Vorlesung "Informationssysteme"
Dr. Ralf Schenkel, Prof. Dr. Gerhard Weikum
Universität des Saarlandes, Sommersemester 2005

Inhalt und Lernziel:

Die Vorlesung vermittelt grundlegende Kenntnisse über Konzepte und Schnittstellen von Datenbanksystemen und anderen Arten von Informationsdienstsoftware (z.B. Search-Engines oder Data-Mining-Tools) sowie der Anwendungsentwicklungswerkzeuge zur Realisierung von Informationssystemen mit strukturierten, semistrukturierten und unstrukturierten Daten. Schwerpunktthemen sind Anfragesprachen für Datenbanksysteme, Daten- und Prozeßmodellierung sowie Relevanzmodelle für Information-Retrieval und Data-Mining. Die entsprechenden Grundlagen aus der Logik und Stochastik werden in der Vorlesung eingeführt.

Organisatorisches:

Vorlesungs- und Übungsunterlagen

Unterlagen zur Vorlesung und zu den Übungen werden unter der folgenden URL zur Verfügung gestellt: http://www.mpi-sb.mpg.de/units/ag5/teaching/ss05/is05/ .

Termine und Betreuung

Vorlesung: Dienstag 9-11 und Donnerstag 9-11 Uhr in 45/002
Übungen: siehe Webseite
Die Vorlesung beginnt am 12. April; die erste Übungsstunde ist in der zweiten Vorlesungswoche.

Die Übungskoordinatoren sind
Dipl.-Math. Sergej Sizov (sizov@mpi-sb.mpg.de) für die praktischen Übungen und
Dipl.-Inform. Christian Zimmer (czimmer@mpi-sb.mpg.de) für die Papierübungen.
Sprechstunde von Dr. Schenkel ist Donnerstag 14-15 Uhr in 46/404 oder n.V. (schenkel@mpi-sb.mpg.de)
Sprechstunde von Prof. Weikum ist Dienstag 14-15 Uhr in 46/401 oder n.V. (weikum@mpi-sb.mpg.de).

Leistungsprüfung

Es werden 9 benotete Leistungspunkte vergeben, wenn folgende Voraussetzungen erfüllt sind:

1. a) erfolgreiche Teilnahme an zwei Teilklausuren in der Mitte und am Ende des Semesters (voraussichtlich am Samstag, dem 11.6., und am Donnerstag, dem 21.7.)
   oder
   b) erfolgreiche Teilnahme an einer Teilklausur und der Wiederholungsklausur Mitte Oktober,
2. mindestens zweimaliges Präsentieren von Lösungen in der Übungsgruppe,
3. erfolgreiche Bearbeitung der praktischen Übungen (Teamarbeit in Dreiergruppen möglich).

Die Note wird aus den Ergebnissen der zwei bestandenen (Teil-) Klausuren berechnet.
Durch zusätzliche Präsentationen in der Übungsgruppe sowie durch besonders gute Lösungen der praktischen Übungen können Bonuspunkte erworben werden, die die Gesamtnote verbessern.

Geplante Gliederung:

1. Einführung und Überblick: Anwendungen, Systeme, Prinzipien

Teil I: Suchmaschinen

2. Vektorraummodell für Suchmaschinen
3. Automatische Klassifikation von Dokumenten
4. Linkanalyse für Autoritäts-Ranking

Teil II: Datenbankschnittstellen

5. Relationenmodell und algebraorientierte Anfragesprachen
6. Logikorientierte Anfragesprachen
7. Datenbanksprache SQL
8. Anwendungsentwicklung mit SQL und JDBC
9. Integritätssicherung mit SQL
10. Objektorientierte und objekt-relationale Datenmodelle

Teil III: Datenbank- und Anwendungsentwurf

11. Relationale Entwurfstheorie
12. Datenbankentwurf mit UML
13. Prozessmodellierung mit Statecharts

Teil IV: Implementierungskonzepte von Datenbanksystemen

14. Datenspeicherung, Indexstrukturen und Anfrageauswertung
15. Transaktionsverwaltung: Concurrency-Control
16. Transaktionsverwaltung: Recovery

Teil V: Informationsdienste

17. OLAP und Data-Warehouses
18. Daten-Mining: Klassifikation, Assoziationsregeln
19. Semistrukturierte Daten und XML-Anfragesprachen

Literatur:

Zu Teil I: Suchmaschinen

Ricardo Baeza-Yates, Berthier Ribeiro-Neto: Modern Information Retrieval, Addison-Wesley, 1999

Christopher D. Manning, Hinrich Schütze: Foundations of Statistical Natural Language Processing, MIT Press, 1999

Soumen Chakrabarti: Mining the Web, Morgan Kaufmann, 2002

Pierre Baldi, Paolo Frasconi, Padhraic Smyth: Modeling the Internet and the Web, Wiley&Sons, 2003

Ian Witten, Alistair Moffat, Timothy C. Bell: Managing Gigabytes, Academic Press, 1999

Zu Teil II: Datenbankschnittstellen

Jeffrey D. Ullman, Jennifer Widom: First Course in Database Systems, Prentice Hall, 1997

Patrick O'Neil, Elizabeth O'Neil: Database Principles, Programming, and Performance, Morgan Kaufmann, 2001

Joachim Biskup: Grundlagen von Informationssystemen, Vieweg-Verlag, 1995

Andreas Heuer, Gunter Saake: Datenbanken - Konzepte und Sprachen, International Thomson Publishing, 2000

Alfons Kemper, Andre Eickler: Datenbanksysteme - eine Einführung, 5. Auflage, Oldenbourg, 2004

Gottfried Vossen: Datenbankmodelle, Datenbanksprachen und Datenbankmanagement-Systeme, Oldenbourg, 1999

Can Türker: SQL:1999 & SQL:2003, dpunkt-Verlag, 2003

Zu Teil III: Datenbank- und Anwendungsentwurf

Carlo Batini, Stefano Ceri, Shamkant Navathe: Conceptual Database Design - An Entity-Relationship Approach, Addison-Wesley, 1991

Toby J. Teorey: Database Modeling & Design, Morgan Kaufmann, 1998

Robert J. Muller: Database Design for Smarties - Using UML for Data Modeling, Morgan Kaufmann, 1999

Martin Fowler, Kendall Scott, Grady Booch: UML Distilled, Addison-Wesley, 1999

David Harel, Michal Politi: Modeling Reactive Systems - the Statemate Approach, McGraw Hill, 1998

Edmund M. Clarke, Orna Grumberg, Doron A. Peled: Model Checking, MIT Press, 2000

Zu Teil IV: Implementierungskonzepte von Datenbanksystemen

Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom: Database System Implementation, Prentice Hall, 1999

Theo Härder, Erhard Rahm: Datenbanksysteme - Konzepte und Techniken der Implementierung, Springer, 2001

Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems, McGraw Hill, 2000

Gerhard Weikum, Gottfried Vossen: Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001

Zu Teil V: Informationsdienste

Jiawei Han, Micheline Kamber: Data Mining - Concepts and Techniques, Morgan Kaufmann, 2001

Martin Ester, Jörg Sander: Knowledge Discovery in Databases – Techniken und Anwendungen, Springer-Verlag 2000

Harald Schöning: XML und Datenbanken, Hanser-Verlag, 2003

Henk Blanken, Torsten Grabs, Hans-Jörg Schek, Ralf Schenkel, Gerhard Weikum (Editors): Intelligent Search on XML Data, Springer-Verlag, 2003

Kapitel 1: Einführung und Überblick -
Anwendungen, Systeme, Prinzipien

1.1 Anwendungen

Die Vorlesung vermittelt grundlegende Kenntnisse über die Informatikmethoden zum Entwurf, der Implementierung und dem Betrieb moderner Informationssysteme, insbesondere Konzepte und Schnittstellen von Datenbanksystemen und anderen Arten von Informationsdienstsoftware (z.B. Search-Engines oder Data-Mining-Tools) sowie Einblicke in Anwendungsentwicklungsmethoden. Die folgenden Beispiele nennen einige typische Informationssysteme, die von außerordentlich hohem wirtschaftlichem und gesellschaftlichem Nutzen sind:

