueinander stehen, d. h. ein und derselbe Merkmalswert kann sowohl in der Spalte R.A als auch in S.B mehrfach vorkommen. Dann gibt es bekanntlich zu einem bestimmten Tupel aus R mehrere verbundverträgliche Tupel aus S und umgekehrt. Somit müssen für solche Merkmalswerte die entsprechenden Tupel aus R und S durch einen geschachtelten Teilverbund miteinander verknüpft werden.

In unserer Abfrage in den Tabellen MITARBEITER und ABTEILUNG stellen wir fest, dass der Sortier-Verschmelzungsschritt wegen des Schlüsselmerkmals A# linear von

[1] Das Datenbanksystem baut für jeden Primärschlüssel automatisch eine Indexstruktur auf; bei zusammengesetzten Schlüsseln werden erweiterte Indexstrukturen verwendet.

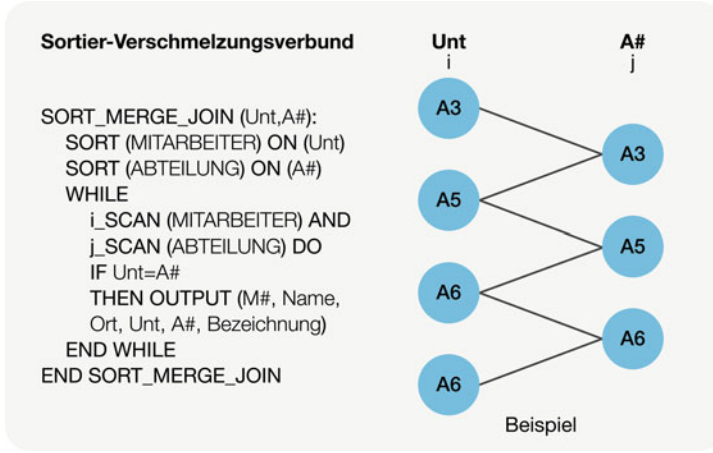

Abb. 5.10 Durchlaufen der Tabellen in Sortierreihenfolge

den Vorkommen der Tupel abhängt. Die beiden Tabellen MITARBEITER und ABTEILUNG müssen zur Berechnung des Verbundes lediglich einmal durchlaufen werden.

Grundsätzlich kann die Auswahl einer geeigneten Verbundstrategie – wie überhaupt jeder Zugriffsstrategie – durch das Datenbanksystem nicht a priori getroffen werden. Im Gegensatz zur algebraischen Optimierung hängt sie vom aktuellen inhaltlichen Zustand der Datenbank ab. Aus diesem Grund ist es wesentlich, dass die in den Systemtabellen enthaltenen statistischen Angaben entweder periodisch oder durch die Datenbankspezialisten ausgelöst regelmäßig aktualisiert werden.

5.4 Parallelisierung mit Map/Reduce

Für die Analyse von umfangreichen Datenvolumen strebt man eine Aufgabenteilung an, die Parallelität ausnutzt. Nur so kann ein Resultat in vernünftiger Zeit berechnet werden. Das Verfahren Map/Reduce kann sowohl für Rechnernetze als auch für Großrechner verwendet werden, wird hier aber für den verteilten Fall diskutiert.

In einem verteilten Rechnernetz, oft bestückt mit billigen und horizontal skalierten Komponenten, lassen sich Berechnungsvorgänge leichter verteilen als Datenbestände. Aus diesem Grund hat sich das Map/Reduce-Verfahren bei webbasierten Such- und Analysearbeiten durchgesetzt. Es nutzt Parallelität bei der Generierung von einfachen Datenextrakten und deren Sortierung aus, bevor die Resultate ausgegeben werden:

- **Map-Phase**: Hier werden Teilaufgaben an diverse Knoten des Rechnernetzes verteilt, um Parallelität auszunutzen. In den einzelnen Knoten werden aufgrund einer Anfrage

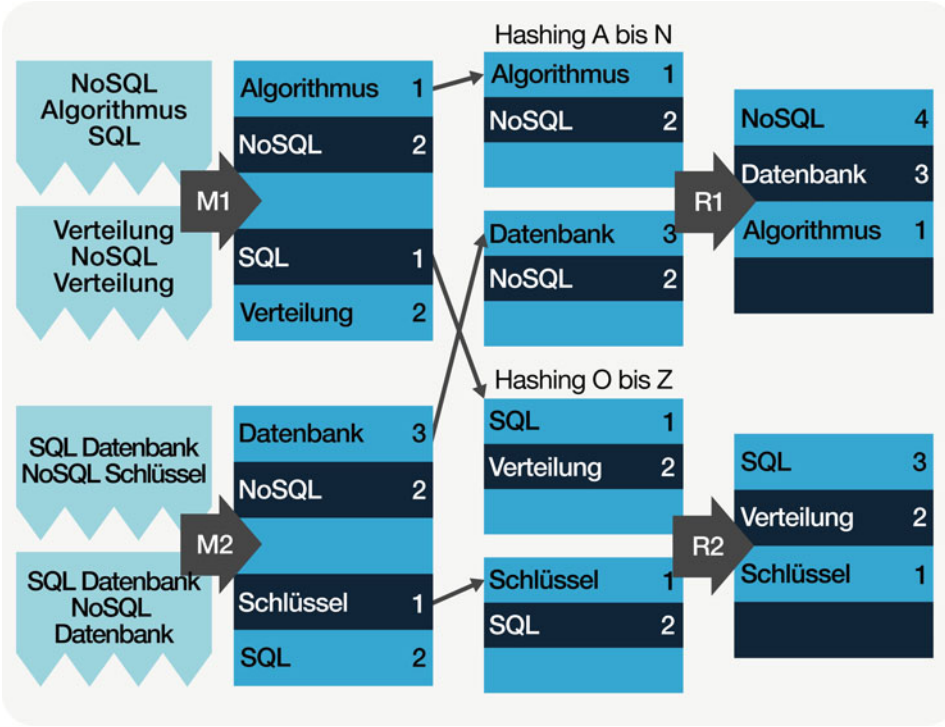

Abb. 5.11 Häufigkeiten von Suchbegriffen mit Map/Reduce

einfache Key/Value-Paare extrahiert, die danach (z. B. mit Hashing) sortiert und als Zwischenergebnisse ausgegeben werden.

- **Reduce-Phase**: Für jeden Schlüssel resp. Schlüsselbereich fasst die Phase Reduce die obigen Zwischenergebnisse zusammen und gibt sie als Endresultat aus. Dieses besteht aus einer Liste von Schlüsseln mit den zugehörigen aggregierten Wertvorkommen.

In der Abb. 5.11 ist ein einfaches Beispiel eines Map/Reduce-Verfahrens gegeben: Es sollen Dokumente oder Webseiten auf Begriffe wie Algorithmus, Datenbank, NoSQL, Schlüssel, SQL und Verteilung durchsucht werden. Gesucht werden die Häufigkeiten dieser Begriffe in verteilten Datenbeständen.

Die Map-Phase besteht hier aus den beiden parallelen Map-Funktionen M1 und M2. M1 generiert eine Liste von Key/Value-Paaren, wobei als Key die Suchbegriffe und als Value deren Häufigkeiten aufgefasst werden. M2 führt zur selben Zeit wie M1 einen entsprechenden Suchvorgang auf einem anderen Rechnerknoten mit weiteren Dokumenten resp. Webseiten durch. Danach werden die Teilresultate mit der Hilfe eines Hashing-Algorithmus alphabetisch sortiert. Im oberen Teil der Zwischenergebnisse dienen die Anfangsbuchstaben A bis N der Schlüssel (hier Suchbegriffe) als Sortierkriterium, im unteren Teil die Buchstaben O bis Z.

In der Reduce-Phase werden in Abb. 5.11 die Zwischenergebnisse zusammengefasst. Die Reduce-Funktion R1 addiert die Häufigkeiten für die Begriffe mit Anfangsbuchstaben A bis N, R2 diejenigen von O bis Z. Als Antwort, nach Häufigkeiten der gesuchten Begriffe sortiert, erhält man eine Liste mit NoSQL (4), Datenbank (3) und Algorithmus (1) sowie eine zweite mit SQL (3), Verteilung (2), und Schlüssel (1). Im Endresultat werden die beiden Listen kombiniert und nach Häufigkeiten geordnet.

Das Map/Reduce-Verfahren basiert auf bekannten funktionalen Programmiersprachen wie LISP (List Processing). Dort berechnet die Funktion map() für alle Elemente einer Liste ein Zwischenergebnis als modifizierte Liste. Die Funktion reduce() aggregiert einzelne Ergebnisse und reduziert diese zu einem Ausgabewert.

Map/Reduce ist von den Entwicklern bei Google für immense semi-strukturierte und unstrukturierte Datenmengen verfeinert und patentiert worden. Diese Funktion ist aber auch in vielen Open Source Werkzeugen verfügbar. Bei NoSQL-Datenbanken (siehe Kap. 7) spielt dieses Verfahren eine wichtige Rolle; verschiedene Hersteller nutzen den Ansatz zur Abfrage ihrer Datenbankeinträge. Dank der Parallelisierung eignet sich das Map/Reduce-Verfahren nicht nur für die Datenanalyse, sondern auch für Lastverteilungen, Datentransfer, verteilte Suchvorgänge, Kategorisierungen oder Monitoring.

5.5 Schichtarchitektur

Bei der Systemarchitektur von Datenbanksystemen gilt als unabdingbares Prinzip, künftige Veränderungen oder Erweiterungen lokal begrenzen zu können. Wie beim Implementieren von Betriebssystemen oder von anderen Softwarekomponenten führt man auch bei relationalen wie bei nicht-relationalen Datenbanksystemen *voneinander unabhängige Systemebenen* ein, die über vordefinierte Schnittstellen miteinander kommunizieren.

Die fünf Ebenen der Systemarchitektur, hier angelehnt an die relationale Datenbanktechnologie, erläutern wir im Überblick in Abb. 5.12. Gleichzeitig ordnen wir diesen Ebenen die wichtigsten Funktionen der vorangehenden Abschnitten der Kap. 4 und 5 zu:

Schicht 1: Mengenorientierte Schnittstelle
In der ersten Schicht werden Datenstrukturen beschrieben, Operationen auf Mengen bereitgestellt, Zugriffsbedingungen definiert und Konsistenzanforderungen geprüft (vgl. Kap. 4). Entweder bereits bei vorgezogener Übersetzung und Generierung von Zugriffsmodulen oder erst zur Laufzeit müssen die Syntax geprüft, Namen aufgelöst und Zugriffspfade ausgewählt werden. Bei der Auswahl von Zugriffspfaden können wesentliche Optimierungen vorgenommen werden.

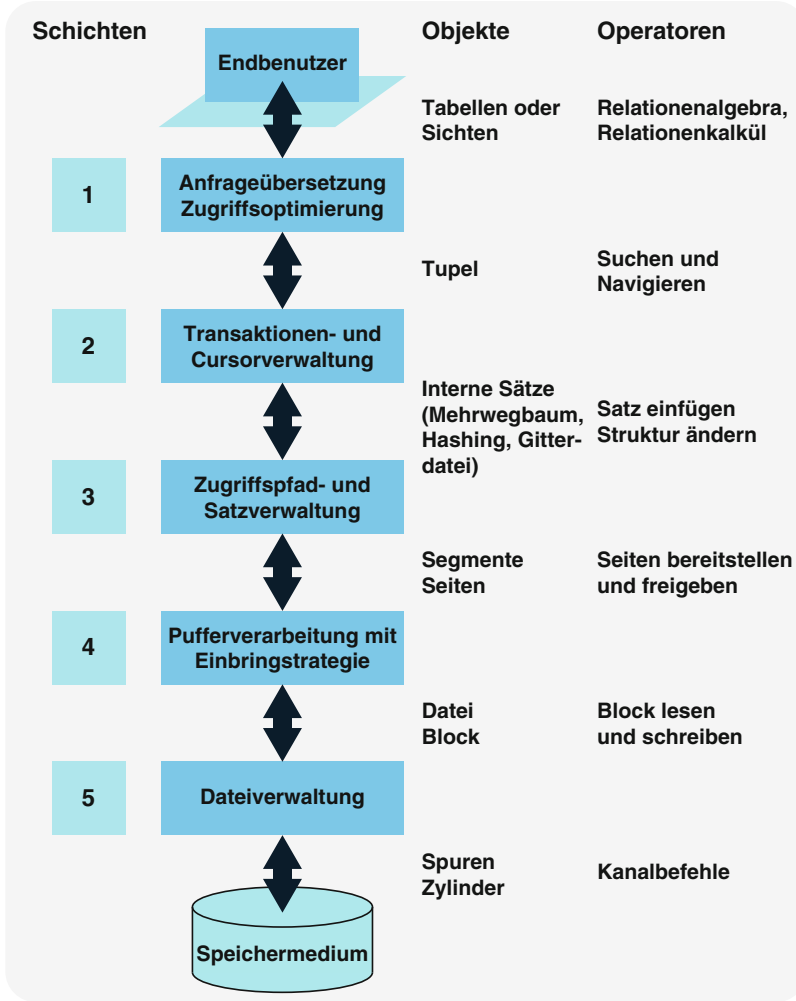

Abb. 5.12 Fünf-Schichtenmodell für relationale Datenbanksysteme

Schicht 2: Satzorientierte Schnittstelle

Die zweite Schicht überführt logische Sätze und Zugriffspfade in physische Strukturen. Ein Cursorkonzept erlaubt das Navigieren oder Durchlaufen von Datensätzen nach der physischen Speicherungsreihenfolge, das Positionieren bestimmter Datensätze innerhalb einer Tabelle oder das Bereitstellen von Datensätzen in wertabhängiger Sortierreihenfolge. Dabei muss mit Hilfe einer Transaktionenverwaltung gemäß Abschn. 4.2 gewährleistet werden, dass die Konsistenz der Datenbank erhalten bleibt und sich mehrere Benutzeranforderungen nicht gleichzeitig behindern.

Schicht 3: Speicher- und Zugriffsstrukturen
Die dritte Schicht implementiert physische Datensätze und Zugriffspfade auf Seiten. Die Zahl der Seitenformate ist zwar begrenzt, jedoch sollten neben Baumstrukturen und Hash-Verfahren künftig auch mehrdimensionale Datenstrukturen unterstützt werden. Diese typischen Speicherstrukturen sind so ausgelegt, dass sie den Zugriff auf externe Speichermedien effizient bewerkstelligen. Daneben können physische Clusterbildung und mehrdimensionale Zugriffspfade bei der Satz- und Zugriffspfadverwaltung weitere Optimierungen erzielen.

Schicht 4: Seitenzuordnung
Aus Effizienzgründen und für die Implementierung von Recovery-Verfahren unterteilt die vierte Schicht den linearen Adressraum in sogenannte Segmente mit einheitlichen Seitengrenzen. Auf Anforderung werden durch die Dateiverwaltung in einem Puffer Seiten bereitgestellt. Umgekehrt können durch eine Einbringungs- oder Ersetzungsstrategie Seiten in den Datenbankpuffer eingebracht oder in diesem ersetzt werden. Es gibt nicht nur die direkte Zuordnung von Seiten zu Blöcken, sondern auch indirekte Zuordnungen wie sogenannte Schattenspeicher- oder Cacheverfahren, durch die sich mehrere Seiten atomar in den Datenbankpuffer einbringen lassen.

Schicht 5: Speicherzuordnung
Die fünfte Schicht realisiert Speicherzuordnungsstrukturen und bietet der übergeordneten Schicht eine blockorientierte Dateiverwaltung. Dabei bleiben die Hardware-Eigenschaften den datei- und blockbezogenen Operationen verborgen. Normalerweise unterstützt die Dateiverwaltung dynamisch wachsende Dateien mit definierbaren Blockgrößen. Außerdem sollte die Clusterbildung von Blöcken möglich sein sowie die Ein- und Ausgabe mehrerer Blöcke mit Hilfe einer einzigen Operation.

5.6 Nutzung unterschiedlicher Speicherstrukturen

Viele webbasierte Anwendungen setzen für die unterschiedlichen Dienste adäquate Datenhaltungssysteme ein. Die Nutzung einer einzigen Datenbanktechnologie, beispielsweise der relationalen, genügt nicht mehr: Die vielfältigen Anforderungen an Konsistenz, Verfügbarkeit und Ausfalltoleranz verlangen nicht zuletzt aufgrund des CAP-Theorems nach einem Mix von Datenhaltungssystemen.

In der Abb. 5.13 ist ein elektronischer Shop schematisch dargestellt. Um eine hohe Verfügbarkeit und Ausfalltoleranz zu garantieren, wird ein Key/Value-Speichersystem (siehe Abschn. 7.2) für die Session-Verwaltung sowie den Betrieb der Einkaufswagen eingesetzt. Die Bestellungen selber werden im Dokumentenspeicher abgelegt (Abschn. 7.4), die Kunden- und Kontoverwaltung erfolgt mit einem relationalen Datenbanksystem.

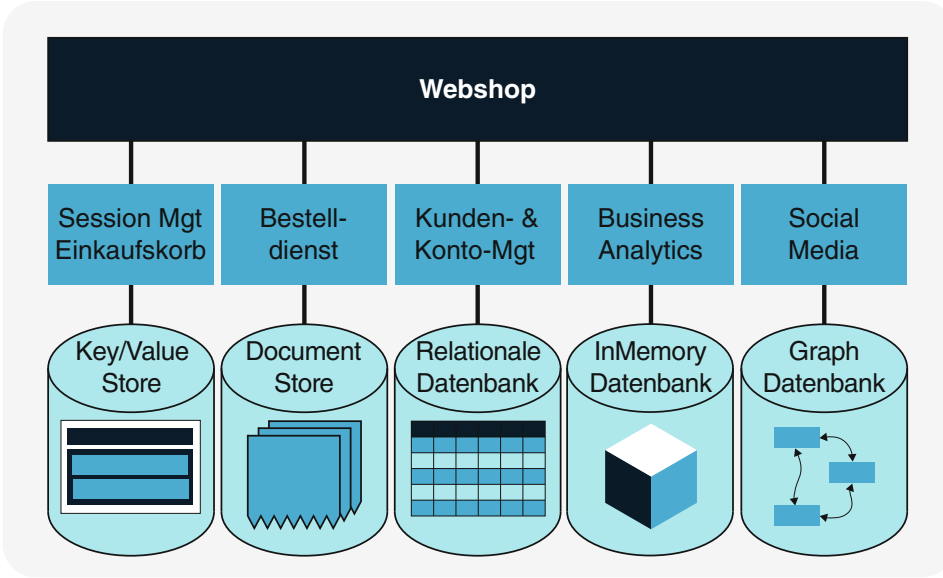

Abb. 5.13 Nutzung von SQL- und NoSQL-Datenbanken im Webshop

Bedeutend für den erfolgreichen Betrieb eines Webshops ist das Performance Management. Mit der Hilfe von Web Analytics werden wichtige Kenngrössen (engl. *key performance indicators–KPI*) der Inhalte (engl. *content*) wie der Webbesucher in einem Data Warehouse (Abschn. 6.5) aufbewahrt. Mit spezifischen Werkzeugen (Data Mining, Predictive Business Analysis) werden die Geschäftsziele und der Erfolg der getroffenen Maßnahmen regelmäßig ausgewertet. Da die Analysearbeiten auf dem mehrdimensionalen Datenwürfel (engl. *data cube*) zeitaufwendig sind, wird dieser InMemory gehalten.

Die Verknüpfung des Webshops mit sozialen Medien drängt sich aus unterschiedlichen Gründen auf. Neben der Ankündigung von Produkten und Dienstleistungen kann analysiert werden, ob und wie die Angebote bei den Nutzern ankommen. Bei Schwierigkeiten oder Problemfällen kann mit gezielter Kommunikation und geeigneten Maßnahmen versucht werden, einen möglichen Schaden abzuwenden oder zu begrenzen. Darüber hinaus hilft die Analyse von Weblogs sowie aufschlussreicher Diskussionen in sozialen Netzen, Trends oder Innovationen für das eigene Geschäft zu erkennen. Falls die Beziehungen unterschiedlicher Bedarfsgruppen analysiert werden sollen, drängt sich der Einsatz von Graphdatenbanken auf (vgl. Abschn. 7.6).

Die Dienste des Webshops und die Einbindung von heterogenen SQL- und NoSQL-Datenbanken werden mit dem Baukasten REST (REpresentational State Transfer) realisiert. Dieser besteht aus den folgenden fünf Bauelementen:

- **Adressierung**: Die Identifikation von Ressourcen im Web erfolgt mit dem Uniform Resource Identifier oder abgekürzt URI. Eine Ressource im Web stellt beispielsweise eine Webseite, eine Datei, ein Dienst oder ein Email-Empfänger dar. Ein URI besteht aus den folgenden fünf Teilen: einem Schema (Typ des URI resp. Protokoll), einer Autorität (Anbieter oder Server), einem Pfad, einer Abfrage (Angaben zur Identifizierung der Ressource) und einem optionalen Fragment (Referenz innerhalb einer Ressource). Ein Beispiel wäre: http://eShop.com/customers/12345.
- **Verknüpfung**: Das Konzept für die Verknüpfungen von Ressourcen basiert auf Hyperlinks, d. h. auf elektronischen Verweisen. Ein Hyperlink oder einfach Link ist ein Querverweis in einem Hypertext-Dokument, der entweder auf eine andere Stelle im Dokument oder auf ein anderes elektronisches Dokument verweist. Ein Beispiel wäre: Hier findest du deinen eShop für zeitgenössische Literatur.
- **Standardmethoden**: Jede Ressource im Web kann mit einem Satz von Methoden bearbeitet werden. Die Standardmethoden von HTTP (HyperText Transfer Protocol) wie GET (eine Ressource wird von einem Server angefordert), POST (Daten werden an einen Server geschickt) oder DELETE (löscht eine Ressource) dienen dazu, eine einheitliche Schnittstelle zu verwenden. Damit bleibt garantiert, dass andere Webdienste mit allen Ressourcen jederzeit kommunizieren können.
- **Darstellungsformen**: Je nach Anwendung und Bedürfnis muss ein Server, der auf REST basiert, unterschiedliche Darstellungen einer Ressource ausliefern. Neben dem Standardformat in HTML (HyperText Markup Language) werden Ressourcen oft in XML (eXentsible Markup Language) bereitgestellt oder angefordert.
- **Zustandslosigkeit**: Weder Anwendungen noch Server sollen Zustandsinformationen zwischen Nachrichten austauschen. Dies begünstigt die Skalierbarkeit von Diensten, wie beispielsweise die Lastverteilung auf unterschiedliche Rechnerknoten (vgl. Map/Reduce-Verfahren).

Der Baukasten REST liefert ein Entwurfsmuster für die Entwicklung von verteilten Anwendungen mit heterogenen SQL- und NoSQL-Komponenten. Er garantiert eine Skalierung in die Breite, falls das Geschäftsvolumen zunimmt oder neue Dienste notwendig werden.

5.7 Literatur

Einige Arbeiten widmen sich ausschließlich der Architektur von Datenbanksystemen, teilweise beschränkt auf einzelne Ebenen der Schichtarchitektur. Härder (1978) sowie Härder und Rahm (2001) stellen grundlegende Konzepte für die Implementierung relationaler Datenbanksysteme vor (Schichtenmodell). Auch das Datenbankhandbuch von Lockemann und Schmidt (1993) behandelt schwerpunktmäßig Aspekte der Datenarchi-

tektur. Optimierungsfragen werden in Maier (1983); Paredaens et al. (1989) und Ullman (1982) von der theoretischen Seite her angegangen.

Wiederhold (1983) befasst sich mit Speicherstrukturen für Datenbanksysteme. Der Mehrwegbaum stammt von Bayer (Bayer 1992), Hash-Funktionen bieten Maurer und Lewis (1975) im Überblick, und die Gitterdatei geht auf Nievergelt et al. (1984) zurück. Hash-Funktionen werden zu unterschiedlichen Zwecken in der Informatik und Wirtschaftsinformatik verwendet. Bei Big Data und insbesondere bei Schlüssel-Wert-Datenbanken wurde das Consistent Hashing eingeführt, das den Adressraum der Adressen in einem Ring vereint. So verwendet Amazon.com in ihrem Key/Value Store Dynamo diese Technik (DeCandia et al. 2007). Ein Grundlagenpapier zum Consistent Hashing und zu verteilten Protokollen wurde von den MIT-Forschern Karger et al. (1997) verfasst.

In den Grundlagenwerken über NoSQL von Celko (2014); Edlich et al. (2011) sowie Sadalage und Fowler (2013) werden die Grundlagen zu Big Data und NoSQL erläutert. Insbesondere enthalten diese Werke eine Einführung in das Verfahren Map/Reduce, CAP-Theorem und teilweise auch zu Consistent Hashing. Aufschlussreich ist das Werk von Redmond und Wilson (2012), das die Grundkonzepte von SQL- und NoSQL-Datenbanken anhand von sieben konkreten Systemen erläutert.

Im Jahre 2010 erhielt Google Inc. ein Patent der USA für das Map/Reduce-Verfahren, welches die nebenläufige Berechnung umfangreicher Datenmengen auf massiv verteilten Rechnern betrifft. Die Forscher Dean und Ghemawat (2004) von Google Inc. haben das Map/Reduce-Verfahren an einem Symposium für Betriebssysteme in San Franciso vorgestellt.

Das Paradigma Representational State Transfer oder REST ist ein Architekturvorschlag zur Entwicklung von Webdiensten. Es wurde vom World Wide Web Consortium (W3C 2014) vorgeschlagen und dient heute als Grundlage für die Entwicklung webbasierter Anwendungen. Tilkov (2011) hat dazu ein Grundlagenwerk veröffentlicht.

Postrelationale Datenbanken

6

6.1 Die Grenzen von SQL – und darüber hinaus

Die relationale Datenbanktechnologie und insbesondere SQL-basierte Datenbanken haben sich in den 1980er- und 1990er-Jahren breit im Markt durchgesetzt. Auch heute noch sind SQL-Datenbanken ein De-facto-Standard für die meisten betrieblichen Datenbankanwendungen. Diese bewährte und breit abgestützte Technologie wird voraussichtlich noch Dekaden im Einsatz bleiben. Dennoch stellt sich die Frage, wohin die Reise in der Datenbankwelt führt. Da ist die Rede von NoSQL-Datenbanken, Graphdatenbanken, verteilten Datenbanksystemen; von temporalen, deduktiven, semantischen, objektorientierten, unscharfen, oder versionenbehafteten Datenbanksystemen etc. Was verbirgt sich hinter all diesen schillernden Begriffen? Das vorliegende Kapitel erläutert einige postrelationale Konzepte und zeigt Methoden und Entwicklungstendenzen auf, wobei die Auswahl subjektiv bleibt. Die NoSQL-Datenbanken werden in Kap. 7 beschrieben.

Beim klassischen Relationenmodell und bei den entsprechenden SQL-basierten Datenbanksystemen treten zugegebenermassen einige Mängel in Erscheinung, die einerseits von *erweiterten Anforderungen in neuen Anwendungsgebieten*, andererseits von *prinzipiellen Grenzen von SQL* herrühren. Die relationale Datenbanktechnik ist breit einsetzbar und kann als Generalist unter den Datenbankmodellen angesehen werden. Es gibt allerdings Nischen und Anwendungsszenarien, in denen die Transaktions- und Konsistenzorientierung SQL-basierter Datenbanken im Weg ist, beispielsweise bei der Anforderung an hochperformante Verarbeitung umfangreicher Datenmengen. In diesen Fällen lohnt es sich, spezialisierte Werkzeuge zu verwenden, welche für diese Nische effizienter sind, auch wenn eine SQL-Datenbank theoretisch einsetzbar wäre. Ebenfalls gibt es Anwendungen, an denen SQL und deren Grundlage, die Relationenalgebra, an prinzipielle Grenzen stösst. SQL ist zwar eine relational vollständige Sprache, aber nicht

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_6

Turing-komplett: Es gibt berechenbare Probleme, welche sich mit relationalen Operationen nicht lösen lassen. Dazu gehört insbesondere die Klasse rekursiver Probleme. Ein Beispiel dafür ist die Netzwerkanalyse mit Zyklen.