- *Flugreservationen und -buchungen einer Fluggesellschaft*. Derartige Systeme verwalten typischerweise einige Millionen Reservationen und Buchungen (bis zu einem Jahr im voraus) und müssen in Spitzenzeiten bis zu 100 Such- und Änderungsoperationen pro Sekunde ausführen können. Darüber hinaus müssen solche Systeme praktisch rund um die Uhr an 365 Tagen im Jahr betriebsbereit sein. Eine Ausfallzeit von mehr als wenigen Minuten im Jahr gilt bereits als unakzeptabel.
- *Verwaltung eines Teilelagers eines Automobilherstellers*. Dies umfaßt u.U. mehrere Millionen Teile für alle Modelle, für die der Hersteller die Lieferung von Ersatzteilen garantiert. Das System dient u.a. zur Buchführung über die Lagerbestände und zum Auffinden der Teile in roboterbedienten Lagerhallen. Zusätzlich gibt das System Auskunft darüber, welche Teile für welche Modelle verwendet werden können, welche Teile untereinander kompatibel sind, usw.
- *Entscheidungsunterstützung für den Wertschriftenhandel einer Bank*. Börsenkurse und damit zusammenhängende Informationen (z.B. Wirtschaftsnachrichten) werden über entsprechende Datenleitungen von verschiedenen Börsen kontinuierlich erfaßt, und es werden komplexe Suchoperationen auf den Daten spezifiziert, die die Händler auf interessante Konstellationen aufmerksam machen sollen. Dies ist ein Beispiel eines Echtzeit-Informationssystems mit harten Anforderungen an die Antwortzeit der Suchoperationen und extrem hohen Änderungsraten der Daten (z.T. mehr als 100 Kurse pro Sekunde).
- *Verwaltung von Patientendaten in einem Krankenhaus*. Dies beinhaltet Informationen über die Krankengeschichte eines Patienten, Untersuchungsergebnisse, Einsatz von Medikamenten, Behandlungen, bis hin zur Kostenabrechnung. Über die für administrative Zwecke notwendigen Such- und Änderungsoperationen hinaus ist u.a. die Unterstützung der Suche nach ähnlichen Krankheitsbildern wünschenswert. Zusätzlich wird die Anbindung von Archivsystemen für Ultraschall- und Röntgenbilder, Computertomographien u.ä. angestrebt.
- *Informationssuche für Zeitungsredakteure*: In der Redaktion einer Zeitung; Zeitschrift oder Nachrichtenagentur werden riesige Mengen an Nachrichten inkl. Photos, Videomaterial, etc. archiviert. Auf diesen Daten suchen Redakteure und Journalisten nach relevantem Hintergrund- oder Vergleichsmaterial zu einer aktuellen Meldung oder Story. Die Suche soll zum einem Ergebnisse liefern, die zur Anfrage textuell ähnlich sind, aber darüber hinaus auch inhaltliche Ähnlichkeit berücksichtigen, die sich auf linguistische Analysen und Begriffsverwandtschaften aus sog. Thesauri oder Ontologien abstützt. Beispielsweise sollte eine Anfrage wie „Sommer 2000 im Saarland“ auch einen Artikel finden, in dem „1. August 2000:

Marienwunder in Marpingen“ steht. Die Daten selbst können dabei reine Textdokumente sein, zunehmend werden aber auch sog. semistrukturierte Datenformate wie XML verwendet, wobei z.B. Zeit- und Ortsangaben mit entsprechenden „Tags“ explizit markiert sind.

- *Verwaltung von Kundenanfragen in einem Customer-Support-System*: Bei Unternehmen mit vielen verschiedenen Kunden und wartungsintensiven Produkten bzw. Dienstleistungen, wie etwas einer großen Softwarefirma, einem Internet-Händler für Unterhaltungselektronik oder einem Internet-Service-Provider, müssen Problemberichte („Bug Reports“) und Anfragen von Kunden verwaltet werden. Eine typische Anfrage, die per Email, Web-Interface oder Call-Center eintrifft, könnte folgendermaßen lauten: “My notebook, which is model … configured with …, has a problem with the driver of its WaveLAN card. I already tried the following fixes … but only received error messages …” Aus einem solchen Text versucht man, möglichst viele strukturierte Attribute wie z.B. NotebookModel zu extrahieren. Auf der Basis dieses strukturierten Teils, aber auch des verbleibenden reinen Textteils versucht man, die Anfrage automatisch zu klassifizieren, z.B. einem Thema wie /notebooks/drivers/network innerhalb einer Taxonomie von Kategorien zuzuordnen. Dies ist entweder die Grundlage für die Zuteilung des Problems an geeignet qualifizierte Techniker oder hilft bei der Suche nach inhaltlich ähnlichen, früheren Problemberichten, für die bereits eine Lösung bekannt ist. Idealerweise könnte man aus einer großen Sammlung völlig unterschiedlich formulierter Kundenanfragen eine Sammlung von FAQs (Frequently Asked Questions) aufbauen und neue Anfragen automatisch darauf abbilden. Schwierige oder völlig neuartige Anfragen, für die es keine sinnvolle Abbildung gibt, erfordern eine tiefergehende Problemanalyse sowie ggf. einen Dialog mit dem Kunden und werden als Workflow im Customer-Support-System verwaltet.

Viele Anwendungen haben heute Web-basierte Benutzerschnittstellen und sind auch in ihrer Systemarchitektur stark mit Web-Technologien verknüpft. Ein weiteres zentrales Thema ist die Interoperabilität zwischen verschiedenen, weitgehend unabhängigen Datenbanken und sonstigen Informationsquellen untereinander sowie mit den verschiedenen Anwendungssystemen vor dem Hintergrund einer verteilten, hochgradig heterogenen Informationslandschaft. Ziel ist es, den Anwendungen einen systemübergreifenden Datenzugriff auf möglichst bequeme Art zu ermöglichen. Im folgenden sind drei Beispielszenarios aufgeführt, für die diese Forderung essentiell ist:

- Integrierte Auswertung verschiedenartiger Informationsquellen zur intelligenten Steuerung des Straßenverkehrs. Die Grundlage solcher Leitsysteme bilden einerseits geographische Daten und Daten über das Straßennetz sowie andererseits ständig aktualisierte Daten über die Verkehrsdichte, die Straßenbeschaffenheit, die Wetterlage und -vorhersage, Auslastung von Parkhäusern usw.
- Integrierte und weitestgehend automatisierte Abwicklung von Vorgängen, sogenannten „Workflows“, die eine Vielzahl von Informationssystemen verschiedener Institutionen betreffen. Beispielsweise beinhaltet der Erwerb eines Einfamilienhauses Prüfungen und Eintragungen beim Grundbuchamt, der finanzierenden Bank, dem Notar usw. Gleichzeitig sollten auch alle mit dem Umzug verbundenen Formalitäten wie beispielsweise die Ab- und Anmeldung von Gas/Wasser/Strom, Telefon, Internetanschluß usw. elektronisch abgewickelt werden.
- Prüfung von Firmenkrediten bei einer Bank mit entsprechenden Bonitätsprüfungen und Risikoabschätzungen bezüglich des Kreditnehmers sowie seiner finanziellen Verflechtungen mit anderen nationalen und internationalen Unternehmen. Dies beinhaltet die Unterstützung ver-

teilter Arbeitsabläufe, die sich über verschiedene Geschäftssparten und verschiedene internationale Niederlassungen einer Bank erstrecken und Zugriff auf ein breites Spektrum von Informationssystemen erfordern - von Wirtschaftsnachrichten und Börsenkursen bis zu Informationen über die Aufsichtsratsmitglieder des potentiellen Kreditnehmers.

Ein moderner Trend ist der Einsatz generischer (d.h. für breite Einsatzfelder konzipierte) Anwendungsklassen wie etwa

- Enterprise Resource Planning (z.B. SAP R/3) für die umfassende – operative und strategische – Führung von Unternehmen,
- E-Commerce und E-Business in den Spielarten B2C (Business-to-Consumer, z.B. Handel, Auktionen, Maklerdienste) und B2B (Business-to-Business, z.B. Logistikketten, Service-Outsourcing) oder
- digitale Bibliotheken und multimediale Archive (z.B. Patentsammlungen, Nachrichtenarchive, virtuelle Museen).

Hierfür gibt es hochgradig „parametrisierte“ und flexibel anpaßbare Softwarelösungen, die für den individuellen Einsatzfall konfigurierbar und – im Sinne eines „Customizing“ – spezialisierbar sind sowie durch weitere Software ergänzt werden können.

Einige interessante Web-basierte Anwendungen

http://www.imdb.com - Internet Movie Database für Filmfans

http://www.amazon.com - E-Commerce-System zum Verkauf von Büchern, CDs, etc.,
Textinformationssystem mit Buchrezensionen und Data-Warehouse mit Kundenprofilen

http://www.ebay.com - Auktionssystem für Artikel aller Arten

http://www.terraserver.com bzw. http://www.terraserver.microsoft.com –
Geographisches Informationssystem mit Landkarten und Satellitenbildern

http://skyserver.sdss.org/dr1/en/ - Himmelsatlas mit astronomischen Karten und Daten

http://www.google.com und http://vivisimo.com/ - Suchmaschinen für das Web

http://www-db.stanford.edu/IMAGE und http://129.132.14.36/Chariot - Ähnlichkeitssuche auf Photos

http://www.hermitagemuseum.org - Virtuelles Museum mit Ähnlichkeitssuche auf Bildern

http://www.musicline.de/de/melodiesuche/input und http://www.name-this-tune.com/index.php?id=9&L=2
Ähnlichkeitssuche auf Musikstücken („Query by Humming“, „Query by Whistling“)

http://chembank.med.harvard.edu
Chemische Datensammlung mit Ähnlichkeitssuche auf Molekülen

http://www.eti.uva.nl/Database/WBD.html
Datenbank mit organischen Spezies

http://au.expasy.org/srs5/
Proteindatenbanken

http://www.wikipedia.org
Online-Enzyklopädie, die von Internet-Benutzern der ganzen Welt freiwillig gepflegt wird

Technische Anforderungen an Informationssysteme

- Textsuche mit nach Relevanz geordneten Trefferranglisten
- Automatische Organisation von Dokumenten
- Verwaltung von (Deep-)Web-Daten, strukturierten Datenbanken und semistrukturierten (XML-)Daten
- Komfortable GUIs (Graphical User Interfaces)
- deklarative Anfragesprachen
- Zuverlässige Verwaltung sehr großer, persistenter Datenmengen (> 5 Terabytes)
- Hohe Verfügbarkeit (7 x 24 Stunden in der Woche) und langfristige Archivierung
- Gute - vorhersagbare und garantierbare - Leistung
    - Hoher Durchsatz (Anzahl der bedienbaren Aufträge pro Zeiteinheit, Anzahl der gleichzeitig aktiven Clients)
    - Kurze, benutzerakzeptable Antwortzeiten im Mehrbenutzerbetrieb („Quality of Service“)
- Konsistenz verteilter Daten
- Integrierter Zugriff auf heterogene Daten
- Daten-Mining nach Korrelationen, Regeln, Klassifikationen, etc.
- Aktive Regeln und Prozesskoordination
- Komplexe Datentypen (Landkarten, Satellitenbilder, Himmelskarten, Moleküle, usw.)
- Ähnlichkeitssuche auf Bildern, wissenschaftlichen Daten, etc.
- Integration mit Web-Applikation und Web-Services

1.2 Systemarchitekturen

Drei-Ebenen-Architektur

Die typische Systemarchitektur moderner Informationssysteme ist in Abbildung 1.1 dargestellt. Sie wird in der Regel als Drei-Ebenen-Architektur (engl. „three-tier architecture" oder noch allgemeiner "multi-tier architecture") bezeichnet und enthält eine Zwei-Ebenen-Architektur als Spezialfall. Die Architektur umfaßt *Clients* (PCs, Workstations, Notebooks, Palmtops, Terminals, digitale TV Set-Top-Boxes, Handies und etliche weitere Arten von "Gizmos" und intelligenten Sensoren und Aktoren). Die Clients kommunizieren mit einem *Applikations-Server* (oder mehreren solcher Server), die wiederum mit einem *Daten-Server* (oder mehreren solcher Server) kommunizieren.

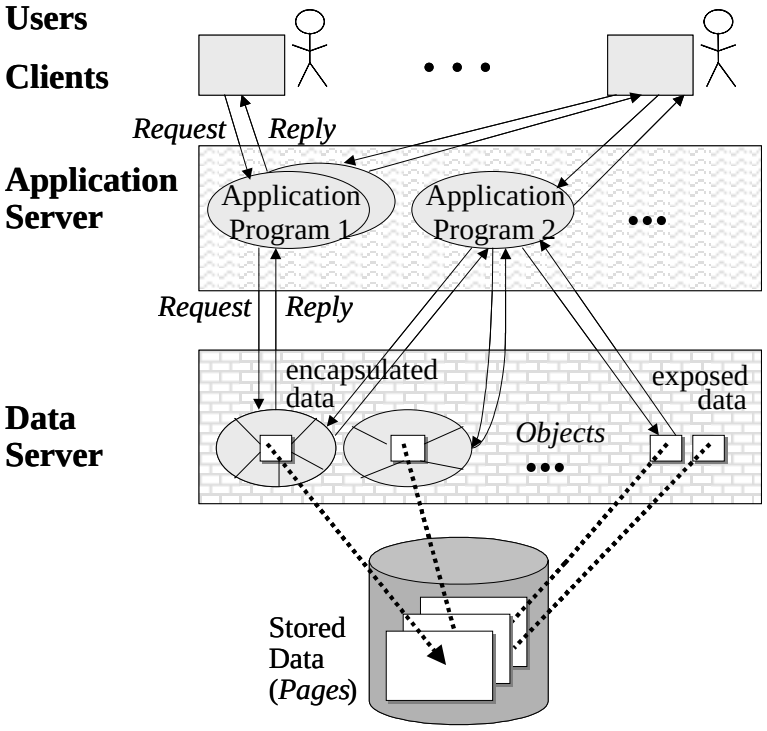


Abbildung 1.1: Drei-Ebenen-Architektur eines Informationssystems

Clients schicken anwendungsorientierte Aufträge (zur Erfüllung eines Geschäftsziels) an den Applikations-Server. Dies kann z.B. der Auftrag zu einer Bankkontogutschrift oder –lastschrift im Home-Banking sein, das Starten einer E-Commerce-Sitzung zum Füllen eines elektronischen Warenkorbs oder das Initiieren eines Workflows zur Planung und Buchung einer individualisierten Reise. In modernen Anwendungen werden diese Aufträge typischerweise mittels eines GUI (Graphical User Interface) generiert; analog wird die Antwort des Applikations-Servers häufig ebenfalls in graphischer Form präsentiert mittels Formularen, Diagrammen oder sogar Animationen in Virtual-Reality-Szenarien. Die Verarbeitung der Präsentation findet sowohl bei der Eingabe als auch bei der Ausgabe (primär) im Client statt. HTML (Hypertext Markup Language), das meistverbreitete Datenformat

des World-Wide-Webs, ist besonders attraktiv als Präsentationsgrundlage, weil zur Verarbeitung auf Client-Seite in der Regel ein Web-Browser genügt, so daß eine riesige Anzahl potentieller Clients ohne besonderen Installationsaufwand mit dem Informationssystem arbeiten kann.

Der Applikations-Server verwaltet einen Pool von Anwendungsprogrammen in ausführbarer (also compilierter oder interpretierbarer) Form und ruft das für einen Client-Auftrag zuständige Programm in seinem Adressraum auf. Programme können in einer konventionellen Programmiersprache wie z.B. C oder Cobol geschrieben sein oder in einer Skriptsprache wie z.B. PHP, mit der sehr einfach dynamische HTML-Seiten erzeugt werden können.

Die Anzahl der in einem Applikations-Server verwalteten Programme und zu unterstützenden Typen von Client-Aufträgen kann sehr groß sein, so daß es sich anbietet, mittels des objektorientierten Paradigmas Struktur in den Gesamtpool der Programme zu bringen. Dies führt auf sog. *Komponenten* oder *Business-Objekte* als Verwaltungseinheiten im Applikations-Server. Solche Objekte sind z.B. Konten, Kunden, Warenkataloge u.ä., die nach dem Prinzip Abstrakter Datentypen (ADTs) gekapselt sind, so daß ihre Implementierung verborgen bleibt und die Objekte nur über eine möglichst schmale Schnittstelle spezifischer Methoden (z.B. Gutschrift, Lastschrift, Kontoauszug, etc.) manipuliert werden können. Für ihre Implementierung dürfen Business-Objekte selbst wieder Methoden anderer Objekte aufrufen, so daß Komponenten-Code möglichst wiederverwendet wird. Die Kapselung wiederum vermeidet Abhängigkeiten von den Interna anderer Objekte, so daß der langfristige Wartungsaufwand für die Software geringer wird. Die Implementierung der Methoden eines Business-Objekts beinhaltet typischerweise einen oder mehrere Aufrufe an einen oder mehrere Daten-Server. Ein Beispiel eines Informationssystems, dessen Applikations-Server-Programme in Business-Objekten organisiert sind, ist das ERP-System (Enterprise Resource Planning) SAP R/3, eine umfassende Softwaresuite für die betriebswirtschaftlichen Belange eines Unternehmens.