SQL ist und bleibt nach wie vor die wichtigste und am weitesten verbreitete Datenbanksprache. Heute gibt es eine große Auswahl kommerzieller Produkte mit erweiterter Datenbankfunktionalität, einige davon sind Open Source. Für Praktiker ist es nicht einfach, sich im Dschungel der Möglichkeiten zurechtzufinden. Auch sind oft der Umstellungsaufwand einerseits und der wirtschaftliche Nutzen andererseits zu wenig ersichtlich. Viele Unternehmen brauchen deshalb noch manche Kopfarbeit, um ihre Anwendungsarchitektur zukunftsgerichtet zu planen und eine sinnvolle Produktwahl zu treffen. Kurz gesagt, es fehlt an klaren Architekturkonzepten und Migrationsstrategien für den optimalen Einsatz postrelationaler Datenbanktechnologien.

In diesem und dem nächsten Kapitel geben wir eine Auswahl von Problemstellungen und Lösungsansätzen. Einige der in rein relationalen Datenbanken nicht abgedeckten Anforderungen können mit punktuellen Erweiterungen relationaler Datenbanksysteme befriedigt werden, andere müssen wir mit grundlegend neuen Konzepten und Methoden angehen. Beide Entwicklungstendenzen sind unter dem Begriff der *postrelationalen Datenbanksysteme* zusammengefasst. Unter dem Begriff postrelational subsummieren wir auch NoSQL, geben diesen Datenbanken aber ein eigenes Kapitel.

6.2 Föderierte Datenbanken

Dezentrale oder *föderierte Datenbanken* (engl. *federated databases*) finden überall dort Anwendung, wo Daten an verschiedenen Orten gesammelt, gepflegt und verarbeitet werden sollen. Eine *Datenbank ist verteilt,* wenn die Datenbasis auf verschiedenen räumlich getrennten Rechnern liegt. Wird die gesamte Datenbasis für den Lastausgleich redundant auf mehrere Rechner kopiert, spricht man von *Replikation*. Bei einer *Fragmentierung* wird die Datenbasis für erhöhtes Datenvolumen effektiv in verschiedene kleinere Teile, sogenannte Fragmente, aufgeteilt, und auf mehrere Rechner verteilt. Fragmente werden heute oft auch Partitionen oder *Shards* (Scherben) genannt; das Konzept der Fragmentierung heisst analog dazu Partitionierung oder *Sharding*. Eine verteilte Datenbank ist *föderiert*, wenn sie zwar durch *mehrere physische Datenfragmente auf örtlich verteilten Rechnern* gehalten wird, aber durch ein *einziges logisches Datenbankschema* zugänglich bleibt. Die Anwendenden einer föderierten Datenbank haben sich lediglich mit der logischen Sicht auf die Daten zu befassen, um die physischen Fragmente brauchen sie sich nicht zu kümmern. Das Datenbanksystem selbst übernimmt es, Datenbankoperationen lokal oder bei Bedarf verteilt auf verschiedenen Rechnern durchzuführen.

Ein einfaches Beispiel einer föderierten relationalen Datenbank zeigt Abb. 6.1. Das Zerlegen der Tabellen MITARBEITER und ABTEILUNG in verschiedene physische Fragmente ist eine wesentliche Aufgabe der Datenbankadministratoren, nicht

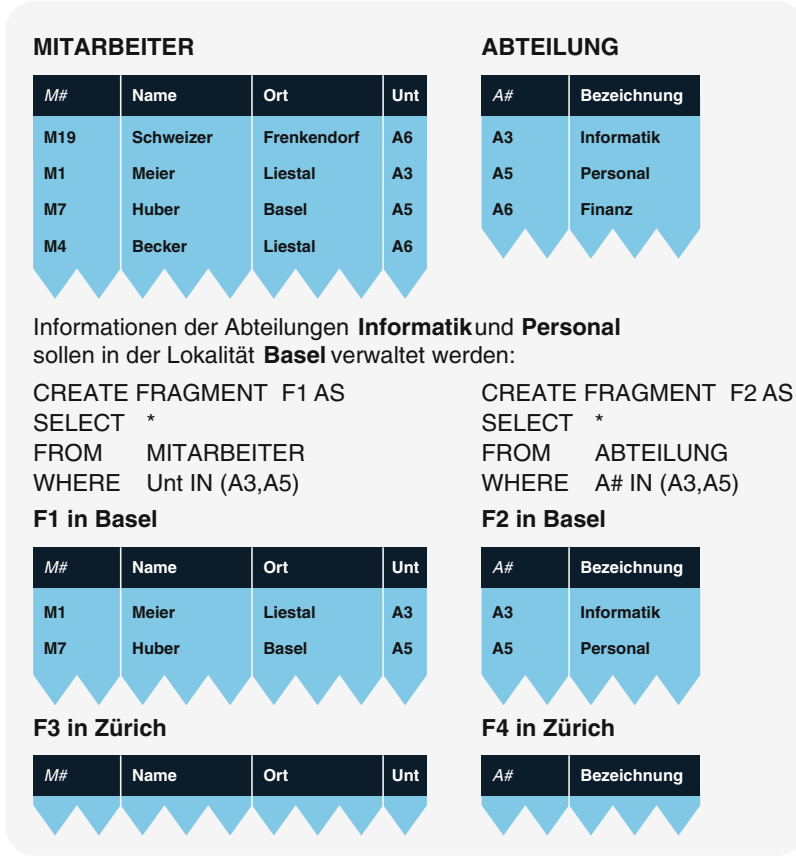

Abb. 6.1 Horizontale Fragmentierung der Tabelle MITARBEITER und ABTEILUNG

der Anwendenden. Gemäß unserem Beispiel seien die Abteilungen Informatik und Personal geografisch in Basel und die Abteilung Finanz in Zürich lokalisiert. Fragment F1 als Teiltabelle der Tabelle MITARBEITER enthält nur Mitarbeitende der Informatik- und der Personalabteilung. Fragment F2 aus der ursprünglichen Tabelle ABTEILUNG beschreibt auf analoge Art diejenigen Abteilungen, die in Basel lokalisiert sind. Die beiden Fragmente F3 und F4 beziehen sich auf den Standort Zürich, und zwar hinsichtlich der Mitarbeitenden und der Abteilungen.

Man spricht von *horizontalen Fragmenten*, wenn eine bestimmte Tabelle horizontal zerlegt worden ist und so die ursprüngliche Form der Tabellenzeilen erhalten bleibt. Dabei sollten sich die verschiedenen Fragmente nicht gegenseitig überlappen, zusammen jedoch die ursprüngliche Tabelle bilden.

Anstelle einer horizontalen Zerlegung kann eine Tabelle in *vertikale Fragmente unterteilt* werden, indem mehrere Spalten mit dem Identifikationsschlüssel versehen zusammengefasst, die Tupel also zergliedert werden. Ein Beispiel dafür wäre eine Tabelle

MITARBEITER, bei der Teile wie Lohn, Qualifikationsstufe, Entwicklungspotenzial etc. aus Gründen der Diskretion in einem vertikalen Fragment in der Personalabteilung gehalten würden. Die restlichen Merkmale hingegen könnten als weiteres Fragment in den verschiedenen Fachabteilungen vorliegen. Mischformen zwischen horizontalen und vertikalen Fragmenten sind ebenfalls möglich.

Eine wichtige Aufgabe eines föderierten Datenbanksystems ist die *Gewährleistung der lokalen Autonomie*. Die Anwendenden können auf ihren lokalen Datenbeständen autonom arbeiten, und zwar auch dann, wenn verschiedene Rechnerknoten im Netz nicht zur Verfügung stehen.[1]

Neben lokaler Autonomie ist das *Prinzip der dezentralen Verarbeitung* äußerst wichtig. Es bedeutet, dass das Datenbanksystem lokal in den verschiedenen Netzknoten Abfragen verarbeiten kann. Für solche dezentrale Anwendungen, die Daten aus verschiedenen Fragmenten beanspruchen, muss das Datenbanksystem einen entfernten Zugriff zum Lesen und Verändern von Tabellen ermöglichen. Dazu muss es ein verteiltes Transaktions- und Recoverykonzept zur Verfügung stellen. Solche Konzepte erfordern bei verteilten Datenbanken besondere Schutzmechanismen.

Von Bedeutung ist die *interne Verarbeitungsstrategie für verteilte Datenbankabfragen,* wie das Beispiel von Interessierten an einer Liste der Namen von Mitarbeitenden und der Abteilungsbezeichnungen in Abb. 6.2 darlegt. Die Abfrage kann ohne Einbeziehung von Fragmentangaben mit dem üblichen SQL formuliert werden. Die Aufgabe des Datenbanksystems besteht darin, für diese dezentrale Abfrage eine optimale Berechnungsstrategie festzusetzen. Sowohl die Tabelle MITARBEITER als auch die Tabelle ABTEILUNG liegen fragmentiert in Basel und Zürich vor. Deshalb werden *gewisse Berechnungen lokal und parallel* durchgeführt. Jeder Knoten organisiert unabhängig vom anderen den Verbund zwischen dem Mitarbeiter- und dem Abteilungsfragment. Nach diesen Teilberechnungen wird das Ergebnis durch Vereinigung der Teilresultate gebildet.

Zur zusätzlichen Optimierung werden in den einzelnen Knoten zunächst Projektionen auf die im Resultat gewünschten Merkmale Name und Bezeichnung realisiert. Sodann werden auf den reduzierten Tabellenfragmenten die Verbundoperatoren in Basel und in Zürich getrennt berechnet. Schließlich werden die beiden Zwischenresultate vor der Vereinigung ein weiteres Mal reduziert, indem sie auf die gewünschten Namen und Bezeichnungen projiziert werden.

Allgemein ist bei der Berechnung dezentraler Abfragen typisch, dass Vereinigungs- und Verbundoperatoren spät evaluiert werden. Dies unterstützt *hohe Parallelität* in der Verarbeitung und fördert die Performance bei verteilten Abfragen. Die Optimierungsmaxime lautet also, Vereinigungsoperatoren im Anfragebaum möglichst in die Nähe des Wurzelknotens zu bringen, hingegen Selektionen und Projektionen im Blattbereich des Anfragebaumes anzusetzen.

[1] Periodisch extrahierte Tabellenteile (sog. Snapshots) erhöhen diese lokale Autonomie.

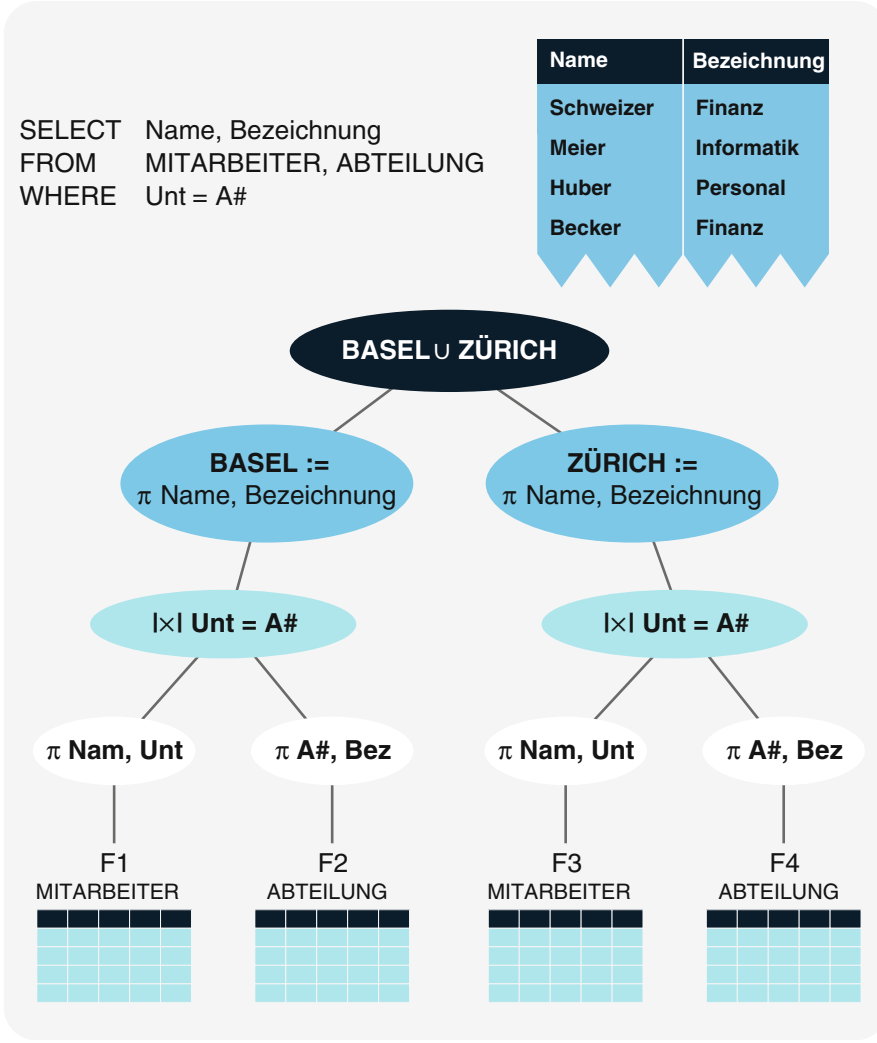

Abb. 6.2 Optimierter Anfragebaum für eine verteilte Verbundstrategie

Föderiertes Datenbanksystem
Ein föderiertes Datenbanksystem erfüllt folgende Bedingungen:

- Es unterstützt ein einziges logisches Datenbankschema und mehrere physische Fragmente auf örtlich verteilten Rechnern.
- Es garantiert *Transparenz bezüglich der Verteilung von Datenbanken*, so dass Ad-hoc-Abfragen oder Anwendungsprogramme auf die physische Verteilung der Daten, d. h. auf die Partitionierung, keine Rücksicht nehmen müssen.

- Es gewährleistet lokale Autonomie, d. h., es erlaubt das lokale Arbeiten auf seinen dezentralen Datenbeständen, selbst wenn einzelne Rechnerknoten nicht verfügbar sind.
- Es garantiert die Konsistenz der verteilten Datenbanken und optimiert intern die verteilten Abfragen und Manipulationen mit einem Koordinationsprogramm.[2]

Erste Prototypen verteilter Datenbanksysteme sind Anfang der Achtzigerjahre entstanden. Heute sind relationale Datenbanksysteme verfügbar, die die oben formulierten Anforderungen an ein föderiertes Datenbanksystem teilweise erfüllen. Zudem bleibt das Spannungsfeld zwischen Partitionierungstoleranz und Schemaintegration bestehen, so dass viele verteilte Datenbanken, insbesondere NoSQL-Datenbanken (siehe Kap. 7), entweder keine Schemaföderation anbieten, wie bei den Schlüssel-Wert-, Spaltenfamilien- oder Dokumentdatenbanken, oder aber, wie bei den Graphdatenbanken, die Fragmentierung der Datenbasis nicht unterstützen.

6.3 Temporale Datenbanken

Heutige relationale Datenbanksysteme sind darauf ausgelegt, gegenwartsbezogene (aktuelle) Informationen in Tabellen verwalten zu können. Möchten die Anwendenden ihre relationalen Datenbanken hingegen allgemeiner zeitbezogen abfragen und auswerten, so müssen sie ihre vergangenheits- und zukunftsgerichteten Sachbestände individuell verwalten und nachführen. Das Datenbanksystem unterstützt sie nämlich nicht direkt beim Abspeichern, Suchen oder Auswerten zeitbezogener Informationen.

Unter dem Begriff Zeit wird eine *eindimensionale physikalische Grösse* verstanden, deren Werte total geordnet sind, so dass je zwei Werte auf der Zeitachse durch die Ordnungsrelationen «kleiner als» oder «grösser als» miteinander verglichen werden können. Als Zeitangaben interessieren nicht nur Tag und Zeitpunkt wie «1. April 2016, 14:00 Uhr», sondern auch die Zeitdauer in Form von Zeitintervallen. Ein Beispiel dazu wäre das Alter eines Mitarbeitenden, festgelegt durch eine Anzahl Jahre. Es gilt zu beachten, dass eine Zeitangabe je nach Auffassung der Anwendenden als Zeitpunkt oder als Zeitdauer interpretiert werden kann.

Bei *temporalen Datenbanken* (engl. *temporal databases*) geht es darum, Datenwerte, einzelne Tupel oder ganze Tabellen mit der *Zeitachse in Beziehung* zu setzen. Die Zeitangabe selbst hat für ein bestimmtes Objekt in der Datenbank unterschiedliche Bedeutung, denn unter einer *Gültigkeitszeit* wird entweder die Angabe eines Zeitpunktes, zu dem ein bestimmtes Ereignis stattfindet, oder auch die Angabe eines Zeitintervalls verstanden, falls die entsprechenden Datenwerte während einer Zeitdauer gültig sind. So ist die Adresse eines Mitarbeitenden bis zur nächsten Adressänderung gültig.

[2] Bei verteilten SQL-Ausdrücken garantiert das Zwei-Phasen-Freigabeprotokoll (engl. *two phase commit protocol*) Konsistenz.

MITARBEITER

M#	Name	Geb.datum	Ort	Ein.datum	Funktion
M19	Schweizer	1948-02-19	Frenkendorf	1979-10-01	Sachbearbeiter
M1	Meier	1958-07-09	Liestal	1984-07-01	Analytiker
M7	Huber	1969-03-28	Basel	1988-01-01	Personalchef
M4	Becker	1952-12-06	Liestal	1978-04-15	Revisor

Gesucht sind alle **Mitarbeiter**, die **vor dem zwanzigsten Altersjahr in die Firma eingetreten sind**:

```
SELECT   Name, Bezeichnung
FROM     MITARBEITER, ABTEILUNG
WHERE    Unt = A#
```

M#	Name
M7	Huber

Abb. 6.3 Tabelle MITARBEITER mit Datentyp DATE

Um eine andere Art von Zeitangabe handelt es sich bei der *Aufzeichnungszeit*, die festhält, zu welchem Zeitpunkt ein bestimmtes Objekt in die Datenbank eingefügt, dort verändert oder gelöscht worden ist. Normalerweise verwaltet das Datenbanksystem die verschiedenen Aufzeichnungszeiten mit Hilfe eines Journals in eigener Regie, weshalb in den nun folgenden Ausführungen unter Zeit immer die Gültigkeitszeit verstanden wird.

Um Zeitpunkte der Gültigkeit erfassen zu können, werden von den meisten relationalen Datenbanksystemen bereits heute zwei Datentypen unterstützt: *DATE dient der Angabe eines Datums* in der Form Jahr, Monat und Tag, *TIME der Angabe einer Uhrzeit* durch Stunden, Minuten und Sekunden. Für die Angabe einer Zeitdauer muss kein spezieller Datentyp gewählt werden, ganze Zahlen und Dezimalzahlen genügen. Damit lässt sich auf natürliche Art mit Zeitangaben rechnen. Ein Beispiel ist die in Abb. 6.3 dargestellte Mitarbeitertabelle, deren Merkmalskategorien durch Geburtsdatum und Eintrittsdatum ergänzt sind. Somit sind diese Merkmale zeitbezogen, und vom System kann nun eine Liste aller Mitarbeitenden verlangt werden, die vor dem zwanzigsten Altersjahr in die Firma eingetreten sind.

Die Tabelle MITARBEITER bildet nach wie vor eine Momentaufnahme des aktuellen Datenbankbestandes. Wir können also weder Abfragen in die Vergangenheit noch Abfragen in die Zukunft stellen, da wir keinen Aufschluss über die *Gültigkeitszeit* der einzelnen Datenwerte erhalten. Wird in der Tabelle beispielsweise die Funktion der Mitarbeiterin

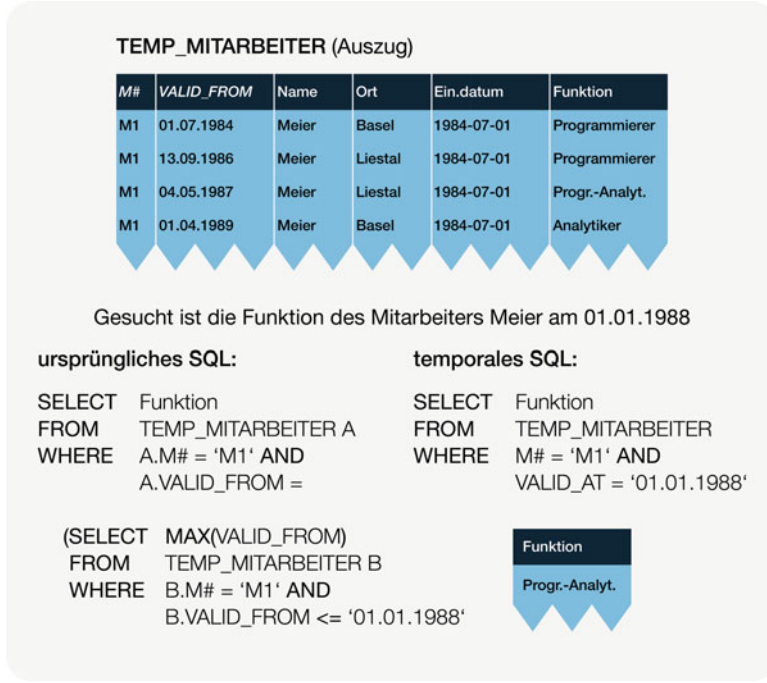

Abb. 6.4 Auszug aus einer temporalen Tabelle TEMP_MITARBEITER

Huber verändert, so überschreiben wir den jetzigen Datenwert und betrachten die neue Funktion als die aktuelle. Hingegen wissen wir nicht, ab wann und bis wann Mitarbeiterin Huber in einer bestimmten Funktion tätig gewesen ist.

Um die *Gültigkeit einer Entität* auszudrücken, werden oft zwei Merkmale verwendet. Der Zeitpunkt «gültig von» (engl. *valid from*) gibt an, ab wann ein Tupel oder ein Datenwert gültig ist. Das Merkmal «gültig bis» (engl. *valid to*) drückt durch den entsprechenden Zeitpunkt das Ende des Gültigkeitsintervalls aus. Anstelle der beiden Zeitpunkte VALID_FROM und VALID_TO kann auf der Zeitachse der Zeitpunkt VALID_FROM bereits genügen. Die Zeitpunkte VALID_TO sind implizit durch die jeweils nächstfolgenden Zeitpunkte VALID_FROM gegeben, da sich die Gültigkeitsintervalle einer bestimmten Entität nicht überlappen können.

Die in Abb. 6.4 dargestellte temporale Tabelle TEMP__MITARBEITER listet für die Mitarbeitertupel M1 (Meier) im Merkmal VALID_FROM sämtliche Gültigkeitsangaben auf. Dieses Merkmal *muss* zum Schlüssel gezählt werden, damit nicht nur die aktuellen Zustände, sondern auch Vergangenheitswerte und Zukunftsangaben eindeutig identifiziert werden können.

Die vier Tupeleinträge lassen sich wie folgt interpretieren: Die Mitarbeiterin Meier wohnte vom 01.07.84 bis zum 12.09.86 in Basel, anschließend bis zum 31.03.89 in Liestal

und ab 01.04.89 wieder in Basel. Seit Eintritt in die Firma bis zum 03.05.87 war sie als Programmiererin tätig, vom 04.05.87 bis zum 31.03.89 als Programmierer-Analytikerin, seit 01.04.89 ist sie Analytikerin. Die Tabelle TEMP_MITARBEITER ist in der Tat temporal, da sie neben aktuellen Zuständen auch Angaben über vergangenheitsbezogene Datenwerte aufzeigt. Insbesondere kann sie Abfragen beantworten, die nicht nur aktuelle Zeitpunkte oder Zeitintervalle betreffen.

Als Beispiel interessieren wir uns in Abb. 6.4 für die Funktion der Mitarbeiterin Meier am 1. Januar 1988. Anstelle des ursprünglichen SQL-Ausdrucks einer geschachtelten Abfrage mit der Funktion MAX (vgl. Abschn. 3.3.1 resp. Tutorium in SQL) könnte man sich eine Sprache vorstellen, die temporale Abfragen direkt unterstützt. Mit dem Schlüsselwort VALID_AT wird ein Zeitpunkt festgelegt, zu dem alle gültigen Einträge gesucht werden sollen.

Temporales Datenbanksystem
Ein temporales Datenbankmanagementsystem

- unterstützt als Gültigkeitszeit die Zeitachse, indem es Merkmalswerte oder Tupel nach der Zeit ordnet und
- umfasst temporale Sprachelemente für Abfragen in die Zukunft, Gegenwart und Vergangenheit.

Auf dem Gebiet der *temporalen Datenbanken* liegen verschiedene Sprachmodelle vor, die das Arbeiten mit zeitbezogenen Informationen vereinfachen. Speziell müssen die Operatoren der Relationenalgebra bzw. des Relationenkalküls erweitert werden, um beispielsweise einen Verbund von temporalen Tabellen zu ermöglichen. Auch die Regeln der referenziellen Integrität müssen angepasst und zeitbezogen ausgelegt werden. Obwohl sich solche Verfahren und entsprechende Spracherweiterungen als Prototypen in Forschungs- und Entwicklungslabors bereits bewährt haben, unterstützen heute die wenigsten Datenbanksysteme temporale Konzepte. Der SQL-Standard sieht ebenfalls die Unterstützung von temporalen Datenbanken vor.

6.4 Multidimensionale Datenbanken

Bei operativen Datenbanken und Anwendungen konzentriert man sich auf einen klar definierten, funktionsorientierten Leistungsbereich. Transaktionen bezwecken, Daten für die Geschäftsabwicklung schnell und präzise bereitzustellen. Diese Art der Geschäftstätigkeit wird oft als *Online Transaction Processing* oder OLTP bezeichnet. Da die operativen Datenbestände täglich neu überschrieben werden, gehen für die Anwendenden wichtige Entscheidungsgrundlagen verloren. Zudem sind diese Datenbanken primär für das laufende Geschäft und nicht für Analyse und Auswertung konzipiert worden.

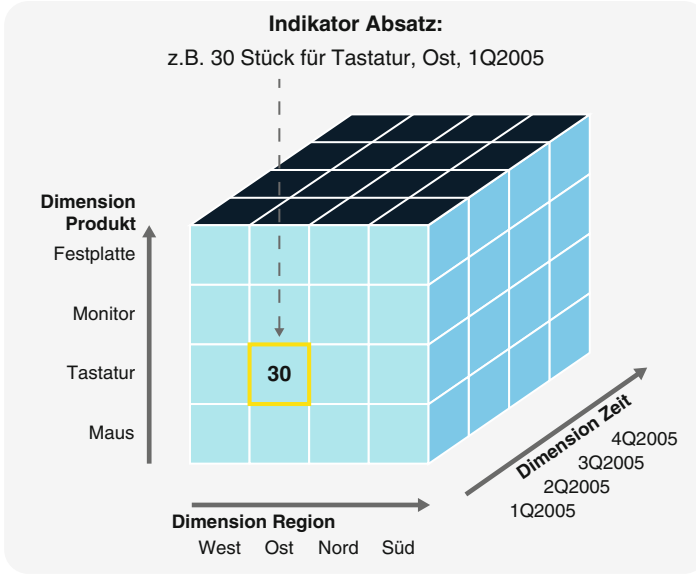

Abb. 6.5 Mehrdimensionaler Würfel mit unterschiedlichen Auswertungsdimensionen

Aus diesen Gründen werden seit einigen Jahren neben transaktionsorientierten Datenbeständen auch eigenständige Datenbanken und Anwendungen entwickelt, die der Datenanalyse und der Entscheidungsunterstützung dienen. Man spricht in diesem Zusammenhang von *Online Analytical Processing* oder OLAP.