Der Applikations-Server bildet die Laufzeitumgebung der Anwendungsprogramme: er erzeugt entsprechende Threads oder Prozesse, überwacht die Programmausführungen und behandelt bestimmte Fehler- und Ausnahmesituationen. Clients kommunizieren mit dem Applikations-Server im Web-Zeitalter in der Regel mittels HTTP (Hypertext Transport Protocol), das wiederum auf TCP/IP-Verbindungen aufsetzt. Da HTTP ein zustandsloses Protokoll ist (der Empfänger eines Auftrags vergisst den Auftrag nach der Ausführung und dem Absenden der Antwort), Clients aber häufig ganze Interaktionsfolgen als eine Einheit betrachten, ist der Applikations-Server auch für die Realisierung von Sitzungen (engl. sessions) zuständig: er muß sich entsprechende Zustandsinformationen über die Sitzungen aktiver Clients merken, Sitzungen überwachen und – nach Timeouts - ggf. zwangsweise terminieren. Insgesamt kann man die Rolle des Applikations-Servers als die eines Auftragsmaklers (engl. request broker) bezeichnen.

Solche Auftragsmakler und Laufzeitsysteme für Anwendungsprogramme wurden früher als *TP-Monitore* (engl. transaction processing monitors) bezeichnet (z.B. CICS, Tuxedo, etc.), weil sie primär für sog. OLTP-Anwendungen (online transaction processing) wie Reservations- und Buchungssysteme konzipiert waren. Eine modernere Variante davon sind die sog. "*Object Request Broker*" (ORBs), die die Infrastruktur für allgemeine verteilte Anwendungen bereitstellen und auf Komponentenmodellen wie CORBA (Common Object Request Broker Architecture), COM+ (Component Object Model) oder EJB (Enterprise Java Beans) basieren (z.B. VisiBroker, Orbix, etc.). Eine letzte Kategorie von Applikations-Servern sind *Web-Server* (wie z.B. Apache oder Internet Information Server), also im wesentlichen eine Kombination aus HTTP-Server und Servlet-Engine, wobei Servlets die Web-Variante von Business-Objekt-Methoden sind, die unter der Kontrolle des Applikations-Servers laufen. Wenn man schließlich den Kontrollfluß der Objektmethodenaufrufe auf langlebige (Geschäfts-) Prozesse wie z.B. den gesamten Zyklus eines Warenkaufs von der Bestellung bis

zur Mahnung ausdehnt und die Koordination dieser Zusammenhänge im Applikations-Server realisiert, spricht man auch von einer *Workflow-Engine* oder einem *Workflow-Management-System* (z.B. MQ Series Workflow, Staffware, etc.). Alle diese Varianten faßt man auch unter dem Begriff *Middleware* zusammen. Die Unterschiede zwischen den verschiedenen Kategorien von Infrastruktursoftware sind konzeptionell eher nebensächlich und nur hinsichtlich der Terminologie und Features von Produkten signifikant.

Wie erwähnt beinhaltet die Implementierung von Business-Objekten in der Regel Aufrufe eines Daten-Servers oder mehrerer solcher Server (die man im Kontext einer Drei-Ebenen-Architektur auch als "Backends" bezeichnet). Typischerweise werden alle persistenten, d.h. die Dauer einer Sitzung überlebenden, Zustandsinformationen im Daten-Server verwaltet. Datenbank-Server (z.B. Oracle, IBM DB2 oder MS SQL Server) sind mit Abstand die wichtigste Daten-Server-Kategorie, aber auch spezialisierte Textdokument- oder Multimedia-Systeme (z.B. Google oder Verity) oder Mail-Server (z.B. MS Exchange oder Lotus Domino) können von Business-Objekten aufgerufen werden. Zur Kommunikation mit einem Datenbank-Server wird die standardisierte Sprache SQL (Structured Query Language) verwendet, wobei SQL-Kommandos und -Resultate oft in ein Datenverbindungsprotokoll wie ODBC (Open Database Connectivity) oder JDBC (Java Database Connectivity) eingebettet sind. Viele Datenbanksysteme unterstützen auch sog. Stored Procedures, mit denen sich Abstrakte Datentypen bzw. Business-Objekte im Datenbank-Server selbst realisieren lassen; damit hat man Flexibilität, inwieweit man die Kapselung von Business-Objekten im Applikations-Server und/oder im Daten-Server realisiert.

Zwei-Ebenen-Architektur

Durch Weglassen des Applikations-Servers, also der mittleren Ebene einer Drei-Ebenen-Architektur erhält man eine Spezialisierung, die als *Client-Server-Architektur* bezeichnet wird und historisch vor der mit der Web-Revolution einhergehenden Verbreitung der Drei-Ebenen-Architektur die häufigste Informationssystemarchitektur war. Wenn man dabei die Business-Objekte der Anwendung im Client realisiert, hat man sozusagen "fette Clients" (engl. "fat clients"); wenn man sie im Datenbank-Server mittels Stored Procedures o.ä. realisiert, hat man "dünne Clients" (engl. "thin clients"), unter die der historisch noch ältere Ansatz fällt, als Clients einfache Terminals zu verwenden. Der Hauptgrund, der zu der heute dominierenden Drei-Ebenen-Architektur geführt hat, liegt in der Anforderung der Skalierbarkeit, also der Möglichkeit, das System auf einfache Weise für eine größer werdende Anzahl von Clients aufzurüsten. Bei einer Zwei-Ebenen-Architektur kann insbesondere die große Anzahl von Client-Server-Sitzungen leicht zu Engpässen im Datenbank-Server führen. Bei einer Drei-Ebenen-Architektur dagegen erreicht man die Skalierbarkeit einfach durch Replikation des Applikations-Servers, wobei im Internet-Einsatz alle Applikations-Server durch entsprechende Router unter derselben IP-Nummer bzw. URL erreichbar sind.

Föderative Systemstrukturen

Anspruchsvolle Informationssysteme können Daten aus verschiedenen Informationsquellen integrieren, so daß ihre Business-Objekte mit mehr als einem Daten-Server kommunizieren. Außerdem ist es möglich, daß der oder die Applikations-Server ihrerseits die Dienste eines anderen Applikations-Servers oder mehrerer Applikations- und Daten-Server in Anspruch nehmen. Alle diese Möglichkeiten können frei kombiniert werden. Dabei sind die beteiligten Server häufig heterogen und autonom in dem Sinne, daß sie nicht exklusiv für eine Anwendung arbeiten, sondern vielmehr sowohl "lokale" Aufträge bedienen als auch ihre Dienste in einem Systemverbund anbieten. Solche

verteilten Systemstrukturen mit lose gekoppelten Server-Systmen nennt man daher auch föderative Informationssysteme. Abbildung 1.2 illustriert diesen Fall. Ein Praxisbeispiel eines solchen Dienstes ist das "virtuelle" Reisebüro Expedia, das intern sowohl "lokale" Daten-Server hat als auch mit autonom betriebenen anderen Diensten wie Amadeus, Sabre und weiteren Großhändlern der Reisebranche kommuniziert

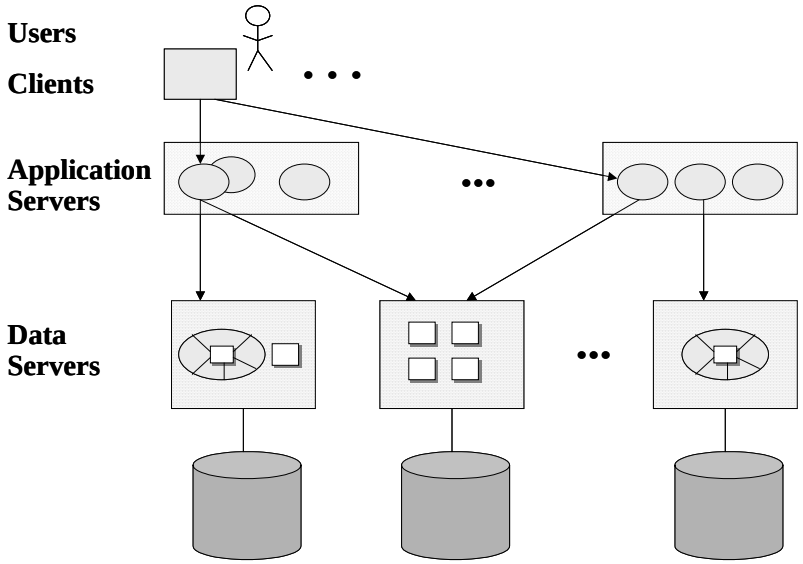


Abbildung 1.2: Föderative Systemarchitektur

Föderative Informationssysteme benötigen offensichtlich eine besonders zuverlässige und zugleich leicht wartbare - "offene" - Kommunikationsinfrastruktur, die über die einfachen Protokolle des Internet wie TCP/IP oder HTTP hinausgeht. Dafür sind die bereits erwähnten Varianten komponentenorienter Middleware geeignet, wie z.B. CORBA, EJB, .Net und sogenannten Web Services. Die Middleware verbirgt weitgehend die Heterogenität der Systemlandschaft und dient als Auftragsmakler für verteilte Programmaufrufe (engl. remote method invocation).