Kernstück von OLAP ist eine *multidimensionale Datenbank* (engl. *multi-dimensional database*), in der alle entscheidungsrelevanten Sachverhalte nach beliebigen Auswertungsdimensionen abgelegt werden (mehrdimensionaler Datenwürfel oder Data Cube). Eine solche Datenbank kann recht umfangreich werden, da sie Entscheidungsgrößen zu unterschiedlichen Zeitpunkten enthält. Beispielsweise können in einer multidimensionalen Datenbank Absatzzahlen quartalsweise, nach Verkaufsregionen und Produkten abgelegt und ausgewertet werden.

Betrachten wir dazu ein Beispiel in Abb. 6.5, das zugleich den Entwurf einer multidimensionalen Datenbank illustrieren soll. In diesem Beispiel interessieren uns die drei Auswertungsdimensionen Produkt, Region und Zeit. Der Begriff *Dimension* (engl. *dimension*) beschreibt die Achsen des mehrdimensionalen Würfels. Der Entwurf dieser Dimensionen ist bedeutend, werden doch entlang dieser Achsen Analysen und Auswertungen vorgenommen. Die Reihenfolge der Dimensionen spielt keine Rolle, jeder Anwendende kann und soll aus unterschiedlichen Blickwinkeln seine oder ihre gewünschten Auswertungen vornehmen können. Beispielsweise priorisieren Produktemanager die Produktedimension oder Verkäufer möchten Verkaufszahlen nach ihrer Region auflisten.

Die Dimensionen selber können weiter strukturiert sein: Die Dimension Produkt kann Produktegruppen enthalten; die Dimension Zeit könnte neben Quartalsangaben auch

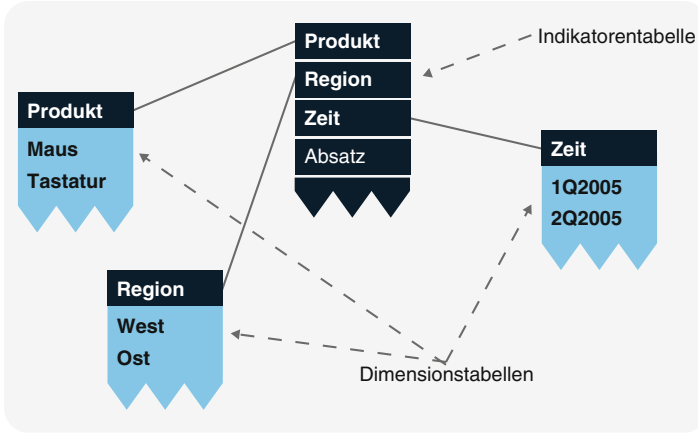

Abb. 6.6 Sternschema für eine mehrdimensionale Datenbank

Tage, Wochen und Monate pro Jahr abdecken. Eine Dimension beschreibt somit auch die gewünschten Aggregationsstufen, die für die Auswertung des mehrdimensionalen Würfels gelten.

Aus logischer Sicht müssen bei einer mehrdimensionalen Datenbank resp. beim Datenwürfel neben den Dimensionen auch die Indikatoren[3] spezifiziert werden. Unter einem *Indikator* (engl. *indicator*) versteht man eine Kennzahl resp. Kenngröße, die für die Entscheidungsunterstützung gebraucht wird. Diese Kennzahlen werden durch Auswertung aggregiert und anhand der Dimensionswerte gruppiert. Indikatoren können quantitative wie qualitative Eigenschaften der Geschäftstätigkeit betreffen. Neben finanziellen Kenngrössen sind Indikatoren über Markt und Absatz, Kundenstamm und Kundenbewegung, Geschäftsprozesse, Innovationspotenzial oder Know-how der Belegschaft von Bedeutung. Die Indikatoren bilden neben den Dimensionen die Grundlage für die Entscheidungsunterstützung des Managements, für interne und externe Berichterstattung sowie für ein rechnergestütztes Performance-Measurement System.

Hauptmerkmal eines *Sternschemas* (engl. *star schema*) ist die Klassifikation der Daten in die zwei Gruppen Indikatorendaten und Dimensionsdaten. Beide Gruppen werden in Abb. 6.6 tabellarisch illustriert. Die Indikatorentabelle steht im Zentrum, um welches die deskriptiven Dimensionstabellen angesiedelt sind; pro Dimension je eine Tabelle. Die Dimensionstabellen hängen also sternartig an der Indikatorentabelle.

Falls einzelne Dimensionen strukturiert sind, können an der entsprechenden Dimensionstabelle weitere Dimensionstabellen angehängt werden. Dadurch entsteht ein *Schneeflocken-Schema* (engl. *snowflake schema*), das Aggregationsstufen einzelner Dimensionen zeigt. Beispielsweise könnte in der Abb. 6.6 an der Dimensionstabelle Zeit für das erste

[3] In der Literatur werden die Indikatoren oft als Fakten bezeichnet, vgl. dazu auch Abschn. 6.7 über Fakten und Regeln wissensbasierter Datenbanken.

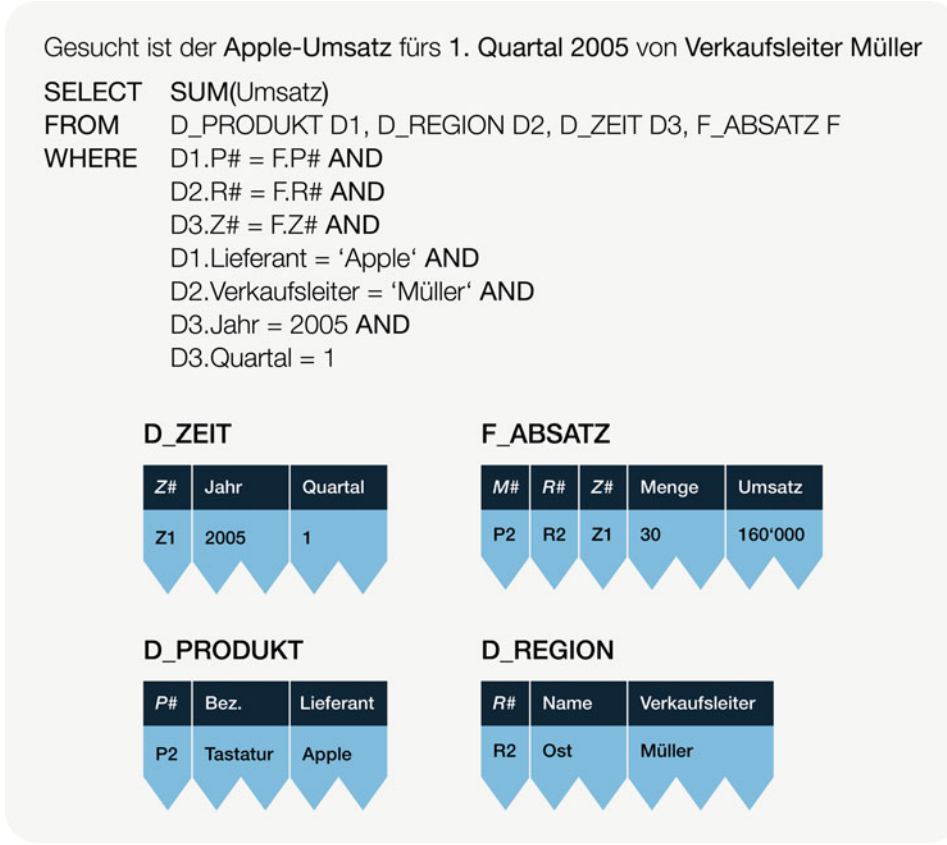

Abb. 6.7 Implementierung eines Sternschemas mit dem Relationenmodell

Quartal 2005 eine weitere Dimensionstabelle angeschlossen werden, die die Tage der Monate Januar bis März 2005 aufzählt. Falls die Dimension Monat ebenfalls für Analysezwecke gebraucht wird, würde man eine weitere Dimensionstabelle Monat definieren und mit der Dimensionstabelle Tag verbinden.

Für die Implementierung einer mehrdimensionalen Datenbank kann das klassische Relationenmodell verwendet werden. Abbildung 6.7 zeigt, wie die Indikatoren- und Dimensionstabellen des Sternschemas umgesetzt werden. Die Indikatorentabelle wird mit der Relation F_ABSATZ dargestellt, wobei ein mehrdimensionaler Schlüssel resultiert. Dieser zusammengesetzte Schlüssel muss als Komponenten die jeweiligen Schlüssel der Dimensionstabellen von D_PRODUKT, D_REGION und D_ZEIT enthalten. Möchte man nun den Umsatz des Verkaufsleiters Müller für das erste Quartal 2005 und für die Apple-Geräte ermitteln, so muss ein aufwendiger Verbund aller beteiligter Indikatoren- und Dimensionstabellen formuliert werden (siehe SQL-Statement in Abb. 6.7).

Die Leistungsfähigkeit eines relationalen Datenbanksystems stößt bei umfangreichen multidimensionalen Datenbanken an Grenzen. Auch das Formulieren von Abfragen ist mit dem Sternschema aufwendig und fehleranfällig. Darüber hinaus gibt es eine weitere Anzahl von Nachteilen, falls mit dem klassischen Relationenmodell und dem herkömmlichen SQL gearbeitet wird: Für die Bildung von Aggregationsstufen muss das Sternschema zu einem Schneeflocken-Schema ausgebaut werden, die entsprechenden physischen Tabellen verschlechtern das Antwortzeitverhalten zusätzlich. Möchten die Anwendenden einer multidimensionalen Datenbank zu Analysezwecken Details nachfragen (sog. Drill-down) oder weitere Aggregationsstufen auswerten (sog. Roll-up), so hilft ihnen das klassische SQL nicht weiter. Auch das Herausschneiden oder Rotieren einzelner Teile aus dem mehrdimensionalen Würfel, was zu Analysezwecken gebräuchlich ist, muss mit spezifischer Software oder sogar Hardware erkauft werden. Wegen dieser Mängel haben sich einige Hersteller von Datenbankprodukten entschlossen, ihre Softwarepalette mit geeigneten Werkzeugen anzureichern. Zudem ist auf der Sprachebene der SQL-Standard erweitert worden, um Würfeloperationen inkl. Aggregationen einfach formulieren zu können.

Multidimensionales Datenbanksystem
Ein multidimensionales Datenbankmanagementsystem unterstützt einen mehrdimensionalen Datenwürfel mit folgender Funktionalität:

- Für den Entwurf können unterschiedliche Dimensionstabellen mit beliebigen *Aggregationsstufen* definiert werden, insbesondere auch für die Dimension Zeit.
- Die Auswertungs- und Analysesprache stellt Funktionen für Drill-down und Roll-up zur Verfügung.
- Das Auswählen einer Scheibe aus dem mehrdimensionalen Würfel (Slicing) und ein Wechsel der Dimensionsreihenfolge (Dicing, Rotation) werden unterstützt.

Multidimensionale Datenbanken bilden oft das Kernstück im Gesamtsystem eines betrieblichen Datenlagers (engl. *data warehouse*). Im Unterschied zur multidimensionalen Datenbank an und für sich vereinigt das Data Warehouse als verteiltes Informationssystem Mechanismen zur Integration, Historisierung und Auswertung von Daten über viele Applikationen des Betriebs, zusammen mit Prozessen zur Entscheidungsunterstützung und zur Verwaltung und Weiterentwicklung des Datenflusses innerhalb der Organisation.

6.5 Das Data Warehouse

Mit der Verfügbarkeit von digitalen Daten steigt das Bedürfnis nach Auswertung dieser Daten zur Entscheidungsunterstützung. So soll das Management eines Unternehmens Entscheidungen auf Fakten basieren, welche durch Analyse der vorhandenen Daten

hergestellt werden können. Dieser Prozess nennt man *Business Intelligence*. Aufgrund der *Heterogenität, Flüchtigkeit und Fragmentierung* der Datenbasis ist die applikationsübergreifende Datenanalyse oft aufwendig: Daten in Organisationen sind heterogen in vielen verschiedenen Datenbanken gespeichert. Zudem sind sie oft nur in der aktuellen Version verfügbar. In den Quellsystemen sind Daten zu einem übergeordneten Themenbereich wie Kunden oder Verträge selten an einem Ort zu finden, sondern müssen über verschiedene Schnittstellen zusammengefasst, also integriert werden. Mit diesen Daten, die auf vielen Datenbanken verteilt sind, sollen zudem zu einem bestimmtem Themenbereich Zeitreihen über mehrere Jahre gebildet werden können. Business Intelligence stellt also drei Anforderungen an die auszuwertenden Daten:

- Integration von heterogenen Datenbeständen.
- Historisierung von aktuellen und flüchtigen Datenbeständen.
- Vollständige Verfügbarkeit von Daten zu bestimmten Themenbereichen.

Bisher wurden drei postrelationale Datenbanksysteme vorgestellt, welche im Prinzip je eine dieser drei Anforderungen abdecken würden: Die Integration von Daten kann mit *föderierten* Datenbanksystemen mit zentralem logischem Schema erreicht werden, die Historisierung von Daten wird mit *temporalen* Datenbanken durchgeführt und schliesslich dienen *multidimensionale* Datenbanken der auswertungsorientierten Bereitstellung von Daten zu bestimmten Themenbereichen (Dimensionen).

Da sich in der Praxis die relationale Datenbanktechnologie so breit durchgesetzt hat, können die Eigenschaften verteilter, temporaler und multidimensionaler Datenbanken mit herkömmlichen relationalen Datenbanken und einigen softwaretechnischen Erweiterungen gut simuliert werden. Das Konzept des *Data Warehousing* setzt diese Aspekte von föderierten, temporalen und multidimensionalen Datenbanksystemen mit herkömmlichen Technologien um.

Zu diesen drei Aspekten kommt aber noch die Anforderung der *Entscheidungsunterstützung* hinzu. In Organisationen besteht das Bedürfnis nach Auswertung von Daten als Zeitreihen, so dass zu Themenbereichen vollständige Datenbestände an einem Ort verfügbar sind. Da aber die Daten in größeren Organisationen über eine Vielzahl von einzelnen Datenbanken verstreut sind, braucht es ein Konzept,[4] welches die verstreuten Daten für die Analyse und Nutzung aufbereitet.

Data Warehouse

Das Data Warehouse oder DWH ist ein verteiltes Informationssystem mit folgenden Eigenschaften:

[4] Vgl. den KDD-Prozess für das Knowledge Discovery in Databases.

- **Integriert**: Daten aus verschiedenen Datenquellen und Applikationen (Quellsystemen) werden periodisch zusammengefasst[5] und in einem einheitlichen Schema abgelegt.
- **Read only**: Daten im Data Warehouse werden, sobald sie festgeschrieben sind, nicht mehr verändert.
- **Historisiert**: Daten können dank einer Zeitachse nach verschiedenen Zeitpunkten ausgewertet werden.
- **Auswertungsorientiert**: Alle Daten zu verschiedenen Themenbereichen (Subject Areas) wie Kunden, Verträge und Produkte sind an einem Ort vollständig verfügbar.
- **Entscheidungsunterstützend**: Die Fakten in mehrdimensionalen Datenwürfeln bilden die Grundlage von Management-Entscheiden.

Ein Data Warehouse bietet teilweise Funktionen von föderierten, temporalen und multidimensionalen Datenbanken an. Darüber hinaus existieren programmierbare Ladeprozesse (Skripts) sowie spezifische Auswertungs- und Aggregationsfunktionen. Ausgehend von verteilten und heterogenen Datenquellen sollen die geschäftsrelevanten Fakten in der Form vorliegen, dass sie effizient und effektiv für Entscheidungsunterstützung und Steuerung der Geschäftstätigkeiten genutzt werden können.

Im Data Warehouse lassen sich unterschiedliche interne und externe Datenbestände (sogenannte Datenquellen) integrieren. Das Ziel bleibt, einen einheitlichen konsistenten und historisierten Datenbestand über alle in der Unternehmung verstreuten Daten für unterschiedliche betriebswirtschaftliche Sichten zu erhalten und auswerten zu können. Dazu werden Daten von vielen Quellen über Schnittstellen ins Data Warehouse integriert und oft über Jahre aufbewahrt. Darauf aufbauend können Datenanalysen erstellt werden, welche den Entscheidungsträgern präsentiert und in den Geschäftsabläufen genutzt werden können. Zudem muss Business Intelligence als Prozess im Management gesteuert werden.

Im Folgenden werden die einzelnen Schritte des Data Warehousing kurz beschrieben (vgl. Abb. 6.8).

Die *Daten* einer Organisation sind auf verschiedene Quellsysteme verteilt wie die Webplattform, die Buchhaltung (Enterprise Resource Planning, ERP) oder die Kunden-Datenbank (Customer Relationship Management, CRM). Damit diese Daten für die Auswertung genutzt und miteinander in Verbindung gebracht werden können, müssen sie zusammengezogen, also integriert, werden

Für die *Integration* dieser Daten sind Extraktions-, Transformations- und Ladeschritte notwendig (ETL-Prozess mit Extract, Transform und Load). Diese *Schnittstellen* übermitteln Daten meistens am Abend oder über das Wochenende, wenn die IT-Systeme nicht von Benutzern verwendet werden. Heute werden bei hochperformanten Systemen kontinuierliche Ladeprozesse eingesetzt, bei denen die Daten rund um die Uhr (7/24-Stundenbetrieb) nachgeführt werden (engl. *trickle feed*). Für die Nachführung eines Data Warehouse wird

[5] vgl. den ETL-Prozess für Extract, Transform und Load weiter unten.

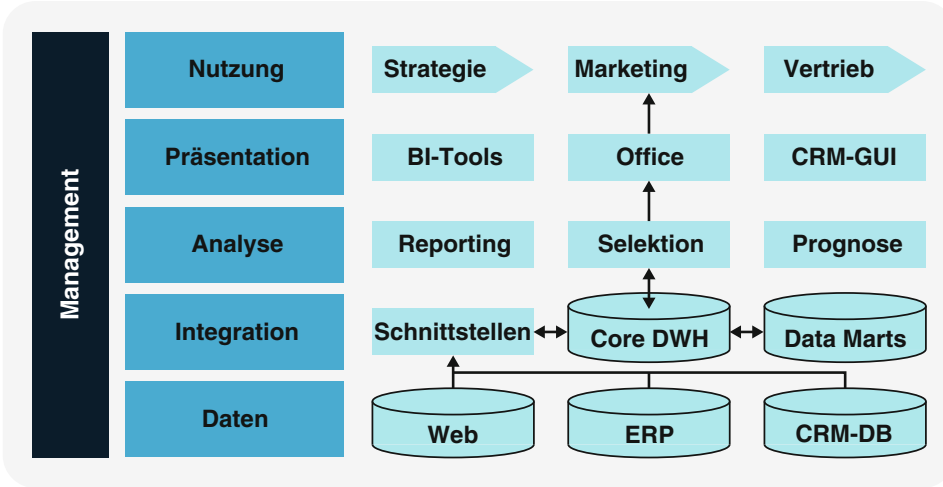

Abb. 6.8 Das Data Warehouse im Zusammenhang mit Business-Intelligence-Prozessen

die *Periodizität* berücksichtigt, damit die Anwendenden über die *Aktualität* ihrer Auswertungsdaten Bescheid haben; d. h. je häufiger Daten über die Schnittstellen ins Data Warehouse geladen werden, desto aktueller sind die Auswertungsdaten. Ziel dieses Datenzusammenzugs ist die *Historisierung*, also der Aufbau einer Zeitreihe an einem logisch zentralen Speicherort. Der Kern des Data Warehouse (Core DWH) ist häufig in zweiter oder dritter Normalform modelliert. Die Historisierung erfolgt wie im Abschn. 6.3 zu temporalen Datenbanken beschrieben mit der Angabe von Gültigkeitszeiträumen (valid_from, valid_to) in zusätzlichen Spalten in den Tabellen. Um die Auswertungsdaten themenorientiert für die OLAP-Analyse zur Verfügung zu stellen, werden einzelne Themenbereiche in sogenannte *Data Marts* geladen, welche oft multidimensional mit Stern-Schemas realisiert werden.

Das Data Warehouse dient ausschließlich der *Analyse von Daten*. Die Dimension Zeit ist fester Bestandteil einer solchen Datensammlung, wodurch statistische Auswertungen an Aussagekraft gewinnen. Periodische Berichte (Reporting) produzieren Listen mit relevanten Informationen (Key Performance Indicators). Werkzeuge des sog. *Data Mining* wie Klassifikation, Selektion, Prognose und Wissensakquisition verwenden die Daten aus dem Data Warehouse, um beispielsweise das Kunden- und Kaufverhalten zu analysieren und für Optimierungszwecke zu nutzen.

Damit die Daten einen Wert generieren, müssen die gewonnenen Erkenntnisse samt den Resultaten der Analysen den Entscheidungsträgern und Anspruchsgruppen vermittelt werden. Mit verschiedenen Oberflächen von Business-Intelligence-Werkzeugen (BI-Tools) oder mit grafischen Benutzerschnittstellen (GUI = Graphical User Interface)

für Büroautomation oder Customer Relationship Management werden Auswertungen oder Grafiken dazu verfügbar gemacht. Die Entscheidungsträger können die Resultate der Analysen aus dem Data Warehouse dann für die *Nutzung in den Geschäftsabläufen* sowie für Strategie, Marketing und Vertrieb anwenden.

6.6 Objektrelationale Datenbanken

Wollen wir in eine relationale Datenbank Informationen über Bücher ablegen, so müssen wir mehrere Tabellen definieren, drei davon sind in Abb. 6.9 dargestellt: In der Tabelle BUCH versehen wir ein Buch mit dem Titel und dem Verlag.

Da an einem bestimmten Buch mehrere Autoren beteiligt sein können und umgekehrt ein Autor mehrere Bücher schreiben kann, reihen wir jede einzelne Urheberschaft an einem Buch in einer zusätzlichen Tabelle AUTOR auf.

Das Merkmal «Name» ist nicht voll funktional vom zusammengesetzten Schlüssel der Autoren- und Buchnummer abhängig, weshalb sich die Tabelle weder in zweiter noch in höherer Normalform befindet. Dasselbe gilt für die Tabelle SCHLAGWORT, da zwischen den Büchern und ihren Schlagwörtern ebenfalls eine komplex-komplexe Beziehung besteht. Die «Gewichtung» ist zwar ein typisches Beziehungsmerkmal, hingegen ist die «Bezeichnung» nicht voll funktional vom Schlüssel der Schlagwort- und Buchnummer abhängig. Für die Verwaltung von Büchern würden somit aus Gründen der Normalisierung mehrere Tabellen resultieren, da zusätzlich zu den Beziehungstabellen AUTOR und SCHLAGWORT eigenständige Tabellen für die Autoren- und Schlagwortmerkmale hinzukommen müssten. Sicher würden in einer relationalen Datenbank auch die Verlagsangaben in einer eigenständigen Tabelle VERLAG aufgenommen, idealerweise ergänzt mit einer Beziehungsrelation zwischen BUCH und VERLAG.

Es ist von Nachteil und aus Sicht der Anwendenden kaum verständlich, Buchinformationen auf mehrere Tabellen zu zerstreuen, möchten doch die Benutzer die Eigenschaften eines bestimmten Buches strukturiert in einer einzigen Tabelle aufgelistet haben. Die relationale Abfrage- und Manipulationssprache wäre eigentlich dazu da, durch einfache Operatoren unterschiedliche Buchinformationen zu verwalten. Auch hinsichtlich der Leistung (Performance) ergeben sich Nachteile, wenn das Datenbanksystem mehrere Tabellen durchsuchen und zeitaufwendige Verbundoperatoren berechnen muss, um ein bestimmtes Buch aufzufinden. Wegen solcher Nachteile sind für das Relationenmodell Erweiterungen vorgeschlagen worden.

Eine erste Erweiterung relationaler Datenbanktechnologie besteht darin, dem *Datenbanksystem Struktureigenschaften explizit bekanntzugegeben*. Dies kann durch die Vergabe sogenannter *Surrogate* geschehen. Ein Surrogat ist ein *vom System definierter, fixer und unveränderbarer (invarianter) Schlüsselwert*, der jedes Tupel in der Datenbank

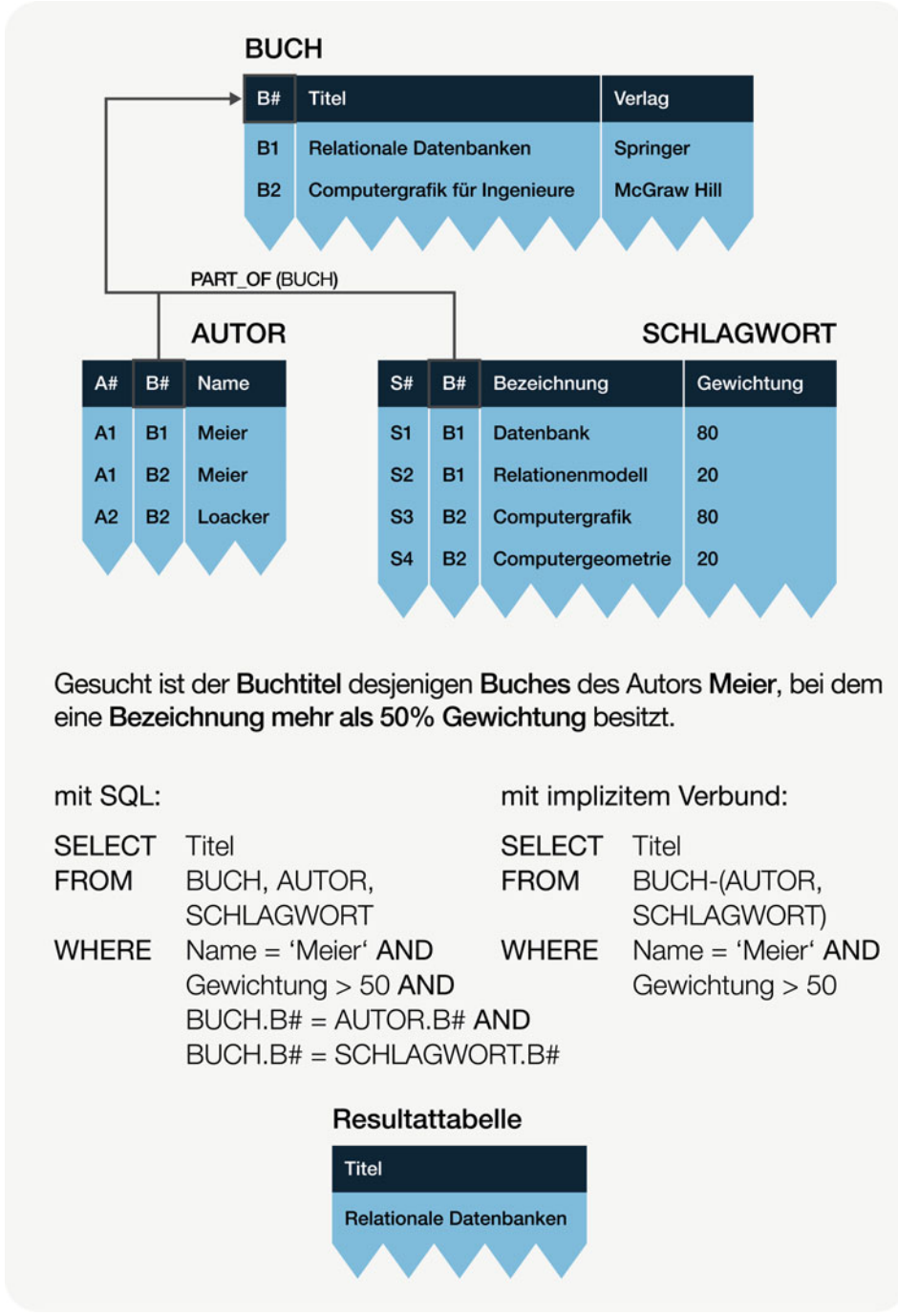

Abb. 6.9 Abfrage eines strukturierten Objektes mit und ohne impliziten Verbundoperator

eindeutig identifiziert. Surrogate können als invariante Werte zur Definition systemkontrollierter Beziehungen benutzt werden, und zwar an verschiedenen Stellen innerhalb einer relationalen Datenbank. Solche Surrogate unterstützen die referenzielle Integrität, aber auch Generalisations- und Aggregationsstrukturen.

Kehren wir zurück zur Tabelle BUCH und zur Abb. 6.9. Dort sei die Buchnummer B# als Surrogat definiert. Zusätzlich wird diese Nummer in den beiden abhängigen Tabellen AUTOR und SCHLAGWORT unter dem Hinweis PART_OF(BUCH) nochmals verwendet (vgl. Regel 7 zur Aggregation in Abschn. 2.3.2). Diese Referenz sorgt dafür, dass das Datenbanksystem die Struktureigenschaft der Bücher-, Autoren- und Schlagwortangaben explizit kennt und diese bei Datenbankabfragen ausnutzen kann, sofern die relationale Abfrage- und Manipulationssprache entsprechend erweitert wird. Als Beispiel diene der implizite hierarchische Verbundoperator, der in der FROM-Klausel steht und die zur Tabelle BUCH gehörenden Teiltabellen AUTOR und SCHLAGWORT miteinander verknüpft. Die Verbundprädikate in der WHERE-Klausel anzugeben erübrigt sich, da diese mit der expliziten Definition der PART_OF-Struktur dem Datenbanksystem bereits bekannt sind.

Wird dem Datenbanksystem eine PART_OF- oder auf analoge Art eine IS_A-Struktur (vgl. Abschn. 2.2.3) mitgeteilt, so lassen sich auch die Speicherstrukturen effizienter implementieren. Beispielsweise wird zwar die logische Sicht der drei Tabellen BUCH, AUTOR und SCHLAGWORT beibehalten, physisch hingegen werden die Buchinformationen als strukturierte Objekte[6] abgespeichert, so dass ein einziger Datenbankzugriff das Auffinden eines Buches bewerkstelligt. Die klassische Sicht der Tabellen bleibt erhalten und einzelne Tabellen der Aggregation können wie bisher abgefragt werden.

Eine andere Möglichkeit zur Verwaltung strukturierter Informationen liegt darin, die *erste Normalform aufzugeben*[7] und als Merkmale selbst Tabellen zuzulassen. Ein Beispiel dazu ist in Abb. 6.10 illustriert, wo Informationen über Bücher, Autoren und Schlagwörter in einer Tabelle untergebracht sind. Auch dieses zeigt einen objektrelationalen Ansatz, da ein Buch als ein Objekt in einer einzigen Tabelle BUCHOBJEKT verwaltet wird. Ein *objektrelationales Datenbanksystem* (engl. *object-relational database system*) kann Struktureigenschaften explizit aufnehmen und Operatoren für Objekte und Teilobjekte anbieten.

Unterstützt ein Datenbanksystem strukturierte Objekttypen wie in der Darstellung Abb. 6.10, so ist es *strukturell objektrelational*. Zusätzlich zur Objektidentifikation, zur Strukturbeschreibung und zum Angebot generischer Operatoren (Methoden wie impliziter Verbund etc.) sollte ein vollständig objektrelationales Datenbanksystem den Benutzern auch das Definieren neuer Objekttypen (Klassen) und Methoden anbieten. Die Anwendenden sollten dabei die für einen individuellen Objekttyp notwendigen Methoden selbst festlegen können. Auch sollten sie auf tatkräftige Unterstützung von

[6] Die Forschungsliteratur nennt diese auch «komplexe Objekte».

[7] Das sogenannte NF^2-Modell (NF^2 = Non First Normal Form) erlaubt geschachtelte Tabellen.

BUCHOBJEKT

			Autor		Schlagwort		
B#	Titel	Verlag	A#	Name	S#	Bezeichnung	Gew.
B1	Relationale...	Springer	A1	Meier	S1	Datenbank	80
					S2	Rel.modell	20
B2	Computer...	McGraw Hill	A1	Meier	S3	Comp.grafik	50
			A2	Loacker	S4	C.geometrie	50

Abb. 6.10 Tabelle BUCHOBJEKT mit Merkmalen vom Typ Relation

«Vererbungseigenschaften» zählen können, damit sie nicht alle Objekttypen und Methoden von Grund auf neu zu definieren brauchen, sondern auf bereits bestehende Konzepte zurückgreifen können.

Objektrelationale Datenbanksysteme machen es möglich, strukturierte Objekte als Einheiten zu behandeln und entsprechende generische Operatoren auf diese Objekte anzuwenden. Klassenbildung durch PART_OF- und IS_A-Strukturen ist erlaubt und wird durch Methoden zum Speichern, Abfragen und Manipulieren unterstützt.

Objektrelationales Datenbanksystem
Ein objektrelationales Datenbankmanagementsystem (ORDBMS) lässt sich wie folgt umschreiben:

- Es erlaubt die *Definition von Objekttypen* (in Anlehnung an die objektorientierte Programmierung oft Klassen genannt), die ihrerseits aus weiteren Objekttypen zusammengesetzt sein können.
- Jedes Datenbankobjekt kann aufgrund von Surrogaten strukturiert und identifiziert werden.
- Es unterstützt *generische Operatoren* (Methoden), die auf Objekte oder Teilobjekte wirken. Dabei bleibt die interne Darstellung der Objekte nach aussen unsichtbar (Datenkapselung).
- Eigenschaften von Objekten lassen sich vererben. Diese *Vererbungseigenschaft* bezieht sich sowohl auf die Struktur als auch auf die zugehörigen Operatoren.

Der SQL-Standard unterstützt seit einigen Jahren gewisse objektrelationale Erweiterungen: Objektidentifikationen (Surrogate), vordefinierte Datentypen für Menge, Liste

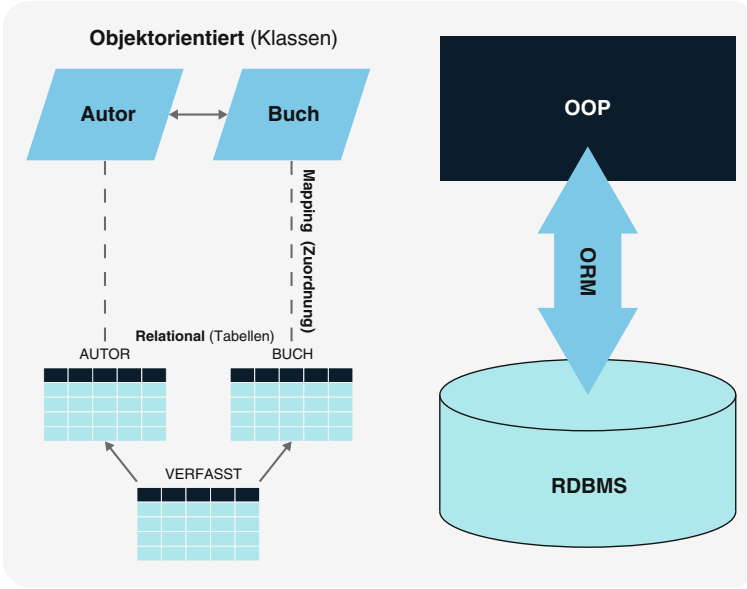

Abb. 6.11 Objektrelationales Mapping

und Feld, allgemeine abstrakte Datentypen mit Kapselungsmöglichkeiten, parametrisierbare Typen, Typ- und Tabellenhierarchien mit Mehrfachvererbung sowie benutzerdefinierte Funktionen (Methoden).

Die meisten modernen Programmiersprachen sind objektorientiert; gleichzeitig ist die Mehrheit der Datenbanksysteme, welche im Einsatz sind, relational. Anstatt auf objektrelationale oder gar auf objektorientierte Datenbanken zu migrieren, was mit grossen Kosten verbunden wäre, kann bei der Software-Entwicklung eine Zuordnung (Mapping) von Objekten zu Relationen hergestellt werden, wenn mit objektorientierten Sprachen auf relationale Daten zugegriffen wird. Dieses Konzept des *objektrelationalen Mapping (ORM)* wird in Abb. 6.11 dargestellt. In diesem Beispiel gibt es ein relationales Datenbankmanagementsystem (RDBMS) mit einer Tabelle AUTOR, einer Tabelle BUCH, und einer Beziehungstabelle VERFASST, da zwischen Büchern und Autoren eine komplex-komplexe Beziehung (Abschn. 2.2.2) besteht. In einem Projekt mit objektorientierter Programmierung (OOP) möchte man die Daten in diesen Tabellen direkt als Klassen in der Softwareentwicklung verwenden.

Eine ORM-Software kann ein Mapping von Klassen zu Tabellen automatisch herstellen, so dass es für die Entwickler aussieht, als ob sie mit objektorientierten Klassen arbeiten, obwohl die Daten im Hintergrund in Datenbanktabellen gespeichert werden. Die Programmierobjekte, die sich im Arbeitsspeicher befinden, werden somit persistent, also auf einen permanenten Speicher, festgeschrieben.

In Abb. 6.11 werden von der ORM-Software für die beiden Tabellen AUTOR und BUCH zwei Klassen Autor und Buch zur Verfügung gestellt. Für jede Zeile in einer Tabelle existiert ein Objekt als Instanz der entsprechenden Klasse. Die Beziehungstabelle VERFASST wird nicht als Klasse abgebildet: Die Objektorientierung erlaubt die Verwendung von nicht-atomaren Objektreferenzen; somit wird im Objekt Autor in einem Vektor-Feld `bücher[]` die Menge der Bücher gespeichert, die der Autor verfasst hat, und im Objekt Buch wird im Feld `autoren[]` die Gruppe der Autoren abgebildet, welche dieses Buch geschrieben haben.

Die Verwendung von ORM ist einfach zu bewerkstelligen. Die ORM-Software kann die entsprechenden Klassen automatisch aufgrund von bestehenden Datenbanktabellen herleiten. Anschließend können Datensätze aus diesen Tabellen als Objekte in der Softwareentwicklung verwendet werden. Somit ist ORM ein möglicher Weg zur Objektorientierung, bei der die zugrunde liegende relationale Datenbanktechnologie beibehalten werden kann.

6.7 Wissensbasierte Datenbanken

Wissensbasierte Datenbanken (engl. *knowledge databases* oder *deductive databases*) können nicht nur eigentliche Datenvorkommen – *Fakten* genannt – sondern auch *Regeln* verwalten, mit deren Hilfe neue Tabelleninhalte oder Fakten hergeleitet werden.

Die Tabelle MITARBEITER ist in Abb. 6.12 der Einfachheit halber auf die Namen der Mitarbeitenden reduziert. Zu dieser Tabelle lassen sich nun Fakten oder Aussagen definieren, in diesem Fall über Mitarbeitende. Allgemein bezeichnet man diejenigen Aussagen als Fakten, die *bedingungslos den Wahrheitsgehalt TRUE annehmen*. So ist es Tatsache, dass Frau Huber eine Mitarbeiterin ist. Dieser Sachverhalt wird durch den Fakt «ist_mitarbeiter (Huber)» ausgedrückt. Für die direkten Vorgesetzten der Mitarbeitenden kann eine neue Tabelle CHEF errichtet werden, die den Namen der direkten Vorgesetzten und der ihnen unterstellten Mitarbeitenden paarweise pro Tupel aufzeigt. Entsprechend werden Fakten «ist_chef_von (A,B)» formuliert, um die Tatsache «A ist direkter Chef von B» auszudrücken.

Die Vorgesetztenhierarchie spiegelt sich anschaulich im Baum von Abb. 6.13. Auf die Frage, wer der direkte Chef der Mitarbeiterin Meier sei, wertet die SQL-Abfrage die Tabelle CHEF aus und stößt auf die Vorgesetzte Huber. Dasselbe Resultat zeigt die Auswertung mit Hilfe einer logischen Programmiersprache (angelehnt an Prolog).

Neben den eigentlichen Fakten können *Regeln zur Herleitung unbekannter Tabelleninhalte* definiert werden. Beim Relationenmodell spricht man in diesem Fall von abgeleiteten Tabellen (engl. *derived relation* oder *deduced relation*). Ein einfaches Beispiel einer abgeleiteten Tabelle und einer entsprechenden Ableitungsregel zeigt Abb. 6.14. Daraus geht hervor, wie für jeden Mitarbeitenden der oder die übernächste Vorgesetzte herausgefunden werden kann. Dies ist beispielsweise für Grossfirmen oder Firmen mit entfernten Niederlassungen interessant, falls der direkte Vorgesetzte des Mitarbeitenden abwesend ist und via E-Mail der nächsthöhere Vorgesetzte angesprochen werden soll.

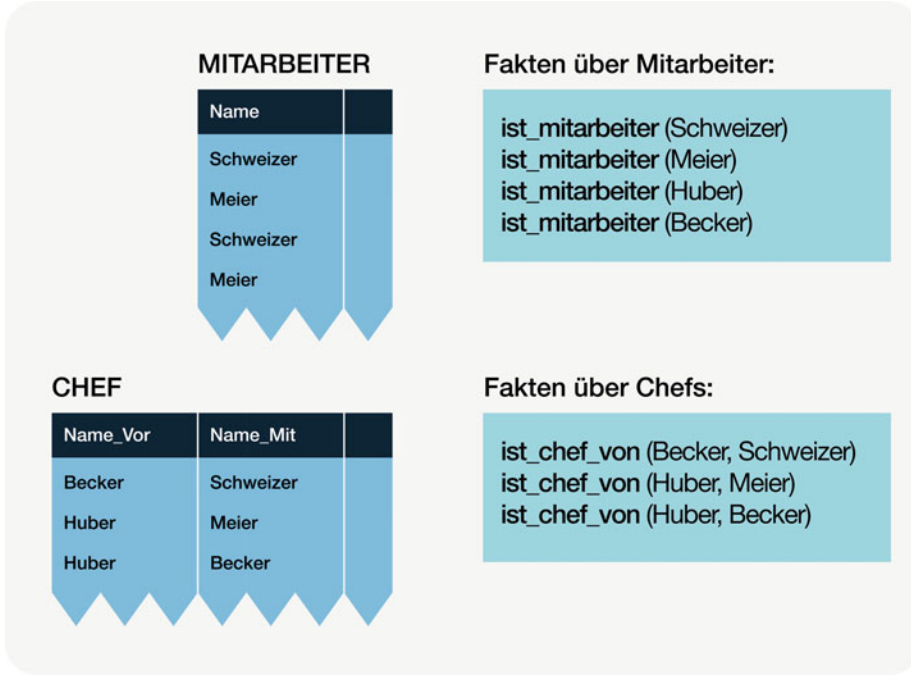

Abb. 6.12 Gegenüberstellung von Tabellen und Fakten

Die Definition einer abgeleiteten Tabelle entspricht der Festlegung einer Sicht. Im Beispiel dient eine solche Sicht unter dem Namen VORGESETZTE zur Bestimmung des nächsthöheren Chefs für jeden Mitarbeitenden, sie entspricht einem Verbund der Tabelle CHEF auf sich selbst. Zu dieser Sicht lässt sich eine Ableitungsregel spezifizieren. Die Regel «ist_vorgesetzter_von (X,Y)» ergibt sich aus der Tatsache, dass es ein Z gibt, so dass X direkter Chef ist von Z und Z wiederum direkter Chef von Y. Damit drücken wir aus, dass X übernächster Vorgesetzter von Y ist, da Z dazwischenliegt.

Eine mit Fakten und Regeln bestückte Datenbank wird unvermittelt zu einer *Methoden- oder Wissensbank*, da sie nicht nur offenkundige Tatsachen wie «Huber ist Mitarbeiterin» oder «Huber ist direkte Chefin von Meier und Becker» enthält, sondern auch abgeleitete Feststellungen wie «Huber ist übergeordnete Chefin von Schweizer». Die übergeordneten Vorgesetzten aufzufinden, dazu dient die in Abb. 6.14 definierte Sicht VORGESETZTE. Die SQL-Abfrage über diese Sicht liefert mit der Resultattabelle die Tatsache, dass im Beispiel nur eine einzige übergeordnete Vorgesetztenbeziehung existiert, und zwar Mitarbeiter Schweizer und seine übergeordnete Vorgesetzte Huber. Zu dieser Erkenntnis führt auch die Anwendung der entsprechenden Ableitungsregel «ist_vorgesetzter_von».

Eine deduktive Datenbank als Gefäß für Fakten und Regeln unterstützt zusätzlich das *Prinzip der Rekursion*, das die Möglichkeit eröffnet, aufgrund der in der deduktiven

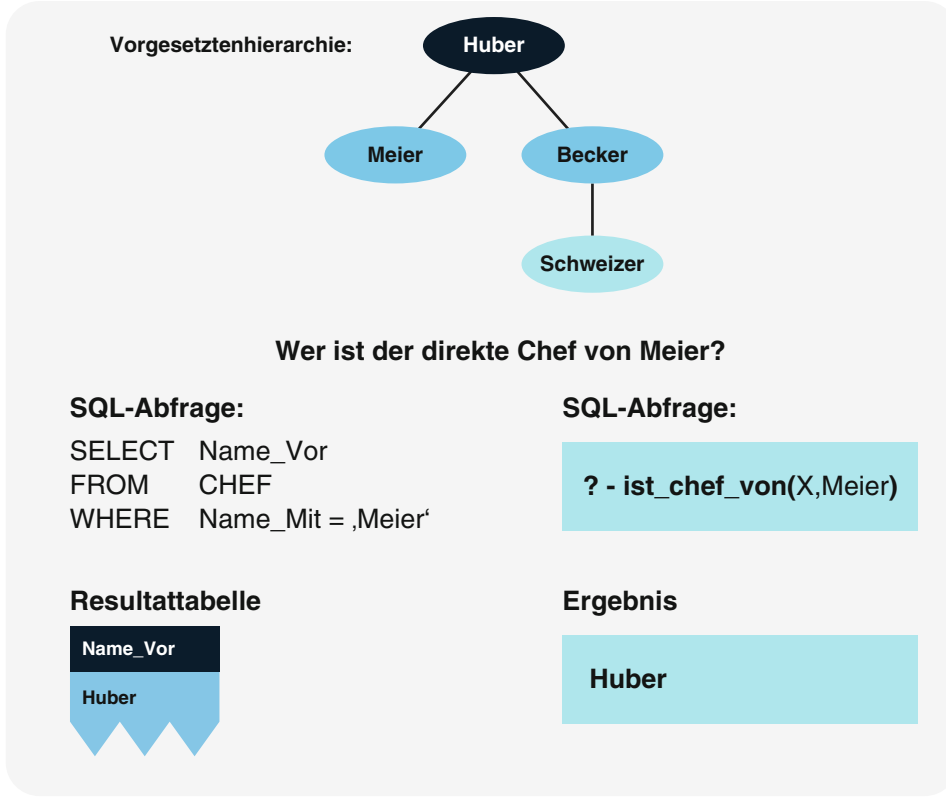

Abb. 6.13 Auswerten von Tabellen und Fakten

Datenbank enthaltenen Regeln beliebig viele korrekte Schlussfolgerungen zu ziehen. Aus einer wahrheitsgetreuen Aussage ergeben sich jeweils neue Aussagen.

Das Prinzip der Rekursion kann sich entweder *auf die Objekte der Datenbank oder auf die Ableitungsregeln* beziehen. Unter rekursiv definierten Objekten werden Strukturen verstanden, die selbst wiederum aus Strukturen zusammengesetzt sind und wie die beiden Abstraktionskonzepte Generalisation und Aggregation als hierarchische oder netzwerkartige Objektstrukturen aufgefasst werden können. Überdies lassen sich auch Aussagen rekursiv berechnen, etwa im Beispiel der Vorgesetztenhierarchie aus den Fakten «ist_mitarbeiter_von» und «ist_chef_von» alle direkten und indirekten Vorgesetztenbeziehungen.

Der Berechnungsvorgang, der aus einer bestimmten Tabelle alle transitiv abhängigen Tupel herleitet, bildet die *transitive Hülle* der Tabelle. Dieser Operator zählt nicht zu den ursprünglichen Operatoren der Relationenalgebra. Die transitive Hülle stellt vielmehr eine natürliche Erweiterung der relationalen Operatoren dar. Dabei kann sie nicht durch eine feste Anzahl von Berechnungsschritten, sondern nur durch eine nach dem jeweiligen Inhalt der Tabelle variierenden Anzahl von relationalen Verbund-, Projektions- und Vereinigungsoperatoren gebildet werden.

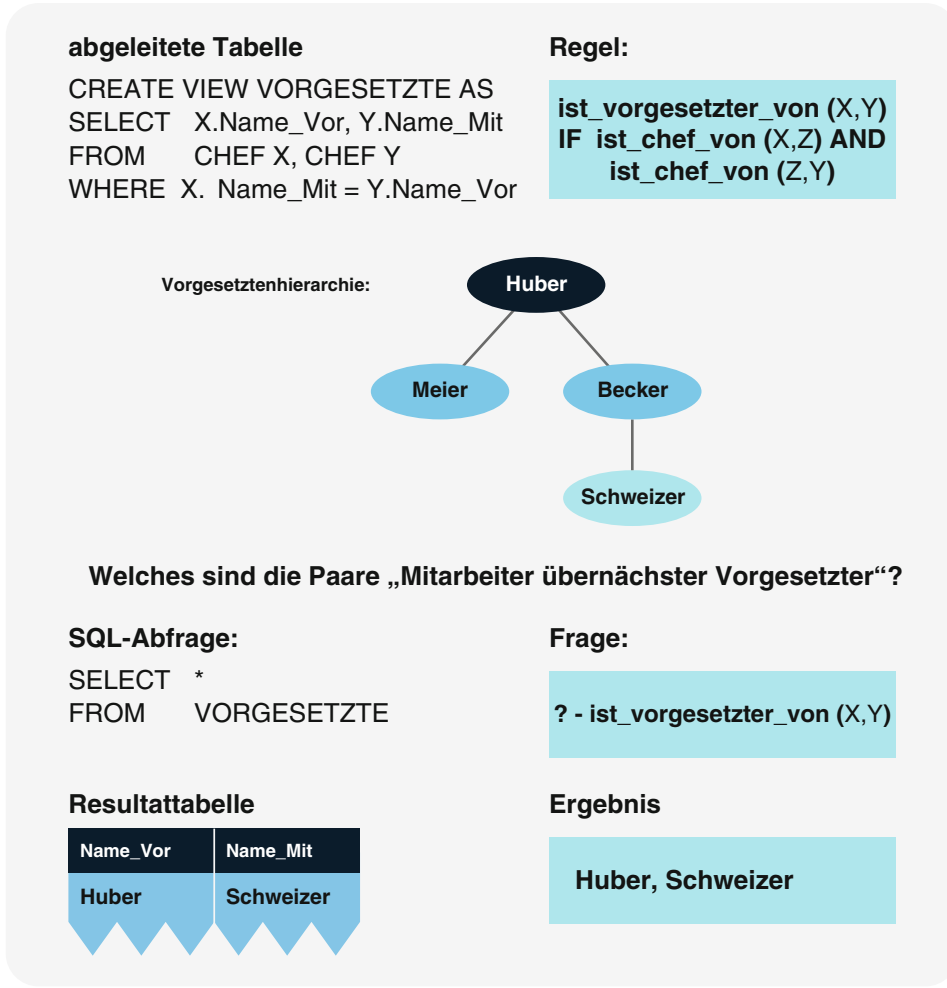

Abb. 6.14 Herleitung neuer Erkenntnisse

Diese Erläuterungen ergeben zusammengefasst die folgende Definition:

Wissensbasiertes Datenbanksystem
Ein wissensbasiertes Datenbankmanagementsystem unterstützt deduktive Datenbanken oder Wissensbanken,

- wenn es neben den eigentlichen Daten, also den *Fakten, auch Regeln* enthält,
- wenn die *Ableitungskomponente* es erlaubt, aus Fakten und Regeln weitere Fakten herzuleiten und
- wenn es die *Rekursion* unterstützt, mit der unter anderem die transitive Hülle einer Tabelle berechnet werden kann.

Unter dem Begriff *Expertensystem* versteht man ein Informationssystem, das für einen abgegrenzten Anwendungsbereich fachspezifische Kenntnisse und Schlussfolgerungen verfügbar macht. Wichtige Komponenten bilden eine Wissensbank mit Fakten und Regeln und eine Ableitungskomponente zur Herleitung neuer Erkenntnisse. Die Fachgebiete Datenbanken, Programmiersprachen und künstliche Intelligenz werden sich mehr und mehr beeinflussen und künftig effiziente Problemlösungsverfahren für die Praxis bereitstellen.

6.8 Fuzzy-Datenbanken

Bei herkömmlichen Datenbanksystemen werden die Merkmalswerte als präzise, sicher und scharf vorausgesetzt und Abfragen ergeben klare Ergebnisse:

- Die Merkmalswerte in den Datenbanken sind präzise, d. h. sie sind eindeutig. Die erste Normalform verlangt, dass die Merkmalswerte atomar sind und aus einem wohldefinierten Wertebereich stammen. Vage Merkmalswerte wie «2 oder 3 oder 4 Tage» oder «ungefähr 3 Tage» beim Terminverzug eines Lieferanten sind nicht zugelassen.
- Die in einer relationalen Datenbank abgelegten Merkmalswerte sind *sicher*, d. h. die einzelnen Werte sind entweder bekannt und somit wahr, oder sie sind unbekannt. Eine Ausnahme bilden Nullwerte, d. h. Merkmalswerte, die nicht oder noch nicht bekannt sind. Darüber hinaus bieten die Datenbanksysteme keine Modellierungskomponente für vorhandene Unsicherheit an. So sind Wahrscheinlichkeitsverteilungen für Merkmalswerte ausgeschlossen; es bleibt schwierig auszudrücken, ob ein bestimmter Merkmalswert dem wahren Wert entspricht oder nicht.
- Abfragen an die Datenbank sind *scharf*. Sie haben immer einen dichotomen Charakter, d. h. ein in der Abfrage vorgegebener Abfragewert muss mit den Merkmalswerten in der Datenbank entweder übereinstimmen oder nicht übereinstimmen. Eine Auswertung der Datenbank, bei der ein Abfragewert mit den gespeicherten Merkmalswerten «mehr oder weniger» übereinstimmt, ist nicht zuläßig.

Seit einigen Jahren werden Erkenntnisse aus dem *Gebiet der unscharfen Logik* (engl. *fuzzy logic*) auf Datenmodellierung und Datenbanken angewendet. Falls man unvollständige oder vage Sachverhalte zulässt, lässt sich ein größeres Anwendungsspektrum erschliessen. Die meisten dieser Arbeiten sind theoretischer Natur; einige Forschungsgruppen versuchen allerdings, mit Implementierungen die Nützlichkeit unscharfer Datenbankmodelle und Datenbanksysteme aufzuzeigen.

Der hier aufgezeigte Forschungsansatz basiert auf einem *Kontextmodell*, um Klassen von Datensätzen im relationalen Datenbankschema festlegen zu können. Bei der Klassifikation kann man zwischen scharfen und unscharfen Verfahren unterscheiden. Bei einer scharfen Klassifikation wird eine dichotomische Zuweisung der Datenbankobjekte zur Klasse vorgenommen, d. h. die Mengenzugehörigkeitsfunktion des Objekts zur

Klasse beträgt 0 für „nicht enthalten“ oder 1 für „enthalten“. Ein klassisches Verfahren würde daher einen Kunden entweder der Klasse «Kunden mit Umsatzproblemen» oder der Klasse «Kunden, zu denen die Geschäftsbeziehung ausgebaut werden soll» zuordnen. Ein unscharfes Verfahren dagegen lässt für die *Mengenzugehörigkeitsfunktion (*engl. *membership function) Werte zwischen 0 und 1* zu: Ein Kunde kann mit einem Wert von 0.3 zur Klasse «Kunden mit Umsatzproblemen» gehören und gleichzeitig mit einer Zugehörigkeit von 0.7 zur Klasse der «Kunden, zu denen die Geschäftsbeziehung ausgebaut werden soll». Eine unscharfe Klassifikation ermöglicht daher eine *differenziertere Interpretation der Klassenzugehörigkeit*: Man kann bei Datenbankobjekten einer unscharfen Klasse zwischen Rand- oder Kernobjekten unterscheiden, zudem können Datenbankobjekte zu zwei oder mehreren unterschiedlichen Klassen gleichzeitig gehören.