1.3 Grundprinzipien von Datenbanksystemen

Datenbanksysteme bilden das Softwarerückgrat der meisten computergestützten Informationssysteme. Sie stellen dem Anwendungsentwickler mächtige, generische Operationen zum Suchen in sehr großen Datenbeständen sowie zum Einfügen, Ändern und Löschen von Daten zur Verfügung. Operationen werden in einer deklarativen Datenmanipulationssprache spezifiziert, meist in der standardisierten Sprache SQL (Structured Query Language), und die Operationsaufrufe werden in Anwendungsprogramme eingebettet, die in einer konventionellen Programmiersprache (z.B. C, C++, Java, Cobol) oder einer Skriptsprache (z.B. Perl, PHP, Visual Basic, PL/SQL) geschrieben sind.

Aus den aufgeführten Anwendungen geht klar hervor, daß Datenbanksysteme u.U. riesige Datenmengen zu verwalten haben. Typische Datenbankgrößen in kommerziellen Anwendungen liegen im Bereich zwischen 10 und einigen 100 Gigabytes, sehr große Datenbanken umfassen mehr als 1 Terabyte und die allergrößten Datenbanken werden bald an die Petabyte-Grenze stoßen. Daraus folgt zwangsläufig, daß Datenbanken in erster Linie auf Sekundärspeichermedien liegen, in aller Regel auf Magnetplatten. Zwischen Hauptspeicher und Magnetplatten aber klafft ein riesiger Geschwindigkeitsunterschied von 5 bis 6 Zehnerpotenzen. Wenn wir uns beispielsweise die Geschwindigkeit verschiedener Speicherhierarchiestufen als eine räumliche Entfernung der Daten vom Prozessor vorstellen und annehmen, daß der CPU-Cache im selben Raum wie der Prozessor, also vielleicht 1 Meter entfernt, ist, dann befindet sich der Hauptspeicher im Nebenraum (ca. 10 Meter entfernt), die Magnetplatten aber stehen 1000 bis 10000 Kilometer entfernt in Moskau oder gar in Tokio. Aus diesem gewaltigen Geschwindigkeitsunterschied sowie aus der Nebenbedingung, daß Daten auf Magnetplatten nur blockweise adressiert werden können, ergibt sich die Notwendigkeit, daß Datenbanken anders als mit klassischen, hauptspeicherorientierten Datenstrukturen organisiert werden. Diese Überlegung führt auf das folgende Grundprinzip.

> *Grundprinzip 1:*
> Um große Datenmengen effizient verwalten zu können, sind Datenbanksysteme im Hinblick auf *Sekundärspeicherzugriffe* optimiert.

Insbesondere sind auch die klassischen, RAM-orientierten Modelle der Komplexitätstheorie für Datenbanksysteme nur von untergeordneter Bedeutung; die entscheidende Effizienzmetrik ist vielmehr die Anzahl der Sekundärspeicherzugriffe.

Da Datenbanksysteme generische Systeme sind, also gleichermaßen in Bankkonten wie auch in Patientendaten suchen und ändern können, sind universelle Optimierungen der Systemeffizienz nur eingeschränkt möglich. Vielmehr hängt der Nutzen einer Optimierungsmaßnahme häufig von den spezifischen Lastmerkmalen der jeweiligen Anwendung ab. In einer Anwendung mit geringer Änderungsrate kann man Daten - ggf. unter Einführung von Redundanz - so speichern, daß verschiedenartige Suchoperationen extrem effizient ausgeführt werden können; in einer Anwendung mit hoher Änderungsrate hingegen kann dies unakzeptabel sein, wenn die Aktualisierung der entsprechenden Datenstrukturen zu aufwendig ist. Aus diesem Grund unterstützt ein Datenbanksystem in der Regel ein breites Spektrum verschiedener Speicherungs- und Zugriffsstrukturen und überläßt es einem Tuning-Spezialisten, diese Strukturen für eine Anwendung festzulegen. Da sich aber die Lastmerkmale einer Anwendung im Laufe der Zeit ändern (z.B. durch Erweiterung der Anwendungsfunktionalität), müssen diese Tuning-Entscheidungen hin und wieder revidiert werden.

Solche Änderungen der Speicherungs- und Zugriffsstrukturen sind nicht gerade eine billige Maßnahme, da dazu häufig große Datenmengen vom Datenbanksystem umgespeichert werden müssen. Von fundamentaler Bedeutung ist jedoch, daß bei einer solchen Revision der Datenspeicherung die Anwendungsprogramme auf keinen Fall geändert werden müssen, denn der entsprechende Änderungsaufwand wäre kostenmäßig in den meisten Fällen absolut unakzeptabel. Datenbanksysteme stellen diese sogenannte *Programm-Daten-Unabhängigkeit* (häufig auch nur: Datenunabhängigkeit) sicher, indem sie die eigentlichen Speicherungs- und Zugriffsstrukturen vor dem Anwendungsprogramm verbergen. Mit anderen Worten: Datenbanksysteme wenden das Prinzip der Abstrakten Datentypen auf generische Weise an, indem sie die verwalteten Daten kapseln und nur deskriptive Operationen anbieten, die unabhängig von den zugrundeliegenden konkreten Datenstrukturen sind. In eingeschränkter Form gilt dieses Prinzip sogar auch bei einer anwendungsbedingten Änderung des Datenformats (z.B. der Umstellung der Postleitzahlen in Deutschland) sowie bei der nachträglichen Erweiterung existierender Daten um zusätzliche Informationen (z.B. der Hinzunahme einer Kreditkartennummer für alle Privatpatienten einer Klinik).

> *Grundprinzip 2:*
> Die Speicherung von Daten kann geändert werden, ohne daß Anwendungsprogramme geändert werden müssen (*Programm-Daten-Unabhängigkeit*).

Dieses fundamentale Prinzip hat ganz entscheidend zum kommerziellen Erfolg von Datenbanksystemen beigetragen. Etliche Informationssysteme, die ohne echtes Datenbanksystem entwickelt wurden (also beispielsweise auf der Basis eines File-Systems), sind mittelfristig gescheitert, weil sie aufgrund fehlender Programm-Daten-Unabhängigkeit unakzeptable Programmwartungskosten verursachten.

Das Grundprinzip der Programm-Daten-Unabhängigkeit und die damit einhergehende Konzeption deskriptiver Datenmanipulationssprachen wurde vor allem durch die Entwicklung *relationaler Datenbanksysteme* in den Siebziger und Achtziger Jahren entscheidend vorangebracht. Bei diesen Systemen, die heute den Markt dominieren, werden alle Daten konzeptionell in Form von Tabellen (mathematisch: Relationen), repräsentiert. Einen Schritt weiter geht man seit einigen Jahren mit *objektorientierten Datenbanksystemen (*bzw. entsprechenden „postrelationalen“ Weiterentwicklungen relationaler Systeme). Diese Systemklasse erlaubt es insbesondere, Aggregationsbeziehungen zwischen Datenelementen in Form „komplexer Objekte“ direkt zu repräsentieren; beispielsweise können alle Teile eines Motors zu einem Objekt „Motor“ zusammengefaßt werden -- eine Form der Aggregation, die mit relationalen Systemen nur auf umständliche Art möglich ist. Generische Such- und Änderungsoperationen werden sowohl auf der Ebene der Gesamtobjekte als auch auf der Ebene der Teilobjekte angeboten. Darüber hinaus ist es in objektorientierten Datenbanksystemen auch möglich, anwendungsspezifische Operationen für eine Menge von Objekten zu definieren, und diese können in Ausdrücken der Datenmanipulationssprache verwendet werden. Das Datenbanksystem, welches an sich als generisches Softwarepaket konzipiert ist, wird damit anwendungsspezifisch erweiterbar. Beispielsweise kann man eine Funktion „Volumen“ für geometrische Objekte definieren, die auf Polygonen für die Begrenzungsflächen eines dreidimensionalen Körpers operiert. Wie in objektorientierten Programmiersprachen können diese Funktionen mittels „Vererbung“ auf flexible Art und Weise für speziellere Objekte wiederverwendet werden (z.B. für die speziellen Körper, die beim mechanischen CAD auftreten); dabei können bei Bedarf spezifische Details „überschrieben“, d.h. neu festgelegt werden (z.B. die Interpolationsvorschrift zwischen Stützpunkten eines Polygons).