Im fuzzy-relationalen Datenmodell mit Kontexten, kurz im Kontextmodell, ist jedem Attribut Aj definiert auf einem Wertebereich D(Aj) ein Kontext zugeordnet. *Ein Kontext K(Aj) ist eine Partition von D(Aj) in Äquivalenzklassen*. Ein relationales Datenbankschema mit Kontexten besteht daher aus einer Menge von Attributen A = (A1,. . .,An) und einer weiteren Menge assoziierter Kontexte K = (K1(A1),. . .,Kn(An)).

Für eine Bewertung von Kunden dienen Umsatz und Treue als Beispiel. Zusätzlich werden die beiden qualifizierenden Attribute je in zwei Äquivalenzklassen zerlegt. Die entsprechenden Attribute und Kontexte für das Kundenbeziehungsmanagement lauten:

- **Umsatz in Euro pro Monat**: Der Wertebereich für den Umsatz in Euro soll durch [0..1000] definiert sein. Zudem werden die beiden Äquivalenzklassen [0..499] für kleinen Umsatz und [500..1000] für großen Umsatz gebildet.
- **Treue der Kunden**: Der Wertebereich {schlecht, schwach, gut, top} gilt für das Attribut der Kundentreue. Der Wertebereich wird in die beiden Äquivalenzklassen {schlecht, schwach} für negative Treue und {gut, top} für positive Treue zerlegt.

Die beiden vorgeschlagenen Merkmale mit ihren Äquivalenzklassen zeigen je ein Beispiel für ein numerisches und für ein qualitatives Attribut. Die entsprechenden Kontexte sind:

- K(Umsatz) = { [0..499], [500..1000] }
- K(Treue) = { {schlecht, schwach}, {gut, top} }

Die Partitionierung der Wertebereiche Umsatz und Treue ergibt in Abb. 6.15 die vier Äquivalenzklassen C1, C2, C3 und C4. Die inhaltliche Bedeutung der Klassen wird durch semantische Klassennamen ausgedrückt; so wird beispielsweise für Kunden mit kleinem Umsatz und schwacher Treue die Bezeichnung «Don’t Invest» für die Klasse C4 gewählt; C1 könnte für «Commit Customer» stehen, C2 für «Improve Loyalty» und C3 für «Augment Turnover». Es gehört zu den Aufgaben der Datenbankadministratoren, in Zusammenarbeit mit den Marketingspezialisten sowohl die Attribute wie die Äquivalenzklassen festzulegen und diese als Erweiterung des Datenbankschemas zu spezifizieren.

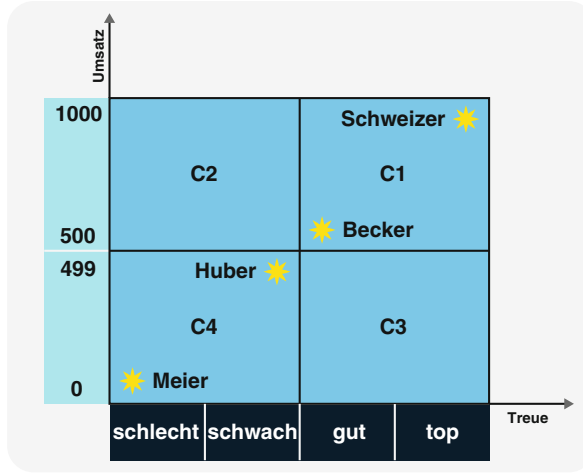

Abb. 6.15 Klassifikationsraum aufgespannt durch die Attribute Umsatz und Treue

Das Kundenbeziehungsmanagement bezweckt, anstelle produktbezogener Argumentationslisten und Anstrengungen, die kundenindividuellen Wünsche und das Kundenverhalten miteinzubeziehen. Sollen Kunden als Vermögenswert (engl. *customer value*) aufgefasst werden, so müssen sie entsprechend ihrem Markt- und Ressourcenpotenzial behandelt werden. Mit scharfen Klassen, d. h. traditionellen Kundensegmenten, ist dies kaum möglich, da alle Kundinnen und Kunden in einer Klasse gleich behandelt werden. In Abb. 6.15 beispielsweise besitzen Becker und Huber einen ähnlichen Umsatz und zeigen ein ähnliches Treueverhalten. Trotzdem werden sie bei einer scharfen Segmentierung unterschiedlich klassifiziert: Becker gehört zur Premiumklasse C1 (Commit Customer) und Huber zur Verliererklasse C4 (Don't Invest). Zusätzlich wird der topgesetzte Kunde Schweizer gleich behandelt wie Becker, da beide zum Segment C1 gehören.

Gemäß Abb. 6.15 können bei einer scharfen Kundensegmentierung folgende Konfliktsituationen auftreten:

- Kunde Becker hat wenige Anreize, seinen Umsatz zu steigern oder die Kundenbindung und -treue zu verbessern. Er liegt in der Premiumklasse C1 und geniesst die entsprechenden Vorteile.
- Kunde Becker kann überrascht werden, falls sein Umsatz ein wenig zurückgeht oder sein Treuebonus abnimmt. Plötzlich sieht er sich einem anderen Kundensegment zugeordnet; im Extremfall fällt er von der Premiumklasse C1 in die Verliererklasse C4.
- Kundin Huber verfügt über einen ordentlichen Umsatz und eine mittlere Kundentreue, wird aber als Verlierer behandelt. Es wird kaum überraschen, wenn sich Huber im Markt umsieht und abspringt.
- Eine scharfe Kundensegmentierung lässt auch für Kunde Schweizer eine kritische Situation entstehen. Er ist im Moment der profitabelste Kunde mit ausgezeichnetem Ruf, wird aber vom Unternehmen nicht entsprechend seinem Kundenwert wahrgenommen und behandelt.

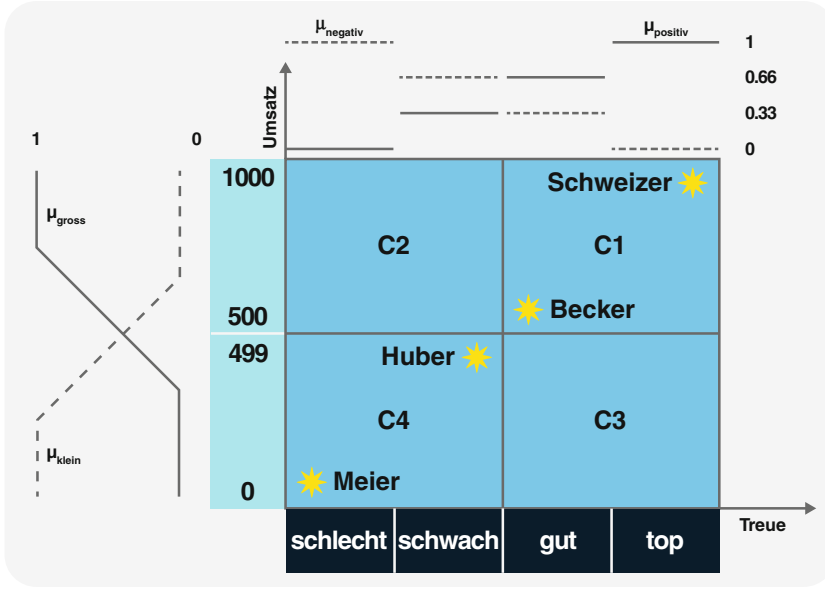

Abb. 6.16 Unscharfe Partitionierung der Wertebereiche mit Zugehörigkeitsfunktionen

Die hier exemplarisch aufgezeigten Konfliktsituationen können entschärft oder eliminiert werden, falls unscharfe Kundenklassen gebildet werden. Die Positionierung eines Kunden im zwei- oder mehrdimensionalen Datenraum entspricht dem Kundenwert, der jetzt aus unterschiedlichen Klassenzugehörigkeitsanteilen besteht.

Gemäß Abb. 6.16 kann für einen bestimmten Kunden die Treue als linguistische Variable (engl. *linguistic variable*) gleichzeitig «positiv» und «negativ» sein; zum Beispiel ist die Zugehörigkeit von Becker zur unscharfen Menge $\mu_{positiv}$ 0.66 und diejenige zur Menge $\mu_{negativ}$ ist 0.33. Der Treuegrad von Becker ist also nicht ausschließlich stark oder schwach wie bei scharfen Klassen.

Die lingusitische Variable Treue mit den vagen Termen «positiv» und «negativ» und den Zugehörigkeitsfunktionen $\mu_{positiv}$ und $\mu_{negativ}$ bewirkt, dass der Wertebereich D(Treue) unscharf partitioniert wird. Analog wird der Wertebereich D(Umsatz) durch die Terme «groß» und «klein» unterteilt. Dadurch entstehen im Kontextmodell Klassen mit kontinuierlichen Übergängen, d. h. unscharfe Klassen.

Die Zugehörigkeit eines Objekts zu einer Klasse ergibt sich aus der Aggregation über alle Terme, die die Klasse definieren. Die Klasse C1 wird durch die Terme «gross» (für die linguistische Variable Umsatz) und «positiv» (für die linguistische Variable Treue) beschrieben. Die Aggregation muss daher einer Konjunktion der einzelnen Zugehörigkeitswerte entsprechen. Dazu sind in der Theorie der unscharfen Mengen verschiedene Operatoren entwickelt worden.

Klassifikationsabfragen mit der *Sprache fCQL* (fuzzy Classification Query Language) operieren auf der linguistischen Ebene mit vagen Kontexten. Das hat den Vorteil, dass die

Anwendenden keinen scharfen Zielwert und keinen Kontext kennen müssen, sondern lediglich den Spaltennamen des objektidentifizierenden Merkmals und die Tabelle oder Sicht, in der die Merkmalswerte enthalten sind. Für eine gezielte Betrachtung einzelner Klassen können die Anwendenden eine Klasse spezifizieren oder Merkmale mit einer verbalen Beschreibung ihrer Ausprägungen angeben. Klassifikationsabfragen arbeiten also mit verbalen Beschreibungen auf Merkmals- und Klassenebene:

```
CLASSIFY    Objekt
FROM        Tabelle
WITH        Klassifikationsbedingung
```

Die Sprache fCQL ist an SQL angelehnt, wobei anstelle des SELECT eine CLASSIFY-Klausel die Projektionsliste durch den Spaltennamen des zu klassifizierenden Objekts nennt. Während die WHERE-Klausel bei SQL eine Selektionsbedingung enthält, wird mit der WITH-Klausel die Klassifikationsbedingung festgelegt. Als Beispiel einer fCQL-Abfrage erzeugt

```
CLASSIFY    Kunde
FROM        Kundentabelle
```

eine Klassifikation sämtlicher in der Tabelle vorhandener Kunden. Mit

```
CLASSIFY    Kunde
FROM        Kundentabelle
WITH        CLASS IS Augment Turnover
```

wird gezielt die Klasse C3 abgefragt. Verzichtet man auf die Definition der Klasse, so kann man mit den linguistischen Beschreibungen der Äquivalenzklassen eine bestimmte Objektmenge selektieren. Als Beispiel gilt die Abfrage:

```
CLASSIFY    Kunde
FROM        Kundentabelle
WITH        Umsatz IS klein AND Treue IS positiv
```

Diese Abfrage besteht aus dem Bezeichner des zu klassifizierenden Objekts (Kunde), dem Namen der Grundtabelle (Kundentabelle), den kritischen Merkmalsnamen (Umsatz und Treue) sowie dem Term «klein» der linguistischen Variablen Umsatz sowie dem Term «positiv» der linguistischen Variablen Treue.

Aufgrund des obigen Beispiels und der gemachten Erläuterungen lassen sich unscharfe Datenbanken wie folgt charakterisieren:

Fuzzy-Datenbanksystem
Ein Fuzzy-Datenbankmanagementsystem ist ein Datenbanksystem, das die folgenden Eigenschaften aufweist:

- Das Datenmodell ist unscharf relational, d. h. es lässt *impräzise, vage und unsichere Merkmalswerte* zu.
- Abhängigkeiten zwischen den Attributen oder Merkmalen werden mit *unscharfen Normalformen* ausgedrückt.
- Sowohl der Relationenkalkül wie die Relationenalgebra können mit Hilfe der Fuzzy Logic zum *unscharfen Relationenkalkül* resp. zur *unscharfen Relationenalgebra* ausgebaut werden.
- Mit einer durch linguistische Variablen erweiterten Klassifikationssprache lassen sich *unscharfe Abfragen formulieren*.

Seit Jahren forschen vereinzelte Informatiker auf dem Gebiet der unscharfen Logik und relationaler Datenbanksysteme (vgl. Abschn. 6.9). Diese Arbeiten werden vorwiegend auf dem Gebiet der Fuzzy Logic und weniger im Bereich der Datenbanken publiziert und entsprechend honoriert. Es ist zu hoffen, dass die beiden Forschungsgebiete sich in Zukunft nähern und die Exponenten der Datenbanktechnologie das Potenzial unscharfer Datenbanken und unscharfer Abfragesprachen erkennen.

6.9 Literatur

Ein Standardbuch über verteilte Datenbanksysteme stammt von Ceri und Pelagatti (1985), eine weitere Übersicht über das Gebiet bieten Özsu und Valduriez (1991); Dadam (1996) illustriert verteilte Datenbanken und Client-Server-Systeme. Das deutschsprachige Werk von Rahm (1994) behandelt neben Aspekten verteilter Datenbanksysteme auch Architekturfragen paralleler Mehrrechner. Grundlegende Arbeiten über verteilte Datenbanksysteme basieren auf den Erweiterungen von «System R» nach Williams et al. (1982) und von «Ingres» nach Stonebraker (1986). Die Arbeiten von Rothnie et al. (1980) sind ebenfalls von Interesse, da das entsprechende Datenbanksystem «SDD-1» den ersten Prototypen eines verteilten Systems darstellte.

Zur Integration der Zeit in relationale Datenbanken liegen verschiedene Vorschläge vor; erwähnt seien die Arbeiten von Clifford und Warren (1983); Gadia (1988) und Snodgrass (1987). Snodgrass et al. (1994) hat zusammen mit weiteren Forschungskollegen die SQL-Sprache um temporale Konstrukte erweitert; die Sprache ist unter dem Namen TSQL2 bekannt geworden (Snodgrass et al. 1994). Weitere Forschungsarbeiten

über temporale Datenbanken finden sich in Etzion et al. (1998); Myrach (2005) behandelt temporale Aspekte in betrieblichen Informationssystemen.

Als Standardwerk für Data Warehouse gilt das Buch von Inmon (2005), aus welchem auch die hier verwendete Definition stammt. Kimball et al. (2008) widmen sich ebenfalls dieser Thematik. Im deutschsprachigen Raum sind die Werke von Gluchowski et al. (2008) sowie Mucksch und Behme (2000) bekannt. Jarke et al. (2000) vermitteln Grundlagen über das Data Warehouse und zeigen Forschungsprojekte auf. Über Data Mining haben Berson und Smith (1997) sowie Witten und Frank (2005) Bücher verfasst. Die Integration von Datenbanken im World Wide Web illustrieren verschiedene Autoren in einem Schwerpunktheft von Meier (2000).

Objektorientierte Ansätze zur Erweiterung relationaler Datenbanken werden von Dittrich (1988); Lorie et al. (1985); Meier (1987) sowie von Schek und Scholl (1986) aufgezeigt. Bücher über objektorientierte Datenbanken stammen von Bertino und Martino (1993); Cattell (1994); Geppert (2002); Heuer (1997); Hughes (1991) und Kim (1990); Lausen und Vossen (1996) beschreiben grundlegende Aspekte objektrelationaler und objektorientierter Datenbanksprachen. Die deutschsprachigen Werke von Kemper und Eickler (2013); Lang und Lockemann (1995) sowie Saake et al. (2013) erklären Entwicklungen relationaler und objektorientierter Datenbanksysteme. Meier und Wüst (2003) haben eine Einführung in das Gebiet objektorientierter und objektrelationaler Datenbanken für Praktiker verfasst. Stonebraker (1996) erläutert objektrelationale Datenbanksysteme. Die Entwicklung zur Erweiterung des SQL-Standards ist in Türker (2003) zusammengefasst. Die Veröffentlichungen von Coad und Yourdon (1991) sowie von Martin und Odell (1992) enthalten Bausteine für den objektorientierten Datenbankentwurf. Stein (1994) gibt in seinem Fachbuch einen Vergleich verschiedener objektorientierter Analysemethoden.

Die regelbasierte Sprache bei deduktiven Datenbanksystemen wird oft als Datalog bezeichnet, in Anlehnung an den Begriff Data und die bekannte Logikprogrammiersprache Prolog. Ein Klassiker unter den Prolog-Lehrbüchern ist das Werk von Clocksin und Mellish (1994), eine formale Abhandlung der logischen Datenbankprogrammierung geben Maier und Warren (1988). Das deutschsprachige Werk von Cremers et al. (1994) behandelt auf ausführliche Weise das Gebiet der deduktiven Datenbanken. Die Werke von Gallaire et al. (1984) sowie von Gardarin und Valduriez (1989) befassen sich zu einem grossen Teil mit deduktiven Datenbanken.

Das Forschungsgebiet unscharfer Mengen (Fuzzy Sets) wurde von Lotfi A. Zadeh begründet (Zadeh 1965), u. a. zur Erweiterung der klassischen Logik mit den beiden Werten «wahr» und «falsch» zu einer unscharfen Logik (Fuzzy Logic) mit beliebig vielen Wahrheitswerten. Seit einigen Jahren werden Erkenntnisse daraus auf Datenmodellierung und Datenbanken angewendet, siehe z. B. Bordogna und Pasi (2000); Bosc und Kacprzyk (1995); Chen (1998); Petra (1996) oder Pons et al. (2000). Mit Hilfe der Fuzzy Logic wurden verschiedene Modellerweiterungen vorgeschlagen, sowohl für das Entitäten-

Beziehungsmodell wie für das Relationenmodell. Beispielsweise hat Chen (1992) in seiner Dissertation die klassischen Normalformen der Datenbanktheorie zu unscharfen erweitert, indem er bei den funktionalen Abhängigkeiten Unschärfe zuliess; siehe dazu auch Shenoi et al. (1992). Weitere Vorschläge zu unscharfen Datenmodellen für Datenbanken finden sich in Kerre und Chen (1995). Takahashi (1995) schlägt eine Fuzzy Query Language (FQL) auf der Basis des Relationenkalküls vor. Die Sprache FQUERY von Kacprzyk und Zadrozny (1995) verwendet unscharfe Terme und ist im Microsoftprodukt Access prototypartig implementiert worden. Ein etwas anderer Ansatz wird mit unscharfer Klassifikation gewählt, ursprünglich von Schindler (1998) vorgeschlagen. Dazu wurde die unscharfe Klassifikationssprache fCQL (fuzzy Classification Query Language) entworfen und in einem Prototypen (vgl. Meier et al. 2005; 2008; Werro 2015) realisiert.

NoSQL-Datenbanken

7

7.1 Zur Entwicklung nichtrelationaler Technologien

Der Begriff NoSQL wurde erstmals 1998 für eine (allerdings relationale) Datenbank verwendet, welche keine SQL-Schnittstelle aufwies. Der Aufstieg begann in den 2000er-Jahren, insbesondere mit der Entwicklung des Webs. Der Einsatz von sogenannten *Web-Scale-Datenbanken* wurde mit der wachsenden Popularität globaler Web-Dienste immer häufiger. Es bestand ein Bedarf an Datenhaltungssysteme, welche mit den enorm großen Datenmengen von Web-Diensten (teilweise im Petabyte-Bereich oder größer) umgehen können.

Die relationalen resp. SQL-Datenbanksysteme sind weit mehr als reine Datenspeicher. Sie bieten einen hohen Grad an Verarbeitungslogik:

- Mächtige deklarative Sprachkonstrukte
- Schemata, Metadaten
- Konsistenzgewährung
- Referenzielle Integrität, Trigger
- Recovery, Logging
- Mehrbenutzerbetrieb, Synchronisierung
- User, Rollen, Security
- Indexierung

All diese Funktionalitäten von SQL bieten viele Vorteile bezüglich Konsistenz und Sicherheit der Daten. Dies zeigt, dass SQL-Datenbanken vor allem auf Integrität und Transaktionsschutz ausgerichtet sind, wie sie beispielsweise in Bankanwendungen oder Versicherungssoftware erforderlich sind. Da die Überprüfung von Datenintegrität aber mit viel Aufwand und Rechenleistung verbunden ist, stossen relationale Datenbanken bei

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2_7

umfangreichen Datenmengen schneller an Grenzen. Die Mächtigkeit des Datenbankverwaltungssystems bewirkt Nachteile hinsichtlich der Effizienz und Performanz, aber auch hinsichtlich der Flexibilität in der Datenverarbeitung.

In der Praxis stehen konsistenzorientierte Verarbeitungskomponenten einer effizienten Verarbeitung immenser Datenmengen oft im Weg; gerade auch in Anwendungsbereichen, bei denen die Performanz und nicht die Konsistenz im Vordergrund steht, wie beispielsweise bei sozialen Medien. Aus diesem Grund wurde in der Open Source- und Web-Development-Community schon bald die Entwicklung massiv verteilter Datenbanksysteme vorangetrieben, welche diesen neuartigen Anforderungen standhalten kann.

NoSQL-Datenbank
Eine NoSQL-Datenbank weist folgende Eigenschaften auf (vgl. Abschn. 1.4.3):

- Das Datenbankmodell ist nicht relational
- Ausrichtung auf verteilte und horizontale Skalierbarkeit
- Schwache oder keine Schema-Restriktionen
- Einfache Datenreplikation
- Einfacher Zugriff über eine Programmierschnittstelle (Application Programming Interface, API)
- Anderes Konsistenzmodell als ACID (z. B. BASE, siehe Abschn. 4.3.1)

Obwohl mit der Bezeichnung NoSQL ganz klar Datenbanken gemeint sind, die über keinen SQL-Zugang verfügen, hat sich als Erklärung des Begriffs die Redewendung „not only SQL“ resp. „nicht nur SQL“ verbreitet. Verschiedene Datenbankmodelle sind für verschiedene Zwecke geeignet, und der Einsatz von diversen Datenbankfamilien in einer Anwendung ist ein Mehrwert, wenn jede ihre eigenen Stärken einbringen kann. Dieses Konzept wird mit dem Begriff *Polyglot Persistence* (wörtlich: *mehrsprachige Persistenz*) bezeichnet. So können in einer Anwendung sowohl SQL- als auch NoSQL-Technologien zum Zug kommen.

Die Kerntechnologien im Bereich NoSQL sind:

- Schlüssel-Wert-Datenbanken (Abschn. 7.2),
- Spaltenfamilien-Datenbanken (Abschn. 7.3),
- Dokument-Datenbanken und (Abschn. 7.4) und
- Graphdatenbanken (Abschn. 7.6).

Diese vier Datenbankmodelle, die sogenannten *Core-NoSQL-Modelle*, werden in diesem Kapitel beschrieben. Weitere Arten von NoSQL-Datenbanken werden unter dem Begriff Soft NoSQL kategorisiert. Dazu gehören beispielsweise Objektdatenbanken, Gitter-Datenbanken sowie die Familie der XML-Datenbanken (Abschn. 7.5).

7.2 Schlüssel-Wert-Datenbanken

Die einfachste Art, Daten zu speichern, ist die Zuweisung eines Werts zu einer Variablen bzw. zu einem Schlüssel. Auf Hardware-Ebene arbeiten CPUs mit Registern, die auf diesem Modell aufgebaut sind; in Programmiersprachen gibt es dazu das Konstrukt der assoziativen Arrays. Das einfachste mögliche Datenbankmodell ist analog dazu ein Datenspeicher, welcher *zu einem Datenobjekt als Schlüssel ein Datenobjekt als Wert speichert*.

In einer *Schlüssel-Wert-Datenbank* (engl. *key/value store*) kann zu einem Schlüssel mit einem ganz einfachen Befehl, z. B. SET, ein bestimmter Wert gespeichert werden. Im Folgenden werden Daten zu einem Benutzer einer Webseite gespeichert: Vorname, Nachname, E-Mail und verschlüsseltes Passwort. Beispielsweise wird für den Schlüssel User:U17547:vorname der Wert Max gespeichert.

```
SET User:U17547:vorname Max
SET User:U17547:nachname Müller
SET User:U17547:email max.müller@blue_planet.net
SET User:U17547:pwhash D75872C818DC63BC1D87EA12
SET User:U17548:vorname Mina
SET User:U17548:nachname Maier
...
```

Datenobjekte können mit einer einfachen Abfrage unter Verwendung des Schlüssels abgerufen werden:

```
GET User:U17547:email
> max.müller@blue_planet.net
```

Der Schlüsselraum kann allenfalls mit einem Sonderzeichen wie einem Doppelpunkt oder einem Schrägstrich strukturiert werden. So kann ein Namensraum definiert werden, der eine rudimentäre Datenstruktur abbilden kann. Ansonsten unterstützt ein Schlüssel-Wert-Speicher keine weiteren Arten von Struktur, weder Verschachtelungen noch Referenzen. Eine Schlüssel-Wert-Datenbank ist schemafrei; das heisst Datenobjekte können jederzeit in beliebiger Form gespeichert werden, ohne dass irgendwelche Metadatenobjekte wie Tabellen oder Spalten vorher definiert werden müssen. Aufgrund des Verzichts auf Schema und referenzieller Integrität wird der Schlüssel-Wert-Speicher performant in der Anfrage, einfach partitionierbar und flexibel in der Art der zu speichernden Daten.

Schlüssel-Wert-Datenbank
Eine Datenbank mit den folgenden Eigenschaften wird Schlüssel-Wert-Datenbank (Key/Value Store) genannt:

- Es gibt eine Menge von identifizierenden Datenobjekten, die *Schlüssel*.
- Zu jedem Schlüssel gibt es genau ein assoziiertes deskriptives Datenobjekt, welches den *Wert* zum zugehörigen Schlüssel darstellt.
- Mit der Angabe des Schlüssels kann der zugehörige Wert aus der Datenbank abgefragt werden.

Key/Value Stores erfahren im Zuge der NoSQL-Bewegung einen starken Aufschwung, weil sie für große Datenmengen skalierbar sind. Da die Anforderung der referentiellen Integrität bei Key/Value Stores nicht überprüft wird, können umfangreiche Datenmengen effizient geschrieben und wieder gelesen werden. Sind die Schlüssel-Wert-Paare im Hauptspeicher (Memory, RAM) der Datenbank zwischengespeichert, erhöht sich die Geschwindigkeit der Datenverarbeitung nochmal um ein Vielfaches. In diesem Fall spricht man von *In-Memory-Datenbanken*. Diese verwenden Technologien für das Caching von Werten im Hauptspeicher, wobei laufend mit den im Hintergrundspeicher längerfristig persistierten Daten abgeglichen wird.

Die Skalierbarkeit von Key/Value Stores kann mit Fragmentierung der Datenbasis bzw. *Sharding* quasi beliebig ausgebaut werden. Bei Schlüssel-Wert-Datenbanken ist die Partitionierung aufgrund des einfachen Modells leicht zu bewerkstelligen. Einzelne Rechner im Cluster, sogenannte *Shards*, nehmen dabei nur einen Teilraum der Schlüssel bei sich auf. So kann die Datenbank auf eine große Anzahl von einzelnen Rechnern verteilt werden. Die Verteilung der Schlüssel basiert oft auf dem Prinzip des Consistent Hashing (Abschn. 5.2.3).

Abbildung 7.1 zeigt eine verteilte Architektur für eine Schlüssel-Wert-Datenbank: Aus dem Schlüssel wird ein Zahlenwert (Hash) generiert. Unter Anwendung des Modulo-Operators kann dieser Wert nun auf eine vorgegebene Anzahl von Adressräumen (Hash Slots) verteilt werden. So wird ermittelt, auf welchem Shard der Wert zum Schlüssel innerhalb der verteilten Architektur gespeichert wird. Um die Ausfallsicherheit zu erhöhen, kann die verteilte Datenbank auf andere Rechner kopiert und aktuell gehalten werden. In diesem Fall sprechen wir von Replikation. Die eigentliche Datenbasis, der Master-Cluster, wird dabei mit einer oder mehreren replizierten Datenbasen, den Slave Clusters, synchronisiert.

Die Abb. 7.1 zeigt exemplarisch eine mögliche hochperformante, massiv verteilte Architektur für eine Schlüssel-Wert-Datenbank. Der Master Cluster enthält drei Rechner (Shards A bis C). Für kurze Antwortzeiten werden die Daten direkt im Hauptspeicher (RAM) gehalten. Für die Speicherung auf der Festplatte wird die Datenbasis auf einem Slave-Cluster repliziert, wo die Daten auf die Festplatte geschrieben werden. Ein weiterer

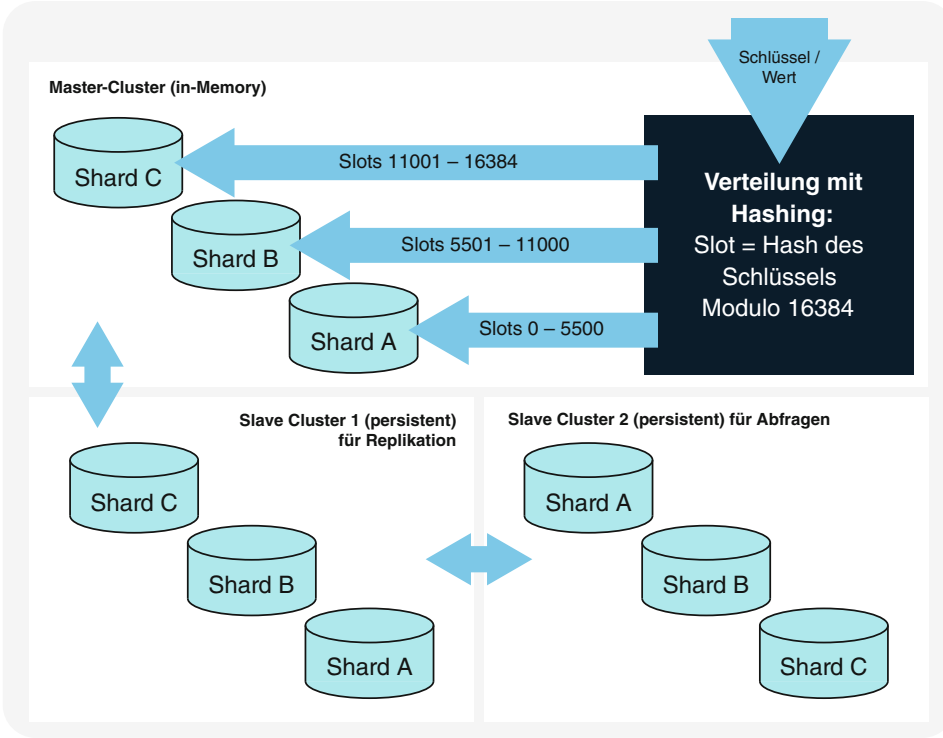

Abb. 7.1 Eine massiv verteilte Schlüssel-Wert-Datenbank mit Sharding und Hash-basierter Schlüsselverteilung

Slave-Cluster erhöht die Performance, indem für komplexe Abfragen und Analysen ein zusätzlicher replizierter Rechnercluster bereitgestellt wird.

Ein weiterer Vorteil von Daten-Wert-Speichern über das effiziente Sharding für große Datenmengen hinaus ist die Flexibilität im Datenschema. In einem relationalen Datenbanksystem muss zu jedem Datensatz, der zu speichern ist, im Voraus ein Schema in der Form einer Relation mit Attributen bestehen (CREATE TABLE). Ist dies nicht der Fall, muss vor der Datenspeicherung eine Schemadefinition vorgenommen werden. Im Fall von Datenbanktabellen mit einer großen Menge von Datensätzen oder beim Arbeiten mit heterogenen Daten ist dies oft mit erheblichem Aufwand verbunden. Schlüssel-Wert-Datenbanken sind quasi schemafrei und somit flexibel in der Art der zu speichernden Daten. Es muss also keine Tabelle mit Spalten und Datentypen spezifiziert werden – Daten können einfach unter einem beliebigen Schlüssel abgelegt werden. Allerdings bringt die Abwesenheit eines Datenbankschemas oft Unordnung in die Datenverwaltung.

7.3 Spaltenfamilien-Datenbanken

Obwohl Schlüssel-Wert-Datenbanken performant große Datenmengen verarbeiten, ist deren Struktur doch sehr rudimentär. Oftmals ist eine Möglichkeit zur Strukturierung des Datenraums erforderlich; ein Schema. Aus diesem Grund gibt es bei *Spaltenfamilien-Datenbanken* (engl. *column family stores*) eine Erweiterung des Schlüssel-Wert-Konzepts mit etwas mehr Struktur.

Bei der Speicherung relationaler Tabellen hat sich gezeigt, dass es effizienter ist, die Daten für die Optimierung des Lesezugriffs nicht zeilenweise, sondern spaltenweise zu speichern. Es ist nämlich so, dass für eine Zeile selten alle Spalten benötigt werden, aber es *Gruppen von Spalten gibt, die häufig zusammen gelesen* werden. Aus diesem Grund macht es für die Optimierung des Zugriffs Sinn, Daten in solchen Gruppen von Spalten – in Spaltenfamilien – als Speichereinheit zu strukturieren. Die danach benannten heutigen Column Family Stores orientieren sich an diesem Modell, aber sie speichern Daten nicht in relationalen Tabellen, sondern in erweiterten und strukturierten mehrdimensionalen Schlüsselräumen.

Google hat 2008 mit BigTable ein Datenbankmodell für die verteilte Speicherung von strukturierten Daten vorgestellt und hat damit die Entwicklung von Spaltenfamilien-Datenbanken maßgeblich beeinflusst.

BigTable
Im BigTable-Modell ist eine *Tabelle* eine dünnbesetzte, verteilte, multidimensionale, sortierte Map. Sie hat folgende Eigenschaften:

- Es handelt sich bei der Datenstruktur um eine Abbildung, welche Elemente aus einer Definitionsmenge Elementen einer Zielmenge zuordnet.
- Die Abbildung ist sortiert, d. h. es existiert eine Ordnungsrelation für die Schlüssel, welche die Zielelemente adressieren.
- Die Adressierung ist mehrdimensional, d. h. die Funktion hat mehr als einen Parameter.
- Die Daten werden durch die Abbildung verteilt, d. h. sie können auf vielen verschiedenen Rechnern an räumlich unterschiedlichen Orten gespeichert sein.
- Die Abbildung ist dünnbesiedelt, muss also nicht für jeden möglichen Schlüssel einen Eintrag aufweisen.

In BigTable hat eine Tabelle drei Dimensionen: Sie bildet für eine Zeile (*row*) und eine Spalte (*column*) zu einem bestimmten Zeitpunkt (*time*) einen Eintrag der Datenbank als String ab:

```
(row:string, column:string, time:int64) → string
```

Tabellen in Spaltenfamilien-Datenbanken sind mehrstufige aggregierte Strukturen. Der erste Schlüssel, der Zeilenschlüssel, ist analog zur Schlüssel-Wert-Datenbank eine Adressierung eines Datenobjekts. Innerhalb dieses Schlüssels befindet sich nun aber eine

weitere Struktur, die Unterteilung der Zeile in verschiedene Spalten, welche selbst wiederum mit Schlüsseln adressiert wird. Einträge der Tabelle werden zusätzlich mittels Zeitstempel versioniert. Die Speichereinheit, die mit einer bestimmten Kombination aus Zeilenschlüssel, Spaltenschlüssel und Zeitstempel adressiert wird, heisst *Zelle*.

Spalten in einer Tabelle werden zu Spaltenfamilien gruppiert. Diese bilden die Einheit für die Zugriffskontrolle, d. h. für die Vergabe von Lese- und Schreibrechten für Benutzer und Applikationen. Zudem dient die Einheit der Spaltenfamilie der Zuordnung von Arbeitsspeicher und Festplatte. Die Spaltenfamilien sind *die einzigen festen Schemaregeln* der Tabelle, deshalb müssen sie explizit durch Änderung des Schemas der Tabelle erstellt werden. Im Unterschied zu relationalen Tabellen können aber innerhalb einer Spaltenfamilie beliebige Spaltenschlüssel für die Speicherung von Daten verwendet werden. Die Spaltenfamilie dient somit als *rudimentäres Schema*, mit einer reduzierten Menge an Metadaten.

Daten innerhalb einer Spaltenfamilie haben den gleichen Typ, denn es wird davon ausgegangen, dass sie zusammen gelesen werden. Die Datenbank speichert deshalb Daten zu einer Spaltenfamilie in einer Zeile der Tabelle immer auf dem gleichen Rechner. Dieser Mechanismus beschleunigt gemeinsame Lesezugriffe innerhalb der Spaltenfamilie. Zu diesem Zweck ordnet das Datenbankverwaltungssystem die Spaltenfamilien zu *Lokalitätsgruppen*. Diese definieren, auf welchem Rechner und in welchem Format die Daten gespeichert werden. Daten in einer Lokalitätsgruppe werden physisch auf dem gleichen Rechner gespeichert. Zudem können für Lokalitätsgruppen bestimmte Parameter gesetzt werden, beispielsweise kann für eine Lokalitätsgruppe definiert werden, dass sie im Arbeitsspeicher gehalten wird. Somit können Daten zu einer Spaltenfamilie schnell und ohne Festplattenzugriff gelesen werden.

Zusammengefasst zeigt Abb. 7.2, wie Daten im beschriebenen BigTable-Modell gespeichert werden: Eine Datenzelle wird mit Zeilenschüssel und Spaltenschlüssel adressiert. In diesem Beispiel gibt es pro Benutzer einen Zeilenschlüssel. Die Inhalte sind ausserdem mit einem Zeitstempel historisiert. Verschiedene Spalten werden als Spaltenfamilie zusammengefasst: Die Spalten Mail, Name und Phone gehören zur Spaltenfamilie Contact. Zugangsdaten wie Benutzernamen und Passwörter könnten in der Spaltenfamilie Access gespeichert werden. Die Spalten innerhalb einer Spaltenfamilie sind dünnbesiedelt (engl. *sparse*). Im Beispiel in Abb. 7.2 enthält die Zeile U17547 einen Wert für die Spalte Contact:Mail, aber nicht für die Spalte Contact:Phone. Ist kein Eintrag vorhanden, wird diese Information in der Zeile auch nicht gespeichert.

Spaltenfamilien-Datenbank
Datenbanken, welche ein dem BigTable-Modell ähnliches Datenmodell verwenden, werden Spaltenfamilien-Datenbanken (Column Family Stores) genannt. Darunter verstehen wir eine NoSQL-Datenbank mit folgenden Eigenschaften:

- Daten werden in mehrdimensionalen Tabellen gespeichert.
- Datenobjekte werden mit Zeilenschlüsseln adressiert.
- Objekteigenschaften werden mit Spaltenschlüsseln adressiert.

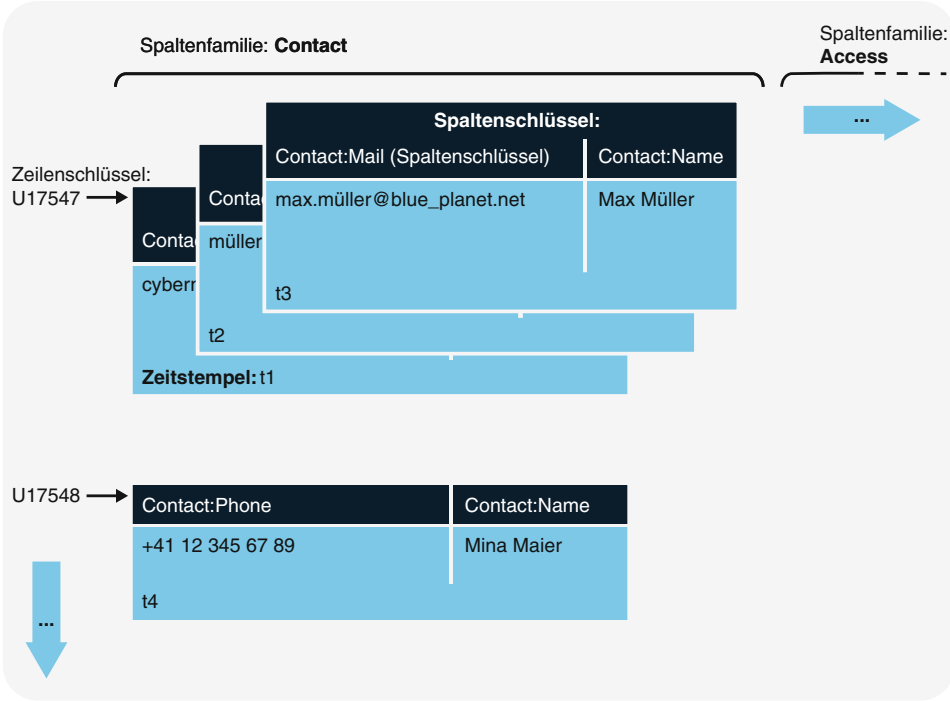

Abb. 7.2 Speicherung von Daten mit dem BigTable-Modell

- Spalten der Tabelle werden zu Spaltenfamilien zusammengefasst.
- Das Schema einer Tabelle bezieht sich ausschließlich auf Spaltenfamilien; innerhalb einer Spaltenfamilie können beliebige Spaltenschlüssel verwendet werden.
- Bei verteilten, fragmentierten Architekturen werden Daten zu einer Spaltenfamilie physisch möglichst am gleichen Ort gespeichert (Ko-Lokation), um die Antwortzeiten zu optimieren.

Die Vorteile von Spaltenfamilien-Datenbanken sind hohe Skalierbarkeit und Verfügbarkeit durch massive Verteilung, gleich wie bei Key/Value Stores. Zudem bieten sie eine nützliche Struktur durch ein Schema mit Zugriffskontrolle und Lokalisation von verteilten Daten auf Ebene Spaltenfamilie; gleichwohl lassen sie innerhalb der Spaltenfamilie genügend Freiraum mit der möglichen Verwendung von beliebigen Spaltenschlüsseln.

7.4 Dokument-Datenbanken

Eine dritte Variante von NoSQL-Datenbanken, die *Dokument-Datenbanken* (engl. *document store*), vereinigen nun die Schemafreiheit von Schlüssel-Wert-Speichern mit der Möglichkeit zur Strukturierung der gespeicherten Daten. Anders als der Name impliziert,

speichern Dokument-Datenbanken nicht beliebige Dokumente wie Web-, Video- oder Audiodateien, sondern strukturierte Daten in Datensätzen, welche *Dokumente* genannt werden.

Die gängigen Dokument-Datenbanken wurden spezifisch für den Einsatz für Webdienste entwickelt. Dadurch sind sie mit Webtechnologien wie JavaScript oder HTTP[1] einfach integrierbar. Zudem sind sie auf einfache Weise horizontal skalierbar, indem verschiedene Rechner zu einem Gesamtsystem zusammengesetzt werden können, welche das Datenvolumen mit Sharding verteilen. Somit liegt der Fokus eher auf der Verarbeitung von großen Mengen heterogener Daten, während bei vielen Webdaten, beispielsweise im Bereich Social Media, Suchmaschinen oder News-Portalen, die Konsistenz der Daten nicht jederzeit gewährleistet sein muss. Eine Ausnahme sind hier sicherheitsrelevante Webdienste wie E-Banking, welche stark auf Schemarestriktionen und garantierte Konsistenz angewiesen sind.

Dokument-Datenbanken sind völlig schemafrei, das heisst, es ist nicht notwendig, vor dem Einfügen von Datenstrukturen ein Schema zu definieren. Die Schemaverantwortung wird somit dem Benutzer bzw. der verarbeitenden Applikation übergeben. Der Nachteil, der sich aus der Schemafreiheit ergibt, ist der Verzicht auf referenzielle Integrität und Normalisierung. Durch die Schemafreiheit ist aber eine extreme Flexibilität in der Speicherung unterschiedlichster Daten möglich, welche sich auf die Variety in den V's von Big Data (vgl. Abschn. 1.3) bezieht. Auch kann so die Fragmentierung der Datenbasis auf einfache Art und Weise erfolgen.

Dokument-Datenbanken sind auf der ersten Ebene eine Art Schlüssel-Wert-Datenbank. Zu einem beliebigen Schlüssel (die Dokument-ID) kann ein Datensatz als Wert gespeichert werden. Diese Datensätze werden *Dokumente* genannt. Auf der zweiten Ebene haben diese Dokumente nun eine *eigene interne Struktur*. Der Begriff des Dokuments ist insofern nicht ganz treffend, als es sich hierbei explizit nicht um Multimedia oder andere unstrukturierte Dokumente handelt. Ein Dokument im Sinne der Dokument-Datenbanken ist eine Datei mit *strukturierten Daten*, beispielsweise im JSON[2]-Format. Diese Struktur stellt eine Liste von Attribut-Wert-Paaren dar. Alle Attributwerte in dieser Datenstruktur können rekursiv wiederum Listen von Attribut-Wert-Paare enthalten. Die Dokumente haben untereinander keine Beziehung, sondern enthalten eine in sich abgeschlossene Sammlung von Daten.

In Abb. 7.3 wird als Beispiel eine Dokument-Datenbank D_USERS dargestellt, die Daten zu Benutzern einer Webseite speichert. Zu einem Benutzerschlüssel mit dem Attribut _id wird ein Dokument gespeichert, welches alle Daten zu einem Benutzer wie Benutzername (userName), Vorname (firstName), Nachname (lastName) und das Geschlecht (gender) speichert. Das Attribut visitHistory enthält einen verschachtelten Attributwert als assoziatives Array, welches selbst wieder Schlüssel-Wert-Paare aufweist.

[1] HyperText Transfer Protocol.

[2] JavaScript Object Notation.

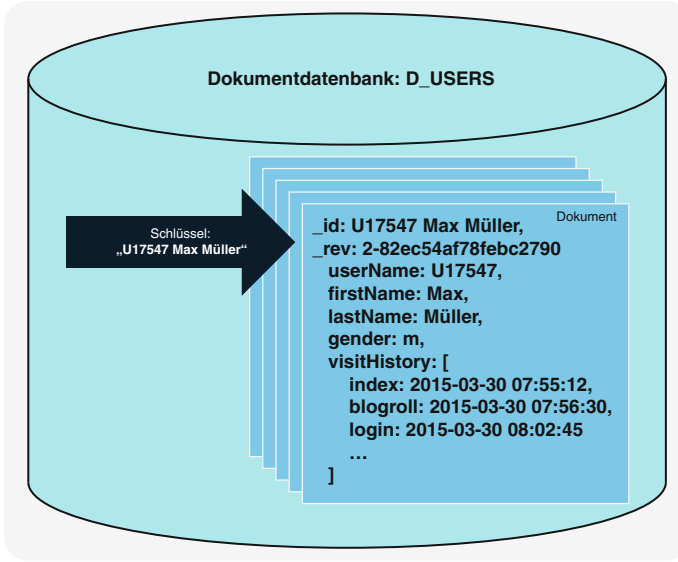

Abb. 7.3 Beispiel einer Dokumentdatenbank

In dieser verschachtelten Struktur wird zu einer Webseite als zugehöriger Wert der Zeitpunkt des letzten Besuchs aufgelistet.

Neben dem Standard-Attribut _id enthält das Dokument ein Feld _rev (revision), welches die Version des Dokuments indiziert. Eine Möglichkeit zur Auflösung konkurrenzierender Anfragen ist die *Multi-Version Concurrency Control*: Die Datenbank stellt sicher, dass jede Anfrage jeweils die Revision eines Dokuments mit der größten Anzahl Änderungen erhält. Da dadurch jedoch nicht eine vollständige transaktionale Sicherheit gewährleistet werden kann, spricht man in diesem Fall von schlussendlicher Konsistenz (engl. *eventual consistency*). Erst nach einer gewissen Zeit wird die Konsistenz der Daten erreicht. Dadurch kann die Datenverarbeitung auf Kosten der transaktionalen Sicherheit stark beschleunigt werden.

Dokument-Datenbank
Zusammengefasst ist eine Dokument-Datenbank oder Document Store ein Datenbankverwaltungssystem mit folgenden Eigenschaften:

- Sie ist eine Schlüssel-Wert Datenbank.
- Die gespeicherten Datenobjekte als Werte zu den Schlüsseln werden Dokumente genannt; die Schlüssel dienen der Identifikation.
- Die Dokumente enthalten Datenstrukturen in der Form von rekursiv verschachtelten Attribut-Wert-Paaren ohne referenzielle Integrität.
- Diese Datenstrukturen sind schemafrei, d. h. in jedem Dokument können beliebige Attribute verwendet werden, ohne diese zuerst in einem Schema zu definieren.

Anfragen an eine Dokument-Datenbank können mit *Map/Reduce-Verfahren* (Abschn. 5.4) parallelisiert und somit beschleunigt werden. Solche Verfahren sind zweiphasig, wobei Map in etwa der Gruppierung (group by) und Reduce der Aggregierung (z. B. count, sum, etc.) in SQL entspricht.

In der ersten Phase wird eine Map-Funktion ausgeführt, welche für jedes Dokument eine vordefinierte Verarbeitung vornimmt, welche einen Index (engl. *map*) aufbaut und retourniert. Eine solche Map ist ein assoziatives Array mit einem oder mehreren Schlüssel-Wert-Paaren pro Dokument. Die Phase Map kann pro Dokument unabhängig vom Rest der Datenbasis berechnet werden, wodurch bei der Verteilung der Datenbank auf mehrere Rechner immer eine Parallelverarbeitung ohne Abhängigkeiten möglich ist.

In der zweiten optionalen Reduce-Phase wird eine Funktion zur Reduzierung (engl. *reduce*) der Daten durchgeführt, welche pro Schlüssel im Index aus der Map-Funktion eine Zeile zurückgibt und deren zugehörige Werte aggregiert. Folgendes Beispiel zeigt, wie mit Map/Reduce die Anzahl Benutzer, gruppiert nach Geschlecht, in der Datenbank aus Abb. 7.3 berechnet wird.

Für jedes Dokument wird wegen der Schemafreiheit in der Map-Funktion geprüft, ob das Attribut userName vorhanden ist. Falls dem so ist, wird mit der Funktion `emit` ein Schlüssel-Wert Paar zurückgegeben: Als Schlüssel wird das Geschlecht des Benutzers und als Wert die Zahl 1 retourniert. Die Reduce-Funktion erhält anschließend im Array `keys` zwei verschiedene Schlüssel, `m` und `f`, und als Werte im Array `values` für jedes Dokument pro Benutzer mit dem entsprechenden Geschlecht die Zahl 1. Die Reduce-Funktion retourniert die Summe der Einsen, gruppiert nach dem Schlüssel, was der jeweiligen Anzahl entspricht.

```
// map
function(doc){
  if(doc.us
    emit(doc.gender, 1)
  }
}

// reduce
function(keys, values) {
return sum(values)
}

// >     key  value
// >     "f"  456
// >     "m"  567
```

Die Resultate von Map/Reduce-Berechnungen, sogenannte *Views*, sollten für eine optimale Performance als permanente Views mittels Design-Dokumenten vorberechnet und indexiert werden. Schlüssel-Wert-Paare werden in Dokument-Datenbanken in B-Bäumen (siehe Abschn. 5.2.1) gespeichert. Dadurch wird ein schneller Zugriff auf einzelne Schlüsselwerte gewährleistet. Die Reduce-Funktion macht sich die B-Baum-Struktur

zu Nutze, indem Aggregate in balancierten Bäumen gespeichert werden, wobei in den Blättern nur wenige Detailwerte gelagert sind. Bei Aktualisierungen von Aggregaten sind daher nur Änderungen des jeweiligen Blattes und der (wenigen) Knoten mit Teilsummen bis zur Wurzel notwendig.

7.5 XML-Datenbanken

Die Auszeichnungssprache XML (eXtensible Markup Language) wurde vom World Wide Web Consortium (W3C) entwickelt. Die Inhalte von Hypertextdokumenten werden wie bei HTML durch Tags markiert. Ein XML-Dokument ist selbstbeschreibend, da es neben den eigentlichen Daten auch Informationen über die Datenstruktur mitführt:

```
<Adresse>
<Straße> Rue Faucigny </Straße>
<Nummer> 2 </Nummer>
<Postleitzahl> 1700 </Postleitzahl>
<Ort> Fribourg </Ort>
</Adresse>
```

Die Grundbausteine von XML-Dokumenten nennt man Elemente. Diese bestehen aus einem Start-Tag (in spitzen Klammern < Name>) und einem End-Tag (in spitzen Klammern mit Schrägstrich </Name>), dazwischen steht der Inhalt des Elementes. Die Bezeichner des Start- und End-Tags müssen übereinstimmen.

Die Tags liefern Informationen über die Bedeutung der konkreten Werte und sagen somit etwas über die Datensemantik aus. Elemente können in XML-Dokumenten beliebig geschachtelt werden. Zur Darstellung solcher hierarchisch strukturierter Dokumente wird sinnvollerweise ein Graph verwendet; ein Beispiel ist in Abb. 7.4 gegeben.

Wie erwähnt, enthalten die XML-Dokumente implizit auch Informationen über die Struktur des Dokumentes. Da es für viele Anwendungen wichtig ist, die Struktur der XML-Dokumente zu kennen, sind explizite Darstellungen (DTD = Document Type Definition oder XML-Schema) von W3C vorgeschlagen worden. Mit einem expliziten Schema wird aufgezeigt, welche Tags im XML-Dokument auftreten und wie sie angeordnet sind. Damit lassen sich unter anderem Fehler in XML-Dokumenten lokalisieren und beheben. Da es für den Einsatz von Datenbanksystemen vorteilhaft ist, soll das XML-Schema hier illustriert werden.

Wir untersuchen nun die Frage, wie ein XML-Schema mit einem relationalen Datenbankschema zusammenhängt: Normalerweise können relationale Datenbankschemas durch eine dreistufige Verschachtelung von Elementen charakterisiert werden, und zwar

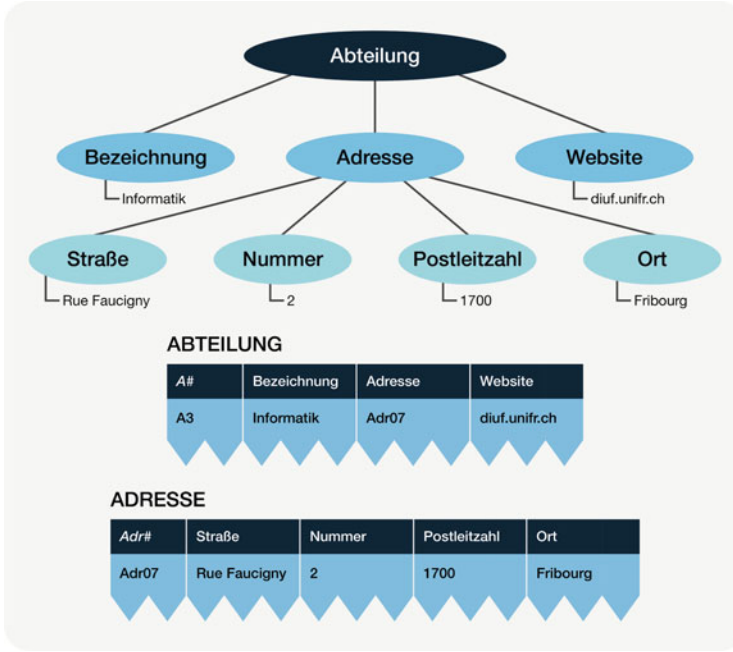

Abb. 7.4 Darstellung eines XML-Dokumentes in Tabellenform

der Bezeichnung der Datenbank, den Relationennamen sowie den Attributnamen. Damit können wir ein relationales Datenbankschema einem Ausschnitt eines XML-Schemas zuordnen und umgekehrt.