Das Prinzip der Programm-Daten-Unabhängigkeit sollte auch für objektorientierte Datenbanksysteme unter Einschluß der anwendungsspezifischen Operationen gelten. Allerdings gibt es durchaus noch offene Fragen in diesem Zusammenhang, die Gegenstand aktueller Forschung sind, beispielsweise die Konzeption und effiziente Implementierung objektorientierter Datenmanipulationssprachen und deren Einbettung in objektorientierte Programmiersprachen.

Die Effektivität eines Informationssystems steht und fällt mit der Korrektheit der zugrundeliegenden Daten. Eine Form der Datenkorrektheit, die von Datenbanksystemen sehr weitreichend unterstützt wird, ist die *Konsistenz* der Daten (oder auch: *Integrität* der Daten). Diese fordert, daß bestimmte Zusammenhänge zwischen verschiedenen Datenelementen, die in der realen Welt gelten sollen, auch in der Datenbank gewährleistet werden. Beispielsweise soll das Konto eines Kunden bei einem Versandhaus nur dann belastet werden, wenn eine entsprechende Bestellung ausgeliefert wird. Datenbanksysteme bieten die Möglichkeit an, daß solche anwendungsspezifischen Konsistenzbedingungen spezifiziert werden, und die Systeme überwachen die Einhaltung dieser Bedingungen. Änderungen, die die Konsistenz verletzen würden, werden zurückgewiesen. Damit werden Systeme gewissermaßen „immun“ gegen fehlerhafte bzw. inkonsistente Dateneingaben. Diese Vorgehensweise ist zwar grundsätzlich auf Plausibilitätsprüfungen beschränkt, da auch fehlerhafte Eingaben formal konsistent sein können; in vielen Anwendungen aber können dadurch fast alle Eingabefehler erkannt und eliminiert werden.

Einige Klassen von Konsistenzbedingungen können leichter spezifiziert und vom System überwacht werden, wenn die Datenbank selbst daraufhin entworfen wurde. In diese Klasse fallen beispielsweise Bedingungen vom Typ „keine zwei Konten dürfen dieselbe Kontonummer haben“ oder „die bei einer Buchung angegebene Kontonummer muß in der Datenbank existieren“. Die Datenbankforschung hat verschiedene Methoden des sogenannten konzeptionellen (oder auch: logischen) Datenbankentwurfs hervorgebracht, die in diesem Sinne auf eine hohe Qualitätssicherung für die Daten abzielen. Diese Methoden, z.B. die relationale Entwurfstheorie und vor allem der Entwurf mit „Entity-Relationship-Modellen“, haben in der Praxis weite Verbreitung gefunden. Moderne Datenbanksysteme gehen noch einen Schritt weiter und erlauben es, anwendungsspezifische Konsistenzbedingungen mit den Sprachmitteln der Datenmanipulationssprache deskriptiv zu spezifizieren. Mit Hilfe sogenannter *aktiver Datenbankmechanismen (*z.B. „Trigger“ und weitergehende Konzepte) ist es ferner möglich, spezifische Reaktionen beim Erkennen einer potentiellen Konsistenzverletzung festzulegen; beispielsweise können anstatt einer Zurückweisung der Änderung bestimmte Folgeänderungen automatisch ausgelöst werden. Diese Delegation der Verantwortung für die Datenkonsistenz an das Datenbanksystem stellt eine fundamentale Vereinfachung für die Anwendungsentwicklung dar.

> *Grundprinzip 3:*
> Die *Konsistenz* der Daten wird vom Datenbanksystem gewährleistet.

Die Konsistenz der Daten ist nicht nur durch Eingabefehler gefährdet, sondern auch durch „Interferenzeffekte“ paralleler Änderungsoperationen im Mehrbenutzerbetrieb sowie durch Fehler in Anwendungsprogrammen und Ausfälle des Datenbanksystems selbst. Beispielsweise ist es möglich, daß ein Bankkonto „Geld verliert“, wenn etwa zwei Operationen zur Einzahlung von jeweils 100 DM zunächst beide den Kontostand lesen, jeweils 100 DM dazu addieren und dann beide denselben Wert in die Datenbank zurückschreiben; die zuerst schreibende Einzahlung wird von der späteren fälschlicherweise überschrieben und geht praktisch verloren. Vermeiden ließe sich dieser Fehler na-

türlich durch entsprechende Synchronisationsmaßnahmen (z.B. unter Verwendung von Semaphoren) von Seiten der Anwendungsprogramme; dies aber wäre eine erhebliche Komplikation der Anwendungsentwicklung für Mehrbenutzersysteme mit einer entsprechenden Einbuße bei der Entwicklungsproduktivität. Ähnliche Fehlereffekte wie der eben beschriebene ergeben sich potentiell auch beim unkontrollierten Abbruch eines Anwendungsprogramms oder einem Ausfall des Datenbanksystems. Beispielsweise würde eine überweisung, die nach dem Belasten des ersten Kontos, aber vor der Gutschrift auf das zweite Konto durch einen Fehler abgebrochen wird, die Datenbank inkonsistent hinterlassen und wie zuvor „Geld verlieren".

Datenbanksysteme stellen alle Mechanismen bereit, um Fehlereffekte dieser Art garantiert zu verhindern. Dazu wird vom Anwendungsentwickler verlangt, daß Folgen von Datenbankoperationen innerhalb eines Anwendungsprogramms zu konsistenzerhaltenden Einheiten, sogenannten *Transaktionen*, zusammengefaßt und explizit spezifiziert werden. Beispielsweise bilden bei einer überweisung die Änderungsoperationen beider Konten zusammen eine Transaktion. Das Datenbanksystem gewährleistet dann die „(virtuelle) Isolation" von Transaktionen im Mehrbenutzerbetrieb, indem es (z.B. für das erste Beispiel) entsprechende Synchronisationsmaßnahmen für parallele Transaktionen automatisch generiert, und die „Atomarität" von Transaktionen, indem die Änderungen unvollständig ausgeführter Transaktionen (wie im zweiten Beispiel) mit Hilfe eines entsprechenden Logbuchs automatisch rückgängig gemacht werden. Mit Hilfe des Logbuchs wird darüber hinaus sichergestellt, daß Änderungen abgeschlossener Transaktionen wirklich „dauerhaft" sind, also weder durch Ausfälle des Datenbanksystems noch durch sonstige Fehler (z.B. Plattenfehler) verloren gehen. Diese systemgarantierten Eigenschaften von Transaktionen werden auch als *ACID-Prinzip* bezeichnet.

> *Grundprinzip 4:*
> Datenbankänderungen werden innerhalb von *Transaktionen* durchgeführt, die atomar (*atomic*), konsistenzerhaltend (*consistency-preserving*), isoliert (*isolated*) und dauerhaft (*durable*) sind.

Die Unterstützung von Transaktionen ist ein Grundpfeiler aller modernen Datenbanksysteme und ein Schlüsselkonzept zur Sicherstellung der Datenkonsistenz. Die dazu notwendigen Synchronisations- und Fehlertoleranzmaßnahmen werden vom Datenbanksystem sowohl für zentralisierte als auch für verteilte Datenbanken realisiert. Wegen seiner Allgemeinheit und Leistungsfähigkeit finden das Transaktionskonzept und die entsprechenden Implementierungstechniken der Transaktionsverwaltung auch in Betriebssystemen sowie in persistenten Programmiersprachen zunehmende Beachtung. Beispielsweise sehen Industriestandards wie X/Open, OMG OTS (Object Transaction Service der Object Management Group), oder EJB (Enterprise Java Beans) Transaktionen als universelles Konzept vor, um atomare Prozesse mit Aufrufen beliebiger verteilter Ressourcenmanager zu realisieren.