In Abb. 7.4 sehen wir die Zuordnung eines XML-Dokumentes zu einem relationalen Datenbankschema. Der Ausschnitt eines XML-Dokuments zeigt die beiden Relationennamen ABTEILUNG und ADRESSE, jeweils mit den zugehörigen Attributnamen resp. den konkreten Datenwerten. Die Verwendung von Schlüsseln und Fremdschlüsseln ist mit Hilfe eines XML-Schemas ebenfalls möglich, worauf wir kurz eingehen.

Das grundlegende Konzept von XML-Schemas ist, Datentypen zu definieren und über Deklarationen Namen zu den Datentypen zuordnen zu können. Dadurch lassen sich beliebige XML-Dokumente erstellen. Zudem besteht die Möglichkeit, Integritätsregeln für die Korrektheit von XML-Dokumenten zu beschreiben.

Es gibt eine Vielzahl von Standarddatentypen wie String, Boolean, Integer, Date, Time etc., daneben können aber auch benutzerdefinierte Datentypen eingeführt werden. Spezifische Eigenschaften der Datentypen lassen sich durch sogenannte Facets deklarieren. So kann die Ordnungseigenschaft eines Datentyps angegeben werden, beispielsweise die Beschränktheit der Werte durch Ober- und Untergrenzen, Längenbeschränkungen oder Aufzählung erlaubter Werte:

```
<xs:simpleType name=«Ort»>
<xs:restriction base=«xs:string»>
<xs:length value=«20»/>
</xs:restriction>
</xs:simpleType>
```

Zur Darstellung der Ortschaften wird ein einfacher Datentyp vorgeschlagen, der auf dem vordefinierten Datentyp String basiert. Zudem wird verlangt, dass die Ortsnamen nicht mehr als 20 Zeichen umfassen sollen.

Es sind verschiedene XML-Editoren entwickelt worden, die eine grafische Darstellung eines XML-Dokumentes resp. eines XML-Schemas erlauben. Diese Editoren können sowohl für die Deklaration der Struktureigenschaften sowie für die Erfassung von Dateninhalten verwendet werden. Durch das Ein- und Ausblenden von Teilstrukturen lassen sich umfangreiche XML-Dokumente resp. XML-Schemas übersichtlich anordnen.

Es ist wünschenswert, dass XML-Dokumente oder XML-Datenbanken ausgewertet werden können. Im Gegensatz zu relationalen Abfragesprachen werden Selektionsbedingungen nicht nur an Werte geknüpft (Wertselektion), sondern auch an Elementstrukturen (Strukturselektion). Weitere Grundoperationen einer XML-Abfrage betreffen die Extraktion von Subelementen eines XML-Dokumentes resp. das Verändern ausgewählter Subelemente. Auch lassen sich durch das Zusammensetzen von einzelnen Elementen aus unterschiedlichen Quellstrukturen neue Elementstrukturen erzeugen. Zu guter Letzt muss eine geeignete Abfragesprache auch mit Hyperlinks resp. Referenzen umgehen können; sogenannte Pfadausdrücke sind deshalb unentbehrlich.

XQuery ist von W3C vorgeschlagen worden, beeinflusst durch die Sprachen SQL, unterschiedliche XML-Sprachen (z. B. XPath als Navigationssprache von XML-Dokumenten) sowie objektorientierten Abfragesprachen. XQuery ist eine Erweiterung von XPath, wobei sie zusätzlich zum Abrufen von Daten in XML-Dokumenten die Möglichkeit zur Formung neuer XML-Strukturen bietet. Die Grundelemente von XQuery bilden *FOR-LET-WHERE-RETURN-Ausdrücke*: FOR und LET binden eine oder mehrere Variablen an die Ergebnisse der Auswertung von Ausdrücken. Mit der WHERE-Klausel können analog zu SQL weitere Einschränkungen an die Ergebnismenge vorgenommen werden. Das Ergebnis der Abfrage wird durch RETURN angezeigt.

Ein einfaches Beispiel soll das Grundkonzept von XQuery skizzieren. Es wird eine Anfrage an das XML-Dokument «Abteilung» (siehe Abb. 7.4) gestellt, dabei interessieren die Straßennamen sämtlicher Abteilungen:

```
<StraßenNamen>
{FOR $Abteilung IN //Abteilung RETURN
$Abteilung/Adresse/Straße }
</StraßenNamen>
```

Die obige Abfrage bindet die Variable $Abteilung im Laufe der Bearbeitung jeweils an die Knoten vom Typ < Abteilung>. Für jede dieser Bindungen wird durch den RETURN-Ausdruck die jeweilige Adresse evaluiert und die Straße ausgegeben. Die Anfrage in XQuery produziert das folgende Ergebnis:

```
<StraßenNamen>
<Straße> Rue Faucigny </Straße>
<Straße> . . . . . </Straße>
<Straße> . . . . . </Straße>
</StraßenNamen>
```

In XQuery werden Variablen mit einem durch das $-Zeichen ergänzten Namen eindeutig gekennzeichnet, um Namen von Elementen zu unterscheiden. Im Gegensatz zu einigen Programmiersprachen können Variablen in XQuery keine Werte zugewiesen werden. Vielmehr muss man Ausdrücke auswerten und das Ergebnis an die Variablen binden. Diese Variablenbindung erfolgt bei XQuery mit den FOR- und LET-Ausdrücken.

Im obigen Anfragebeispiel wird auf die Spezifikation des LET- Ausdrucks verzichtet. Mit der WHERE-Klausel liesse sich zudem die Ergebnismenge weiter einschränken. Die Auswertung der RETURN- Klausel erfolgt für jeden Schleifendurchlauf mit FOR, muss aber nicht zwingend ein Resultat liefern. Die einzelnen Resultate hingegen werden aneinandergereiht und bilden das Ergebnis des FOR-LET-WHERE-RETURN-Ausdrucks.

XQuery ist eine *mächtige Abfragesprache für Hyperdokumente* und wird sowohl für XML-Datenbanken wie auch für einige postrelationale Datenbanksysteme angeboten. Damit relationale Datenbanksysteme XML-Dokumente speichern können, müssen Erweiterungen in der Speicherungskomponente vorgenommen werden.

Viele relationale Datenbanksysteme verfügen heute über XML-Spaltendatentypen und somit über die Möglichkeit, mit XML direkt umzugehen: So können Daten in XML-Spalten strukturiert abgespeichert werden und mit XQuery oder XPath Elemente des XML-Baums direkt gesucht und verändert werden. Um die Jahrtausendwende waren XML-Dokumente für Datenspeicherung und -kommunikation stark im Aufwind und wurden für alles Mögliche, insbesondere auch für Webdienste, angewendet. In Zuge dieser Entwicklung entstand auch eine Reihe von Datenbanken, welche Daten direkt in der Form von XML-Dokumenten verarbeiten können. Gerade im Open Source-Bereich ist die Unterstützung von XQuery bei nativen XML-Datenbanken wesentlich mächtiger als bei relationalen Datenbanken.

Native XML-Datenbank
Eine native XML-Datenbank ist eine Datenbank, welche folgende Eigenschaften erfüllt:

- Die Daten werden in Dokumenten gespeichert. Sie ist also eine Dokument-Datenbank (siehe Abschn. 7.4).

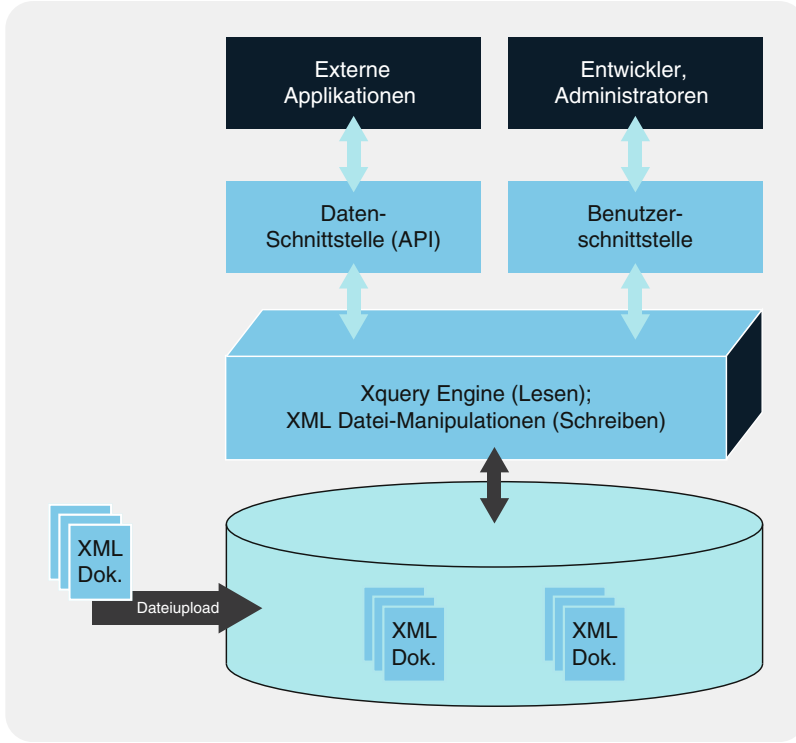

Abb. 7.5 Schema einer Native XML-Datenbank

- Die strukturierten Daten in den Dokumenten sind kompatibel mit dem XML-Standard.
- XML-Technologien wie XPath, XQuery und XSL/T können zur Abfrage und Manipulation der Daten angewendet werden.

Native XML-Datenbanken speichern Daten streng hierarchisch in einer Baumstruktur. Sie sind dann besonders gut geeignet, wenn hierarchische Daten in standardisiertem Format gespeichert werden sollen, beispielsweise bei Web-Services in der serviceorientierten Architektur (SoA). Ein großer Vorteil ist der vereinfachte Datenimport in die Datenbank, der bei einigen Datenbanksystemen durch simples Drag and Drop von XML-Dateien erreicht werden kann. Abbildung 7.5 zeigt eine schematische Darstellung einer Native XML-Datenbank. Sie vereinfacht es Benutzern und Programmen, auf Daten in einer Sammlung von XML-Dokumenten sowohl lesend als auch schreibend zuzugreifen.

Eine XML-Datenbank kann keine Querverweise zwischen gleichen Knoten herstellen. Dies kann insbesondere bei multidimensional verknüpften Daten problematisch sein. Geeignet ist eine XML-Datenbank daher vor allem für Daten, welche in einer Baumstruktur als Folge von verschachtelten Generalisierungen oder Aggregationen dargestellt werden können.

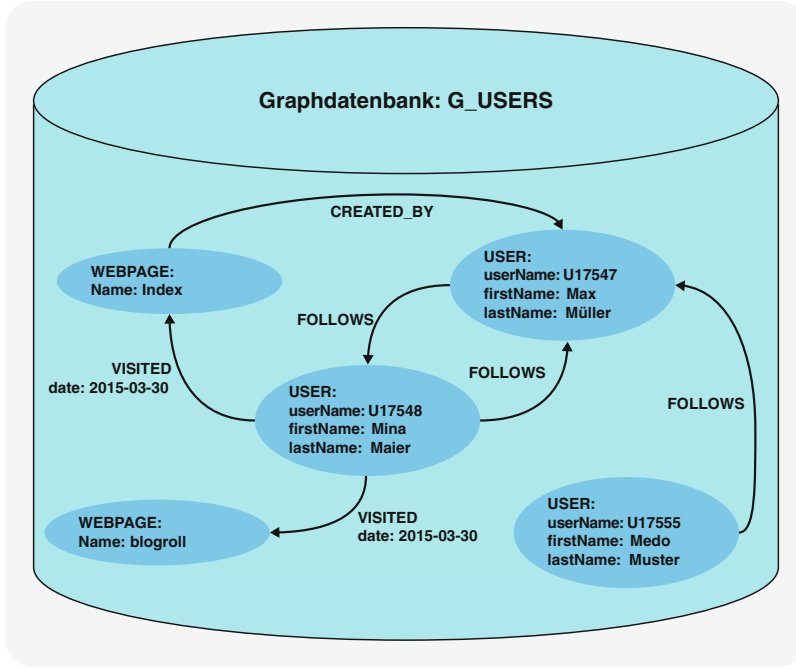

Abb. 7.6 Beispiel einer Graphdatenbank mit Benutzerdaten einer Webseite

7.6 Graphdatenbanken

Als vierte und letzte Vertretung der Gattung der Core-NoSQL-Datenbanken unterscheiden sich die Graphdatenbanken deutlich von den bisher besprochenen Datenmodellen der Schlüssel-Wert-Datenbanken, der Spaltenfamilien-Datenbanken und der Dokument-Datenbanken. Diese drei Datenmodelle verzichten zugunsten von einfacher Fragmentierung (Sharding) auf Datenbankschemas und auf referenzielle Integrität. Die Graphdatenbanken haben nun aber ein strukturierendes Schema und zwar dasjenige des Eigenschaftsgraphen, welches bereits im Abschn. 1.4.1 eingeführt wurde. In einer Graphdatenbank werden Daten also in Form von Knoten und Kanten gespeichert, welche zu einem Knotentyp bzw. zu einem Kantentyp gehören, und Daten in Form von Attribut-Wert-Paaren enthalten. Im Unterschied zu relationalen Datenbanken handelt es sich dabei aber um ein implizites Schema. Das heisst, dass Datenobjekte zu einem bisher nicht existierenden Knoten- oder Kantentyp direkt in die Datenbank eingefügt werden können, ohne diesen Typ vorher zu definieren. Das Datenbankverwaltungssystem vollzieht aus den vorhandenen Angaben implizit die entsprechende Schemaveränderung nach, d. h. es erstellt den entsprechenden Typ.

Als einführendes Beispiel zeigt Abb. 7.6 eine Graphdatenbank, G_USERS, welche Daten eines Webportals mit Benutzern, Webseiten und Beziehungen untereinander abbildet.

Wie in Abschn. 1.4.1 beschrieben, hat die Datenbank ein Schema mit Knoten- und Kantentypen. Es gibt zwei Knotentypen USER und WEBPAGE und drei Kantentypen FOLLOWS, VISITED und CREATED_BY. Der Knotentyp USER hat die Attribute userName, firstName und lastName; der Knotentyp WEBPAGE hat ein Attribut Name; und der Kantentyp VISITED hat ebenfalls ein Attribut, und zwar date mit dem Wertebereich Datum. Es handelt sich also um einen Eigenschaftsgraphen.

Im Prinzip speichert diese Graphdatenbank vom Typ ähnliche Daten wie die Dokument-Datenbank D_USERS in Abb. 7.3. Beispielsweise werden ebenfalls Benutzer mit Benutzernamen, Vornamen und Nachnamen sowie den besuchten Webseiten mit Datum abgebildet. Es besteht jedoch ein wesentlicher Unterschied: *Die Beziehungen zwischen den Datenobjekten sind explizit als Kanten vorhanden*, und die referenzielle Integrität wird vom Datenbankverwaltungssystem sichergestellt.

Graphdatenbank
Eine Graphdatenbank (engl. *graph database*) ist ein Datenbankverwaltungssystem mit folgenden Eigenschaften:

- Die Daten und/oder das Schema werden als *Graphen* (siehe Abschn. 2.4) oder graphähnlichen Strukturen abgebildet, welche das Konzept von Graphen generalisieren (z. B. Hypergraphen).
- Datenmanipulationen werden *als Graph-Transformationen* ausgedrückt, oder als Operationen, welche direkt typische Eigenschaften von Graphen ansprechen (z. B. Pfade, Nachbarschaften, Subgraphen, Zusammenhänge, etc.).
- Die Datenbank unterstützt die Prüfung von *Integritätsbedingungen*, welche die Datenkonsistenz sicherstellt. Die Definition von Konsistenz bezieht sich direkt auf Graph-Strukturen (z. B. Knoten- und Kantentypen, Attribut-Wertebereiche und referenzielle Integrität der Kanten).

Graphdatenbanken kommen überall dort zum Einsatz, wo Daten in Netzwerken organisiert sind. In diesen Fällen ist nicht der einzelne Datensatz wichtig, sondern die Verknüpfung der Datensätze untereinander, beispielsweise bei sozialen Medien, aber auch bei der Analyse von Infrastrukturnetzen (z. B. Wasser- oder Energieversorgung), beim Internet-Routing oder bei der Analyse der Verlinkung von Webseiten untereinander. Der Vorteil der Graphdatenbank ist die Eigenschaft der indexfreien Nachbarschaft (engl. *index-free adjacency*): Zu jedem Knoten kann das Datenbanksystem die direkten Nachbarn finden, ohne sämtliche Kanten berücksichtigen zu müssen, wie dies beispielsweise in relationalen Datenbanken unter Verwendung einer Beziehungstabelle der Fall wäre. Aus diesem Grund bleibt der Aufwand für die Abfrage von Beziehungen zu einem Knoten konstant, und zwar unabhängig vom Datenvolumen. Bei relationalen Datenbanken wächst der Aufwand für die Ermittlung von referenzierten Tupeln mit der Anzahl Tupel, auch wenn Indexe verwendet werden.

Gleich wie bei den relationalen Datenbanken brauchen Graphdatenbanken Indexe, um einen schnellen und direkten Zugriff auf einzelne Knoten und Kanten über ihre Eigenschaften bereitzustellen. Wie in Abschn. 5.2.1 dargestellt, werden für die Indexierung balancierte Bäume (B-Bäume) aufgebaut. Ein Baum ist nun ein Spezialfall eines Graphen, der keine Zyklen enthält; somit kann jeder Baum als Graph abgebildet werden. Dies ist bei den Graphdatenbanken interessant, weil somit der Index für einen Graphen selbst wiederum ein Subgraph dieses Graphen sein kann. Der Graph enthält seine eigenen Indexe.

Weniger einfach ist die Fragmentierung (siehe Abschn. 6.2) von Graphen. Ein Grund, weshalb bei den anderen Arten von Core-NoSQL-Datenbanken keine Beziehungen zwischen Datensätzen sichergestellt werden, ist, dass für die Fragmentierung (Sharding) die Datensätze unreflektiert auf verschiedene Rechner verteilt werden können, weil keine Abhängigkeiten bestehen. Bei Graphdatenbanken ist das Gegenteil der Fall. Beziehungen zwischen den Datensätzen sind das zentrale Element der Datenbank. Deshalb muss bei der Fragmentierung von Graphdatenbanken unbedingt auf die Zusammenhänge in den Daten Rücksicht genommen werden, wobei oft auch domänenspezifisches Wissen gefragt ist. Es gibt jedoch keine effiziente Methode, nach der ein Graph optimal in Teilgraphen zerlegt werden kann. Die existierenden Algorithmen sind NP-komplett, d. h. mit exponentiellem Aufwand verbunden. Als Heuristik können Clustering-Algorithmen stark vernetzte Teilgraphen als Partitionen ermitteln. Heutige Graphdatenbanken unterstützen daher das Sharding noch nicht.

7.7 Literatur

Edlich et al. (2011) stellen die Geschichte von NoSQL-Datenbanken dar. Die Definition des Begriffs NoSQL-Datenbank von Edlich et al. bilden die Grundlage der Definition in diesem Kapitel. Der Begriff Polyglot Persistence wird im Buch von Sadalage und Fowler (2013) erläutert.

Clustering und Replikation einer Spalten-Wert-Datenbank (Redis 2015) wird im Cluster-Tutorial auf der Website von Redis beschrieben. Das Datenmodell für Spaltenfamilien-Datenbanken ist in Sadalage und Fowler (2013) anschaulich dargestellt.

Google's BigTable Datenstruktur wurde ursprünglich von Chang et al. (2008) publiziert.

Exemplarisch für Dokument-Datenbanken liefern Anderson et al. (2010) einen Überblick auf CouchDB. Dort finden sich Erklärungen zur Datenstruktur, Views, Design-Dokumenten und Konsistenz in CouchDB, welche in diesem Kapitel prototypisch und prinzipiell verwendet werden. Für die Herleitung eines gruppierten Aggregats mit Map/Reduce konnte dankenswerter Weise auf einen Blogeintrag von Toby Ho (2009) zurückgegriffen werden.

XML-Datenbanken werden in Sadalage und Fowler (2013) vorgestellt. Ihre Definition wurde in diesem Kapitel sinngemäß übernommen. Zudem werden XML-Datenbanken in

McCreary und Kelly (2014) beschrieben. Xquery sowie eine detaillierte Anleitung der Verwendung von XML in relationalen Datenbanken und eine Anleitung für eine Native XML-Datenbank sind in Fawcett et al. (2012) zu finden.

Eine gute Referenz zur allgemeinen Verarbeitung von graphbasierten Daten findet sich in (Charu und Haixun 2010) oder in Robinson et al. (2013). Einen Überblick auf die bestehenden Graphdatenbank-Modelle geben Angles und Gutierrez (2008). Aus diesem Werk stammt auch die Definition der Graphdatenbank, welche hier verwendet wird. Edlich et al. (2011) stellen das Graphmodell detailliert dar. Aus dieser Quelle stammen die diesbezüglichen Informationen zu Anwendungsfällen, Indexierung und Partitionierung. Angaben zur Index-Free Adjacency Property und zur Komplexität von Algorithmen zum Sharding von Graphdatenbanken stammen von Montag (2013). Anwendungen zu NoSQL-Technologien finden sich in Fasel und Meier (2014, 2016).

Glossar

Abfragesprache Eine Abfragesprache erlaubt, Datenbanken durch die Angabe von Selektionsbedingungen eventuell mengenorientiert auszuwerten.

ACID ACID ist ein Kürzel und steht für Atomicity, Consistency, Isolation und Durability. Dieses Kürzel drückt aus, dass eine Transaktion immer einen konsistenten Zustand in einen konsistenten Zustand in der Datenbank überführt.

Aggregation Aggregation ist das Zusammenfügen von Entitätsmengen zu einem Ganzen. Die Aggregationsstruktur kann netzwerkartig oder hierarchisch (Stückliste) sein.

Anomalie Anomalien sind von der Realität abweichende Sachverhalte, die bei Einfüge-, Änderungs- und Löschoperationen in einer Datenbank entstehen können.

Assoziation Unter einer Assoziation von einer Entitätsmenge zu einer zweiten versteht man die Bedeutung der Beziehung in dieser Richtung. Jede Assoziation kann durch einen Assoziationstyp gewichtet werden, der die Mächtigkeit der Beziehungsrichtung angibt.

BASE BASE steht für Basically Available, Soft State und Eventually Consistent und sagt aus, dass ein konsistenter Zustand in einem verteilten Datenbanksystem eventuell verzögert erfolgt.

Baum Ein Baum ist eine Datenstruktur, bei der jeder Knoten auβer dem Wurzelknoten genau einen Vorgängerknoten besitzt und bei dem zu jedem Blatt ein eindeutiger Weg zur Wurzel existiert.

Big Data Unter Big Data versteht man Datenbestände, die mindestens die folgenden drei charakteristischen V's aufweisen: umfangreicher Datenbestand im Tera- bis Zettabytebereich (Volume), Vielfalt von strukturierten, semi-strukturierten und unstrukturierten Datentypen (Variety) sowie hohe Geschwindigkeit in der Verarbeitung von Data Streams (Velocity).

Business Intelligence Business Intelligence oder BI ist ein unternehmensweites Konzept für die Analyse resp. das Reporting von relevanten Unternehmensdaten.

CAP-Theorem Das CAP-Theorem (C=Consistency, A=Availability, P=Partition Tolerance) sagt aus, dass in einem massiv verteilten Datenhaltungssystem jeweils nur zwei Eigenschaften aus den drei der Konsistenz (C), Verfügbarkeit (A) und Ausfalltoleranz (P) garantiert werden können.

Cursorverwaltung Die Cursorverwaltung ermöglicht, mit Hilfe eines Zeigers, eine Menge von Datensätzen satzweise zu verarbeiten.

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2

Cypher Cypher ist eine deklarative Abfragesprache für die Graphdatenbank Neo4j.

Data-Dictionary-System Ein Data-Dictionary-System dient der Beschreibung und Dokumentation von Datenelementen, Datenbankstrukturen, Transaktionen etc. sowie deren Verknüpfungen untereinander.

Data Mining Data Mining bedeutet das Schürfen nach wertvollen Informationen in den Datenbeständen und bezweckt, noch nicht bekannte Datenmuster zu erkennen.

Data Scientist Ein Data Scientist ist eine Spezialistin oder ein Spezialist des Business Analytics und beherrscht die Methoden und Werkzeuge von NoSQL-Datenbanken, des Data Minings sowie der Statistik und Visualisierung mehrdimensionaler Zusammenhänge innerhalb der Daten.

Data Stream Ein Datenstrom ist ein kontinuierlicher Fluss von digitalen Daten, wobei die Datenrate (Datensätze pro Zeiteinheit) variieren kann. Die Daten eines Data Streams sind zeitlich geordnet, wobei neben Audio- und Video-Daten auch Messreihen darunter aufgefasst werden.

Data Warehouse Ein Data Warehouse ist ein System von Datenbanken und Ladeprogrammen, welches historisierte Daten verschiedener verteilter Datenbestände via Integration für die Datenanalyse zur Verfügung stellt.

Datenarchitektur In der Datenarchitektur wird festgelegt, welche Daten in Eigenregie gesammelt oder über Informationsbroker bezogen, welche Datenbestände zentral resp. dezentral gehalten, wie Datenschutz und Datensicherheit gewährleistet werden und wer für den Unterhalt und die Pflege der Datenbestände die Verantwortung übernimmt.

Datenbankschema Unter einem relationalen Datenbankschema versteht man die formale Spezifikation der Datenbanken und Tabellen unter Angabe von Schlüssel- und Nichtschlüsselmerkmalen sowie von Integritätsbedingungen.

Datenbanksystem Ein Datenbanksystem besteht aus einer Speicherungs- und einer Verwaltungskomponente. Die Speicherungskomponente erlaubt, Daten und Beziehungen abzulegen; die Verwaltungskomponente stellt Funktionen und Sprachmittel zur Pflege und Verwaltung der Daten zur Verfügung.

Datenmanagement Unter dem Datenmanagement fasst man alle betrieblichen, organisatorischen und technischen Funktionen der Datenarchitektur, der Datenadministration und der Datentechnik zusammen, die der unternehmensweiten Datenhaltung, Datenpflege, Datennutzung sowie dem Business Analytics dienen.

Datenmodell Ein Datenmodell beschreibt auf strukturierte und formale Art die für ein Informationssystem notwendigen Daten und Datenbeziehungen.

Datenschutz Unter Datenschutz versteht man den Schutz der Daten vor unbefugtem Zugriff und Gebrauch.

Datensicherheit Bei der Datensicherheit geht es um technische Vorkehrungen gegen Verfälschung, Zerstörung oder Verlust von Datenbeständen.

Datenunabhängigkeit Bei Datenbanksystemen spricht man von Datenunabhängigkeit, wenn die Daten von den Anwendungsprogrammen mittels Systemfunktionen getrennt bleiben.

Dokument-Datenbank Eine Dokument-Datenbank ist eine NoSQL-Datenbank, welche wie ein Schlüssel-Wert-Speicher zu jedem Schlüssel einen Datensatz speichert. Diese

Datensätze enthalten aber eine Menge strukturierter Daten in Form von Attributen und Merkmalen, weshalb sie Dokumente genannt werden.

Endbenutzer Ein Endbenutzer ist eine Anwenderin oder ein Anwender in der Fachabteilung des Unternehmens, die oder der Grundkenntnisse in Informatik besitzt.

Entität Entitäten entsprechen Objekten der realen Welt oder unserer Vorstellung. Sie werden durch Merkmale charakterisiert und zu Entitätsmengen zusammengefasst.

Entitäten-Beziehungsmodell Das Entitäten-Beziehungsmodell ist ein Datenmodell, das Datenklassen (Entitätsmengen) und Beziehungsmengen freilegt. Entitätsmengen werden grafisch durch Rechtecke, Beziehungsmengen durch Rhomben und Attribute durch Ellipsen dargestellt.

Fuzzy-Datenbank Eine Fuzzy-Datenbank unterstützt unvollständige, vage oder unpräzise Sachverhalte durch Anwendung der unscharfen Logik.