Der Stand der Technik auf dem Datenbankgebiet läßt sich folgendermaßen zusammenfassen. Datenbanksysteme sind ausgereifte Systeme, ohne die zahlreiche computergestützte Informationssysteme undenkbar wären. Der Markt wird heute von relationalen Systemen dominiert; objektorientierte Systeme haben einen kleinen Marktanteil, aber viele Anbieter relationaler Systeme haben objektorientierte Konzepte in ihre Produkte integriert und nennen diese dann „objekt-relational". Aktuell sind viele Hersteller im Begriff, Unterstützung für den vom W3C (World Wide Web Consortium) zum universellen Datenaustauschformat erklärten XML-Standard (Extensible Markup Language) in ihre Produkte einzubauen.

1.4 Grundprinzipien von Suchmaschinen

Suchmaschinen für das Web, Intranets oder digitale Bibliotheken basieren auf *Information-Retrieval*-Technologie (IR), insbesondere dem *Vektorraummodell*, bei dem Text- oder HTML-Dokumente als Feature-Vektoren repräsentiert sind. Dieser hochdimensionale Vektorraum wird aufgespannt von den durch Stammformreduktion auf sog. Terme normalisierten Wörtern, die im Korpus (d.h. dem Web oder Intranet) vorkommen; u.U. können auch Ankertexte von Hyperlinks oder weitergehende Aspekte der Hyperlinkstruktur als zusätzliche Features verwendet werden. Die Komponenten eines Dokumentvektors sind Feature-Gewichte, die die Signifikanz eines Features im Dokument ausdrücken und aufgrund der Häufigkeit im Dokument sowie im gesamten Korpus berechnet werden. Anfragen von Benutzern werden dann intern ebenfalls auf Vektoren abgebildet, und es werden die zur Anfrage ähnlichsten Dokumentvektoren ermittelt und zu einer nach Relevanz absteigend sortierten Rangliste zusammengesetzt, wobei verschiedene Ähnlichkeitsmetriken Verwendung finden. Dieses Grundprinzip kann auch auf multimediale Daten, z.B. Bilder oder Musikstücke, verallgemeinert werden, wobei dann natürlich andere Features von Bedeutung sind, die z.B. aus Transformationen von Signalen abgeleitet werden. Die Güte eines Suchresultats wird a posteriori empirisch bewertet: die *Präzision* der Anfrageauswertung ist der Anteil der wirklich relevanten Resultdokumente unter den ersten N (z.B. 10) Treffern der Rangliste, die *Ausbeute* ist der Anteil der überhaupt im Korpus vorhandenen relevanten Dokumente, den die Suche gefunden hat.

> *Grundprinzip 1:*
> Dokumente jeder Art werden als *Feature-Vektoren* in einem hochdimensionalen Datenraum repräsentiert.

> *Grundprinzip 2:*
> Eine Anfrage an eine Suchmaschine liefert als Ergebnis (einen Präfix) eine(r) Rangliste, die die Trefferdokumente nach Relevanz bzw. Ähnlichkeit zur Anfrage absteigend sortiert (*Relevanz-Ranking*).

Web-Suchmaschinen sind ihrer Funktionalität durch den Zwang zur Skalierbarkeit, der Unterstützung von Millionen von Benutzern bei der Suche auf Milliarden von Dokumenten, stark eingeschränkt. In anderen Umgebungen, in denen ebenfalls Information-Retrieval-Technologie verwendet wird, etwa in digitalen Bibliotheken oder Intranets, kommen u.U. ausdrucksstärkere und mathematisch anspruchsvollere Techniken zum Einsatz. Dazu zählen insbesondere sog. Spektralanalysen, die auf linearer Algebra beruhen, sowie statistische Lernverfahren. Mit solchen Techniken können Dokumentensammlung tiefergehender analysiert und besser organisiert werden, beispielsweise durch eine automatische Klassifikation von Dokumenten in eine gegebene Themenhierarchie.

> *Grundprinzip 3:*
> Durch *algebraische Analysen* und *statistiche Lernverfahren* können Dokumentsammlungen besser organisiert und bewertet werden..

Suchmaschinen werden nicht nur für die Internet-Suche durch Endbenutzer verwendet, sondern haben auch Programmierschnittstellen, um als Komponenten zu umfassendere Lösungen etwas bei der Generierung von *Portalen* oder der Suche in *Intranets* beizutragen. Dabei müssen sie in vielen Fällen

mit strukturierten Datenquellen interagieren, z.B. mit ERP-Datenbanken (Enterprise Resource Planning, z.B. SAP R/3) in Intranets oder mit Produktkatalogen oder Gen- und Proteindatenbanken im Internet. Im letzeren Fall, der Einbeziehung von Datenquellen, die selbst eine Suchschnittstelle in Form eines HTML-Formulars oder eines sog. Web Services anbieten und Ergebnisse als dynamisch generierte HTML- (oder XML-) Seiten liefern, spricht man auch von der Erschließung des *Deep Web* (Hidden Web). Es wird geschätzt, dass das Deep Web um den Faktor 100 größer ist als das durch normale Suchmaschinen (wie Google) abgedeckte Oberflächen-Web und dass die wirtschaftlich und wissenschaftlich wichtigsten Daten Teil des Deep Web sind. Ein Trend ist, dass Deep-Web-Daten nicht nur im HTML-Format (Hypertext Markup Language) präsentiert, sondern auch im expliziter strukturierten und reicher annotierten *XML-Format* (Extensible Markup Langauge) exportiert werden. Bei der Suche auf solchen Daten ist es oft nötig, Suchverfahren auf strukturierten Daten (z.B. technischen Eigenschaften von Produkten) und Textdaten (z.B. Produktbeschreibungen in natürlicher Sprache) zu integrieren.

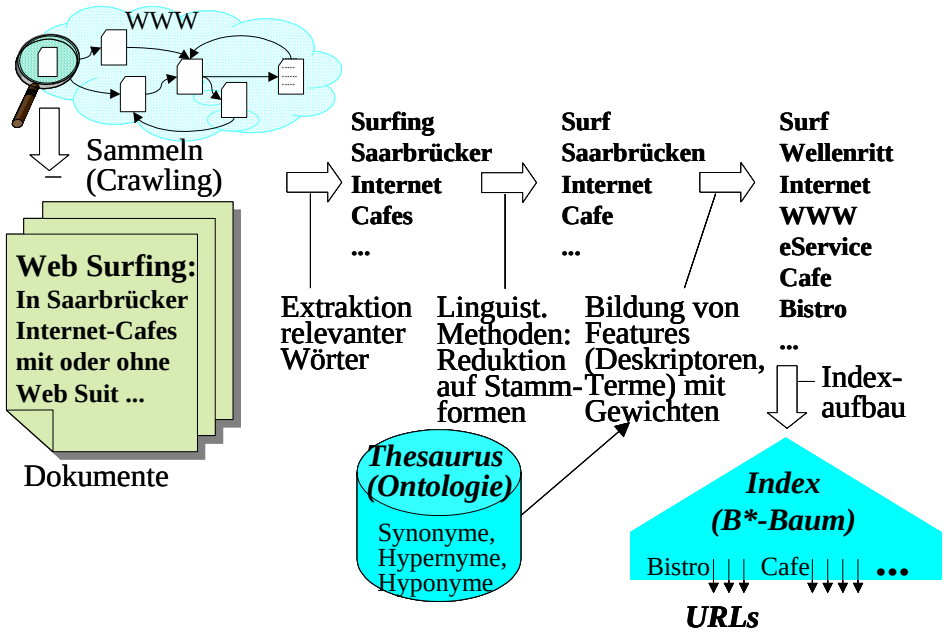
Ein langfristiges Ziel ist das *Semantic Web*, bei dem angestrebt wird, alle Informationen im Internet in einer einheitlichen logikbasierten Wissensrepräsentationssprache zu beschreiben. Damit würden alle Arten von digitalen Bibliotheken, wissenschaftlichen Datenarchiven, aktuellen Nachrichten und Unterhaltungsmedien global und präzise suchbar, doch bis dahin ist es wohl noch ein langer und schwieriger Weg, auf dem sich etliche wissenschaftliche Herausforderungen und spannende Forschungsfragen stellen.