Graphdatenbank Eine Graphdatenbank verwaltet Graphen, d.h. Kanten (Objekte oder Konzepte) und Beziehungen (Beziehungen zwischen Objekten oder Konzepten), wobei Knoten wie Kanten attributiert sein können.

Graphenmodell Ein Graphenmodell beschreibt die Sachverhalte der Realität oder unserer Vorstellung anhand von Knoten (Objekte) und Kanten (Beziehungen zwischen den Objekten). Knoten und Kanten können Eigenschaften aufweisen. Kanten können ungerichtet oder gerichtet sein.

Generalisation Unter Generalisation versteht man das Verallgemeinern von Entitätsmengen zu einer übergeordneten Entitätsmenge; die Subentitätsmengen der Generalisationshierarchie werden Spezialisierungen genannt.

Hashing Hashing ist eine gestreute Speicherorganisation, bei der aufgrund einer Transformation (Hash-Funktion) aus den Schlüsseln direkt die zugehörigen Adressen der Datensätze berechnet werden.

Index Ein Index ist eine physische Datenstruktur, die für ausgewählte Merkmale die internen Adressen der Datensätze liefert.

InMemory-Datenbank Bei der InMemory-Datenbank werden die Datensätze im Hauptspeicher des Rechners gehalten.

Integritätsbedingung Integritätsbedingungen sind formale Spezifikationen über Schlüssel, Merkmale und Wertebereiche. Sie dienen dazu, die Widerspruchsfreiheit der Daten zu gewährleisten.

Map/Reduce-Verfahren Das Map/Reduce-Verfahren besteht aus zwei Phasen: In der Map-Phase werden Teilaufgaben an diverse Knoten des Rechnernetzes verteilt, um Parallelität für die Berechnung von Zwischenresultaten auszunutzen. Die Reduce-Phase fasst die Zwischenresultate zusammen.

Normalform Normalformen sind Regeln, mit denen innerhalb von Tabellen Abhängigkeiten freigelegt werden können, zur Vermeidung redundanter Informationen und damit zusammenhängender Anomalien.

NoSQL NoSQL bedeutet ‚Not only SQL' und charakterisiert Datenbanken, die Big Data unterstützen und keinem fixen Datenbankschema unterworfen sind. Zudem ist das zugrundeliegende Datenmodell nicht relational.

Nullwert Ein Nullwert ist ein Datenwert, der dem Datenbanksystem zur Zeit noch nicht bekannt ist.

Objektorientierung Bei der Objektorientierung werden die Daten durch geeignete Methoden gekapselt. Zudem lassen sich Eigenschaften von Datenklassen vererben.

Optimierung Unter der Optimierung einer Datenbankabfrage versteht man das Umformen des entsprechenden Ausdrucks (z. B. algebraische Optimierung) sowie das Ausnutzen von Speicher- und Zugriffsstrukturen zwecks Reduktion des Berechnungsaufwandes.

QBE QBE (Query by Example) ist eine Datenbanksprache, bei der die Benutzer ihre Auswertungswünsche durch Beispiele entwerfen und ausführen lassen.

Recovery Recovery bedeutet das Wiederherstellen eines korrekten Datenbankzustandes nach einem Fehlerfall.

Redundanz Die mehrfache Speicherung desselben Sachverhaltes in einer Datenbank wird als Redundanz bezeichnet.

Relationenalgebra Die Relationenalgebra bildet den formalen Rahmen für die relationalen Datenbanksprachen. Sie setzt sich aus den Operatoren Vereinigung, Subtraktion, kartesisches Produkt, Projektion und Selektion zusammen.

Relationenkalkül Der Relationenkalkül basiert auf der Aussagenlogik, wobei neben der logischen Verknüpfung von Prädikaten auch Quantoren („für alle gilt . . .“ oder „es existiert . . .“) zugelassen sind.

Relationenmodell Das Relationenmodell ist ein Datenmodell, das sowohl Daten als auch Datenbeziehungen in Form von Tabellen ausdrückt.

Schlüssel Ein Schlüssel ist eine minimale Merkmalskombination, die Datensätze innerhalb einer Datenbank eindeutig identifiziert.

Schlüssel-Wert-Datenbank Eine Schlüssel-Wert-Datenbank ist eine NoSQL-Datenbank, welche die Daten als Schlüssel-Wert-Paare ablegt.

Selektion Die Selektion ist eine Datenbankoperation, die aufgrund einer benutzerspezifizierten Bedingung die entsprechenden Tupel einer Tabelle bereitstellt.

Spaltenfamilien-Datenbank Eine Spaltenfamilien-Datenbank ist eine NoSQL-Datenbank, bei welcher die Daten in Spalten und/oder Mengen von Spalten organisiert werden.

SQL SQL (Structured Query Language) ist die wichtigste relationale Abfrage- und Manipulationssprache; sie wurde durch die ISO (International Organization for Standardization) normiert.

Synchronisation Beim Mehrbenutzerbetrieb versteht man unter der Synchronisation die Koordination gleichzeitiger Zugriffe auf eine Datenbank. Bei der pessimistischen Synchronisation werden Konflikte parallel ablaufender Transaktionen von vornherein verhindert, bei der optimistischen werden konfliktträchtige Transaktionen im Nachhinein zurückgesetzt.

Tabelle Eine Tabelle (Relation) ist eine Menge von Tupeln (Datensätzen) bestimmter Merkmalskategorien, wobei ein Merkmal oder eine Merkmalskombination die Tupel innerhalb der Tabelle eindeutig identifiziert.

Transaktion Eine Transaktion ist eine Folge von Operationen, die atomar, konsistent, isoliert und dauerhaft ist. Die Transaktionenverwaltung dient dazu, mehreren Benutzern ein konfliktfreies Arbeiten zu ermöglichen.

Vektoruhr Vektoruhren sind keine Zeituhren, sondern Zählsysteme, die eine Halbordnung unter der zeitlichen Abfolge von Ereignissen bei konkurrierenden Prozessen ermöglichen.

Verbund Ein Verbund ist eine Datenbankoperation, die zwei Tabellen über ein gemeinsames Merkmal verbindet und eine Resultattabelle erzeugt.

XML Die Auszeichnungssprache XML (eXtensible Markup Language) beschreibt semistrukturierte Daten, Inhalt und Form, auf hierarchische Art und Weise.

Zweiphasen-Sperrprotokoll Das Zweiphasen-Sperrprotokoll untersagt es einer Transaktion, nach dem ersten Entsperren eines Datenbankobjektes eine weitere Sperre anzufordern.

Literatur

Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide. O'Reilly. http://guide.couchdb.org/editions/1/en/index.html. Zugegriffen am 23.03.2015

Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1), 1–39 (2008)

Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., Eswaran, K.P., Gray, J.N., Griffiths, P.P., King, W.F., Lorie, R.A., McJones, P.R., Mehl, J.W., Putzolu, G.R., Traiger, I.L., Wade, B.W., Watson, V.: System R – relational approach to database management. ACM Trans. Database Syst. **1**(2), 97–137 (1976)

Aurenhammer, F.: Voronoi diagrams – a survey of a fundamental geometric data structure. ACM Comput. Surv. **23**(3), 345–405 (1991)

Balzert, H. (Hrsg.): CASE-Systeme und Werkzeuge. BI Wissenschaftsverlag, Mannheim (1993)

Balzert, H.: Lehrbuch der Objektmodellierung – Analyse und Entwurf. Spektrum Akademischer Verlag (2004)

Basta, A., Zgola, M.: Database Security. Cengage Learning (2011)

Bayer, R.: Symmetric binary B-trees: data structures and maintenance algorithms. Acta Informatica **1**(4), 290–306 (1992)

Beaulieu, A.: Einführung in SQL. O'Reilly (2006)

Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison Wesley (1987)

Berson, A., Smith, S.: Data Warehousing, Data Mining and OLAP. McGraw-Hill (1997)

Bertino, E., Martino, L.: Object-Oriented Database Systems. Addison Wesley (1993)

Biethahn, J., Mucksch, H., Ruf, W.: Ganzheitliches Informationsmanagement (Bd. II: Daten- und Entwicklungsmanagement). Oldenbourg (2000)

Blaha, M., Rumbaugh, J.: Object Oriented Modelling and Design with UML. Prentice-Hall (2004)

Booch, G.: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings (2006)

Bordogna, G., Pasi, G. (Hrsg.): Recent Issues on Fuzzy Databases. Physica-Verlag (2000)

Bosc, P., Kacprzyk, J. (Hrsg.): Fuzzyness in Database Management Systems. Physica-Verlag (1995)

Brewer, E.: Keynote – towards robust distributed systems. In: 19th ACM Symposium on Principles of Distributed Computing, Portland, 16–19 July (2000)

Brown, K.Q.: Geometric transformations for fast geometric algorithms. Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh (1979)

Brüderlin, B., Meier, A.: Computergrafik und Geometrisches Modellieren. Teubner (2001)

Castano, S., Fugini, M.G., Martella, G., Samarati, P.: Database Security. Addison Wesley (1994)

Cattell, R.G.G.: Object Data Management – Object-Oriented and Extended Relational Database Systems. Addison Wesley (1994)

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2

Celko, J.: Joe Celko's Complete Guide to NoSQL – What Every SQL Professional Needs to Know About Nonrelational Databases. Morgan Kaufmann (2014)
Ceri, S., Pelagatti, G.: Distributed Databases – Principles and Systems. McGraw-Hill (1985)
Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, D., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 1–26 (2008). Article No. 4
Charu, A., Haixun, W.: Managing and Mining Graph Data, Bd. 40. Springer (2010)
Chen, P.P.-S.: The entity-relationship model – towards a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)
Chen, G.: Design of fuzzy relational databases based on fuzzy functional dependencies. Ph.D. thesis Nr. 84, Leuven (1992)
Chen, G.: Fuzzy Logic in Data Modeling – Semantics, Constraints, and Database Design. Kluwer (1998)
Claus, V., Schwill, A.: Duden Informatik. Ein Fachlexikon für Studium und Praxis, 3. Aufl. Dudenverlag (2001)
Clifford, J., Warren, D.S.: Formal semantics for time in databases. ACM Trans. Database Syst. **8**(2), 214–254 (1983)
Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer (1994)
Coad, P., Yourdon, E.: Object-Oriented Design. Yourdon Press (1991)
Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
Connolly, T., Begg, C.: Database Systems – A Practical Approach to Design, Implementation and Management. Addison-Wesley (2014)
Cremers, A.B., et al.: Deduktive Datenbanken – Eine Einführung aus der Sicht der logischen Programmierung. Vieweg (1994)
Dadam, P.: Verteilte Datenbanken und Client/Server-Systeme. Springer (1996)
Darwen, H., Date, C.J.: A Guide to the SQL Standard. Addison Wesley (1997)
Date, C.J.: An Introduction to Database Systems. Addison-Wesley (2004)
Dean, J., Ghemawat, S.: MapReduce – simplified data processing on large clusters. In: Proceedings of the 6th Symposium on Operating Systems, Design and Implementation (OSDI'04), S. 137–150. San Francisco, 6–8 Dec 2004
DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo – Amazon's highly available key-value store. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07), S. 205–220. Stevenson, 14–17 Oct 2007
Diestel, R.: Graphentheorie. Springer (2006)
Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**, 269–271 (1959)
Dippold, R., Meier, A., Schnider, W., Schwinn, K.: Unternehmensweites Datenmanagement – Von der Datenbankadministration bis zum Informationsmanagement. Vieweg (2005)
Dittrich, K.R. (Hrsg.): Advances in Object-Oriented Database Systems. Lecture Notes in Computer Science, Bd. 334. Springer (1988)
Dutka, A.F., Hanson, H.H.: Fundamentals of Data Normalization. Addison-Wesley (1989)
Edlich, S., Friedland, A., Hampe, J., Brauer, B., Brückner, M.: NoSQL – Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken. Carl Hanser Verlag (2011)
Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley (2015)
Eswaran, K.P., Gray, J., Lorie, R.A., Traiger, I.L.: The notion of consistency and predicate locks in a data base system. Commun. ACM **19**(11), 624–633 (1976)

Etzion, O., Jajodia, S., Sripada, S. (Hrsg.): Temporal Databases – Research and Practice. Lecture Notes in Computer Science. Springer (1998)
Fagin, R.: Normal forms and relational database operators. In: Proceedings of the International Conference on Management of Data, SIGMOD'79, ACM, New York, S. 153–160. (1979)
Fasel, D., Meier, A. (Hrsg.): Big Data. Praxis der Wirtschaftsinformatik, HMD Nr. 298, Bd. 51, Heft 4 (2014)
Fasel, D., Meier A., (Hrsg.): Big Data – Grundlagen, Systeme und Nutzungspotenziale. Edition HMD. Springer (2016)
Fawcett, J., Quin, L.R.E., Ayers, D.: Beginning XML. Wiley (2012)
Ferstl, O.K., Sinz, E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). Wirtschaftsinformatik **33**(6), 477–491 (1991)
Findler, N.V. (Hrsg.): Associative Networks – Representation and Use of Knowledge by Computers. Academic (1979)
Freiknecht, J.: Big Data in der Praxis – Lösungen mit Hadoop, HBase und Hive – Daten speichern, aufbereiten und visualisieren. Carl Hanser Verlag (2014)
Gadia, S.K.: A homogeneous relational model and query languages for temporal databases. ACM Trans. Database Syst. **13**(4), 418–448 (1988)
Gallaire, H., et al.: Logic and databases: a deductive approach. ACM Comput. Surv. **16**(2), 153–185 (1984)
Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems – The Complete Book. Pearson Education Limited (2014)
Gardarin, G., Valduriez, P.: Relational Databases and Knowledge Bases. Addison Wesley (1989)
Geppert, A.: Objektrelationale und objektorientierte Datenbankkonzepte und -systeme. dpunkt (2002)
Gifford, D.K.: Weighted voting for replicated data. In: Proceedings of the seventh ACM Symposium on Operating Systems Principles (SOSP'79), S. 150–162. Pacific Grove, 10–12 Dec (1979)
Gilbert, S., Lynch, N.: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. Massachusetts Institute of Technology, Cambridge (2002)
Gluchowski, P., Gabriel, R., Chamoni, P.: Management Support Systeme und Business Intelligence – Computergestützte Informationssysteme für Fach- und Führungskräfte. Springer (2008)
Gray, J., Reuter, A.: Transaction Processing – Concepts and Techniques. Morgan Kaufmann (1993)
Härder, T.: Implementierung von Datenbanksystemen. Hanser (1978)
Härder, T., Rahm, E.: Datenbanksysteme – Konzepte und Techniken der Implementierung. Springer (2001)
Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15** (4), 287–317 (1983)
He, H., Singh, A. K.: Query language and access methods for graph databases. In: Aggarwal, C.C., Wang, H. (Hrsg.) Managing and Mining Graph Data, S. 125–160. Springer (2010)
Heinrich, L.J., Lehner, F.: Informationsmanagement – Planung, Überwachung und Steuerung der Informationsinfrastruktur. Oldenbourg (2005)
Heuer, A.: Objektorientierte Datenbanken. Addison Wesley (1997)
Hitz, M., Kappel, G., Kapsammer, E., Retschitzegger, W.: UML@Work – Objektorientierte Modellierung mit UML2. dpunkt (2005)
Ho, T.: Taking an Average in CouchDB. http://tobyho.com/2009/10/07/taking-an-average-in-couchdb/ (2009). Zugegriffen am 23.03.2015
Hoffer, I.A., Prescott, M.B., Toppi, H.: Modern Database Management. Prentice Hall (2012)
Hughes, J.G.: Object-Oriented Databases. Prentice-Hall (1991)
Inmon, W.H.: Building the Data Warehouse. Wiley (2005)

Jarke, M., Lenzerini, M., Vassiliou, Y., Vassiliadis, P.: Fundamentals of Data Warehouses. Springer (2000)

Kacprzyk, J., Zadrozny, S.: FQUERY for access – fuzzy querying for a windows-based DBMS. In: Bosc, P., Kacprzyk, J. (Hrsg.) Fuzzyness in Database Management Systems, S. 415–433. Physica-Verlag (1995)

Karger, D., Lehmann, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees – distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing, El Paso (1997)

Kemper, A., Eickler, A.: Datenbanksysteme – Eine Einführung. Oldenbourg (2013)

Kerre, E.E., Chen, G.: An overview of fuzzy data models. In: Bosc, P., Kacprzyk, J. (Hrsg.) Fuzzyness in Database Management Systems. Physica-Verlag, S. 23–41 (1995)

Kim, W.: Introduction to Object-Oriented Databases. The MIT Press (1990)

Kimball, R., Ross, M., Thorntwaite, W., Mundy, J., Becker, B.: The Datawarehouse Lifecycle Toolkit. Wiley (2008)

Kuhlmann, G., Müllmerstadt, F.: SQL – Der Schlüssel zu relationalen Datenbanken. Rowohlt (2004)

Lang, S.M., Lockemann, P.C.: Datenbankeinsatz. Springer (1995)

Lausen, G., Vossen, G.: Objekt-orientierte Datenbanken: Modelle und Sprachen. Oldenbourg (1996)

Liebling, T.M., Pournin, L.: Voronoi Diagrams and Delaunay Triangulations – Ubiquitous Siamese Twins. Documenta Mathematica – Extra Volume ISMP, Deutsche Mathematiker-Vereinigung (DMV), Berlin S. 419–431. (2012)

Lockemann, P.C., Schmidt, J.W. (Hrsg.): Datenbank-Handbuch. Springer (1993)

Lorie, R.A., Kim, W., McNabb, D., Plouffe, W., Meier, A.: Supporting complex objects in a relational system for engineering databases. In: Kim, W., et al. (Hrsg.) Query Processing in Database Systems, S. 145–155. Springer (1985)

Maier, D.: The Theory of Relational Databases. Computer Science Press (1983)

Maier, D., Warren, D.S.: Computing with Logic – Logic Programming with Prolog. Benjamin/Cummings (1988)

Marcus, D.A.: Graph Theory – A Problem Oriented Approach. The Mathematical Association of America (2008)

Martin, J.: Information Engineering – Planning and Analysis. Prentice-Hall (1990)

Martin, J., Odell, J.J.: Object-Oriented Analysis and Design. Prentice-Hall (1992)

Maurer, W.D., Lewis, T.G.: Hash table methods. ACM Comput. Surv. **7**(1), 5–19 (1975)

McCreary, D., Kelly, A.: Making Sense of NoSQL – A Guide for Managers and the Rest of Us. Manning (2014)

Meier, A.: Erweiterung relationaler Datenbanksystems für technische Anwendungen. Informatik-Fachberichte, Bd. 135. Springer (1987)

Meier, A.: Ziele und Aufgaben im Datenmanagement aus der Sicht des Praktikers. Wirtschaftsinformatik **36**(5), 455–464 (1994)

Meier, A. (Hrsg.): WWW & Datenbanken. Praxis der Wirtschaftsinformatik, Jhrg. 37, Nr. 214, dpunkt (2000)

Meier, A., Johner, W.: Ziele und Pflichten der Datenadministration. Theorie und Praxis der Wirtschaftsinformatik **28**(161), 117–131 (1991)

Meier, A., Wüst, T.: Objektorientierte und objektrelationale Datenbanken – Ein Kompass für die Praxis. dpunkt (2003)

Meier, A., Graf, H., Schwinn, K.: Ein erster Schritt zu einem globalen Datenmodell. Inf. Management **6**(2), 42–48 (1991)

Meier, A., Werro, N., Albrecht, M., Sarakinos, M.: Using a fuzzy classification query language for customer relationship management. In: Proceedings of the 31st International Conference on Very Large Databases (VLDB), S. 1089–1096. Trondheim (2005)
Meier, A., Schindler, G., Werro, N.: Fuzzy classification on relational databases (Chapter XXIII). In: Galindo, J. (ed.) Handbook of Research on Fuzzy Information Processing in Databases, Bd. II, S. 586–614. IGI Global (2008)
Melton, J., Simon, A.R.: SQL1999 – Understanding Relational Language Components. Morgan Kaufmann (2002)
Montag, D.: Understanding Neo4j Scalability. White Paper, netechnology. http://info.neo4j.com/rs/neotechnology/images/Understanding%20Neo4j%20Scaability(2).pdf (2013). Zugegriffen am 25.03.2015
Mucksch, H., Behme, W. (Hrsg.): Das Data-Warehouse-Konzept – Architektur, Datenmodelle, Anwendungen. Gabler (2000)
Myrach, T.: Temporale Datenbanken in betrieblichen Informationssystemen – Prinzipien, Konzepte, Umsetzung. Teubner (2005)
Nievergelt, J., Hinterberger, H., Sevcik, K.C.: The grid file: an adaptable symmetric multikey file structure. ACM Trans. Database Syst. **9**(1), 38–71 (1984)
Olle, T.W., et al.: Information Systems Methodologies – A Framework for Understanding. Addison Wesley (1988)
Ortner, E., Rössner, J., Söllner, B.: Entwicklung und Verwaltung standardisierter Datenelemente. Informatik-Spektrum **13**(1), 17–30 (1990)
Österle, H., Brenner, W., Hilbers, K.: Unternehmensführung und Informationssystem – Der Ansatz des St. Galler Informationssystem-Managements. Teubner (1991)
Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems. Prentice Hall (1991)
Panny, W., Taudes, A.: Einführung in den Sprachkern SQL-99. Springer (2000)
Panzarino, O.: Learning Cypher. Packt Publishing, Birmingham (2014)
Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: The Structure of the Relational Database Model. Springer (1989)
Petra, F.E.: Fuzzy Databases – Principles and Applications. Kluwer (1996)
Pistor, P.: Objektorientierung in SQL3: Standard und Entwicklungstendenzen. Informatik-Spektrum **16**(2), 89–94 (1993)
Pons, O., Vila, M.A., Kacprzyk, J. (Hrsg.): Knowledge Management in Fuzzy Databases. Physica-Verlag (2000)
Rahm, E.: Mehrrechner-Datenbanksysteme. Addison Wesley (1994)
Redis: Redis Cluster Tutorial. http://redis.io/topics/cluster-tutorial (2015). Zugegriffen am 02.03.2015
Redmond, E., Wilson, J.R.: Seven Databases in Seven Weeks – A Guide to Modern Databases and the NoSQL Movement. The Pragmatic Bookshelf (2012)
Reuter, A: Fehlerbehandlung in Datenbanksystemen. Hanser (1981)
Riak: Open Source Distributed Database. Siehe http://basho.com/riak/ (2014). Zugegriffen am 18.12.2014
Robinson, I., Webber, J., Eifrem, E.: Graph Databases. O'Reilly and Associates, Cambridge (2013)
Rothnie, J.B., et al.: Introduction to a system for distributed databases. ACM Trans. Database Syst. **5** (1), 1–17 (1980)
Saake, G., Sattler, K.-H., Heuer, A.: Datenbanken – Konzepte und Sprachen. mitp (2013)
Sadalage, P.J., Fowler, M.: NoSQL Distilled – A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley (2013)
Sauer, H.: Relationale Datenbanken – Theorie und Praxis inklusive SQL-2. Addison Wesley (2002)
Schaarschmidt, R.: Archivierung in Datenbanksystemen – Konzept und Sprache. Teubner (2001)

Scheer, A.-W.: Architektur integrierter Informationssysteme – Grundlagen der Unternehmensmodellierung. Springer (1997)
Schek, H.-J., Scholl, M.H.: The relational model with relation-valued attributes. Inf. Syst. **11**(2), 137–147 (1986)
Schindler, G.: Fuzzy Datenanalyse durch kontextbasierte Datenbankabfragen. Deutscher Universitäts-Verlag (1998)
Schlageter, G., Stucky, W.: Datenbanksysteme – Konzepte und Modelle. Teubner (1983)
Schwarz, R., Mattern, F.: Detecting causal relationships in distributed computations: in search of the Holy Grail. Distrib. Comput. **7**(3), 149 (1994)
Shamos, M.I.: Computational geometry. Ph.D. thesis, Department of Computer Science, Yale University, New Haven (1978)
Shamos, M.I., Hoey, D.: Closest point problem. In: Proceedings of the 16th IEEE Annual Symposium Foundation of Computer Science, S. 151–162. (1975)
Shenoi, S., Melton, A., Fan, L.T.: Functional dependencies and normal forms in the fuzzy relational database model. Inform. Sci. **60**, 1–28 (1992)
Silberschatz, A., Korth, H.F., Sudarshan, S.: Database Systems Concepts. McGraw-Hill (2010)
Silverston, L.: The Data Model Resource Book, Bd. 2. Wiley (2001)
Smith, J.M., Smith, D.C.P.: Database abstractions: aggregation and generalization. ACM Trans. Database Syst. **2**(2), 105–133 (1977)
Snodgrass, R.T.: The temporal query language TQuel. ACM Trans. Database Syst. **12**(2), 247–298 (1987)
Snodgrass, R.T., et al.: A TSQL2 tutorial. SIGMOD Rec. **23**(3), 27–33 (1994)
Stein, W.: Objektorientierte Analysemethoden – Vergleich, Bewertung, Auswahl. Bibliographisches Institut (1994)
Stonebraker, M.: The Ingres Papers. Addison-Wesley (1986)
Störl, U.: Backup und Recovery in Datenbanksystemen – Verfahren, Klassifikation, Implementierung und Bewertung. Teubner (2001)
Takahashi, Y.: A fuzzy query language for relational databases. In: Bosc, P., Kacprzyk, J. (Hrsg.) Fuzzyness in Database Management Systems, S. 365–384. Physica-Verlag (1995)
Tilkov, S.: REST und HTTP – Einsatz der Architektur des Web für Integrationsszenarien. dpunkt (2011)
Tittmann, P.: Graphentheorie – Eine anwendungsorientierte Einführung. Fachbuchverlag Leipzig (2011)
Tsichritzis, D.C., Lochovsky, F.H.: Data Models. Prentice-Hall (1982)
Turau, V.: Algorithmische Graphentheorie. Oldenbourg (2009)
Türker, C.: SQL:1999 & SQL:2003 – Objektrelationales SQL, SQLJ & SQL/XML. dpunkt (2003)
Ullman, J.: Principles of Database Systems. Computer Science Press (1982)
Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)
Van Steen, M.: Graph Theory and Complex Networks – An Introduction. Maarten van Steen (2010)
Vetter, M.: Aufbau betrieblicher Informationssysteme mittels pseudo-objektorientierter, konzeptioneller Datenmodellierung. Teubner (1998)
Vogels, W.: Eventually consistent. Commun. ACM **52**(1), 40–44 (2009)
Vossen, G.: Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme. Oldenbourg (2008)
W3C: World Wide Web Consortium. Siehe http://www.w3.org/ (2014). Zugegriffen am 18.12.2014
Wedekind, H.: Datenbanksysteme – Eine konstruktive Einführung in die Datenverarbeitung in Wirtschaft und Verwaltung. Spektrum Akademischer Verlag (1981)
Weikum, G.: Transaktionen in Datenbanksystemen – Fehlertolerante Steuerung paralleler Abläufe. Addison Wesley (1988)

Weikum, G., Vossen, G.: Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann (2002)
Werro, N.: Fuzzy Classification of Online Customers. Springer (2015)
Wiederhold, G.: Database Design. McGraw-Hill (1983)
Williams, R., et al.: R*: An overview of the architecture. In: Scheuermann, P. (Hrsg.) Improving Database Usability and Responsiveness, S. 1–27. Academic (1982)
Witten, I.H., Frank, E.: Data Mining. Morgan Kaufmann (2005)
Wood, P.T.: Query languages for graph databases. SIGMOD Rec. **41**(1), 50–60 (2012). http://doi.org/10.1145/2206869.2206879. Zugegriffen am 24.08.2015
Zadeh, L.A.: Fuzzy sets. Inf. Control **8**, 338–353 (1965)
Zehnder, C.A.: Informationssysteme und Datenbanken. vdf Hochschulverlag (2005)
Zloof, M.M.: Query-by-example: a data base language. IBM Syst. J. **16**(4), 324–343 (1977)

Stichwortverzeichnis

A. Meier, M. Kaufmann (Hrsg.), *SQL- & NoSQL-Datenbanken*, eXamen.press,
DOI 10.1007/978-3-662-47664-2