Kapitel 2: Suchmaschinen für Intranets und das Web

2.1 Information-Retrieval-Systeme

Information-Retrieval-Systeme (IRS) verwalten große Sammlungen unstrukturierter Daten oder semistrukturierter Daten, insbesondere Textdokumente und HTML-Dokumente, und unterstützen vor allem die Suche nach relevanten Daten zu einem spezifizierten Thema bzw. ähnlichen Dokumenten zu einer spezifizierten Anfrage (einem "Musterdokument"). IRS-Funktionalität findet sich in Suchmaschinen für das Web und für Intranets, in digitalen Bibliothekssystemen, Mail-Servern und z.T. auch integriert oder als Erweiterungskomponente in objekt-relationalen Datenbanksystemen. Das Prinzip der Ähnlichkeitssuche ist auch für Multimedia-Anwendungen von großer Bedeutung, z.B. für elektronische Museen u.ä.

Die folgende Abbildung gibt einen Überblick über den Aufbau und die Arbeitsweise eines IRS:



Ein *Crawler* traversiert Dokumente, im Web und in Intranets durch Verfolgen von Hyperlinks mittels HTTP-Aufrufen, und lädt sie zur Inhaltserschließung auf den IRS-Server. Eine oberflächliche *Textanalyse* extrahiert aus einem Dokument alle relevanten Wörter, ggf. durch sog. *Stoppwortelimination*, das Entfernen von "Füllwörtern" wie Artikel, Präpositionen, Konjunktionen. Mit einfachen linguistischen Verfahren, sog. *Stemming*-Verfahren, werden Wörter auf ihre jeweiligen Stammformen reduziert (z.B. Plural auf Singular, Dativ auf Nominativ, Partizip Perfekt auf Infinitiv eines Verbs). Die so entstandene Menge von für das Dokument signifikanten Wortstammformen wird auch als Menge von *Termen*, *Deskriptoren* oder *Features* bezeichnet. Ggf. kann diese Menge durch Nachschlagen in einem Wörterbuch, einem sog. *Thesaurus* bzw. einer sog. *Ontologie* (z.B. WordNet), um Synonyme oder eng verwandte Unterbegriffe (Hyponyme) und Oberbegriffe (Hypernyme) erweitert werden. Die in einem Dokument vorkommenden Terme werden typischerweise *gewichtet*, und zwar aufgrund ihrer Vorkommenshäufigkeit und/oder aufgrund ihrer Position und Präsentation im Dokument (z.B. in einer Überschrift, in einem großen Font, usw.); Techniken zur Gewichtung werden in

Abschnitt 2.2 genauer betrachtet. Schließlich wird ein *Index* über diesen Termen als Suchschlüssel aufgebaut, typischerweise in Form eines B*-Baums (oder eines sonstigen Suchbaums für Sekundärspeicher), der mit jedem Term eine Menge von Dokument-Ids bzw. URLs sowie die Gewichte des Terms in den jeweiligen Dokumenten verbindet.

Anfragen an ein IRS sind - in der am meisten verbreiteten Form - zunächst einfache Mengen oder Listen von Termen, z.B. "Java Lava"; bei einer Interpretation als Liste werden die Terme der Anfrage unterschiedlich stark gewichtet. Das Resultat ist entweder eine Menge von Trefferdokumenten, die alle angegebenen Terme enthalten, oder eine Rangliste von zur Anfrage ähnlichen Dokumenten, die nach einem Ähnlichkeitsmaß bzw. einem Relevanz-Score (auch RSV = Retrieval Status Score genannt) absteigend sortiert ist. Ersteres nennt man *Boolesches Retrieval*, letzteres *Ranked Retrieval*. Beim Ranked Retrieval qualifizieren sich oft auch Dokumente, die nur einen Teil der Terme der Anfrage enthalten. Darüber hinaus werden u.U. auch Dokumente berücksichtigt, die keinen der Suchterme wörtlich enthalten, aber Synonyme dazu oder semantisch eng verwandte Begriffe enthalten; für diese Art des Semantischen Retrievals benötigt man einen Thesaurus bzw. eine Ontologie.

Erweiterte Formen von Anfragen erlauben die Angabe negativer Terme, die als Ausschlusskriterium interpretiert werden, z.B. "Java Lava -Coffee", wenn man keine Dokumente über die "heißesten Kaffeesorten" bekommen möchte. Darüber hinaus sind Boolesche Kombinationen von Suchbedingungen möglich und zusätzliche Bedingungen, die auf die Nachbarschaft bzw. textuelle Distanz von Wörtern abzielen, z.B. "Java NEAR Lava". Suchbedingungen dieser Art sind z.T. auch in SQL-Systemen integriert.

Bei einer Anfrage, die n verschiedene Terme enthält, extrahiert das IRS zunächst die zu diesen Termen gehörigen URL- bzw. Dokument-Id-Listen aus seinem Index. Diese bilden eine Kandidatenmenge. Anschließend werden für alle Kandidaten auf der Basis ihrer Termgewichte Ähnlichkeitsmaße zur Anfrage berechnet; dabei wird die Struktur der Termkombinationen in der Anfrage wie z.B. negative Terme oder diskunktive vs. konjunktive Verknüpfung berücksichtigt. Auf diese Weise werden Kandidaten entfernt, und es werden Relevanz-Scores und das Ranking der verbleibenden Trefferdokumente berechnet.

Gütemaße

Die Güte eines IRS, also seines Fähigkeit, zu einer Anfrage möglichst relevante und nur relevante Dokumente zurückzuliefern, wird vor allem mit den folgenden beiden Metriken bewertet:

Fähigkeit, zu einer Anfrage *nur* relevante Dokumente zu liefern:

Präzision einer Anfrage (engl.: *precision*) = $\frac{Anzahl\ relevanter\ Dokumente\ unter\ Top\ r}{r}$

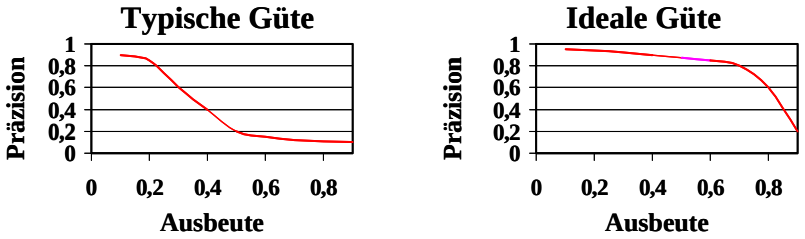
Fähigkeit, zu einer Anfrage *alle* relevanten Dokumente zu liefern:

Ausbeute einer Anfrage (engl.: *recall*) = $\frac{Anzahl\ relevanter\ Dokumente}{Anzahl\ aller\ relevanten\ Dokumente\ im\ Korpus}$

Dabei wird r typischerweise auf 10 oder 20 gesetzt, also denjenigen Präfix der Resultatsrangliste, den sich ein Benutzer in der Regel anschaut. Die Auswertung der beiden Gütemaße erfordert eine intellektuelle Bewertung der Anfrageresultate bzw. des gesamten Korpus (aller Dokumente, die das IRS verwaltet). Dies wird für spezielle Benchmark-Dokumentkollektionen (z.B. die TREC-

Benchmarks) gemacht, und verschiedene IRS werden dann anhand von Anfragemixturen auf diesen Daten getestet.

Zwischen Präzision und Ausbeute gibt es einen inhärenten Zielkonflikt. Ein ideales IRS würde für beide Maße Werte in der Nähe von 1,0 erzielen, aber in der Praxis wird eine hohe Präzision oft um den Preis einer schwächeren Ausbeute und eine hohe Ausbeute oft um den Preis einer schlechteren Präzision erkauft.



In Benchmark-Resultaten wird oft die Präzision als Funktion der erzielten Ausbeute dargestellt. Wenn man das Benchmark-Ergebnis auf eine einzige Zahl kompakt reduzieren möchte, wird entweder der Precision-Recall-Breakeven-Point verwendet, also der Punkt auf der Kurve, bei dem Präzision und Ausbeute gleich sind, oder die sog. uninterpolierte durchschnittliche Präzision. Letzeres ist die durchschnittliche Präzision gemittelt über alle Rangplätze, auf denen ein relevanter Treffer steht. Ein weiteres kompaktes Gütemaß ist das sog. F-Maß, das harmonische Mittel aus Präzision und Ausbeute (für eine bestimmte Top-r-Menge): $\frac{1}{\frac{1}{Pr\ddot{a}zision} + \frac{1}{Ausbeute}}$.

2.2 Web-Crawling und Indexierung

Eine Suchmaschine für das Web – oder auch für große Intranets – besteht aus den folgenden Komponenten:

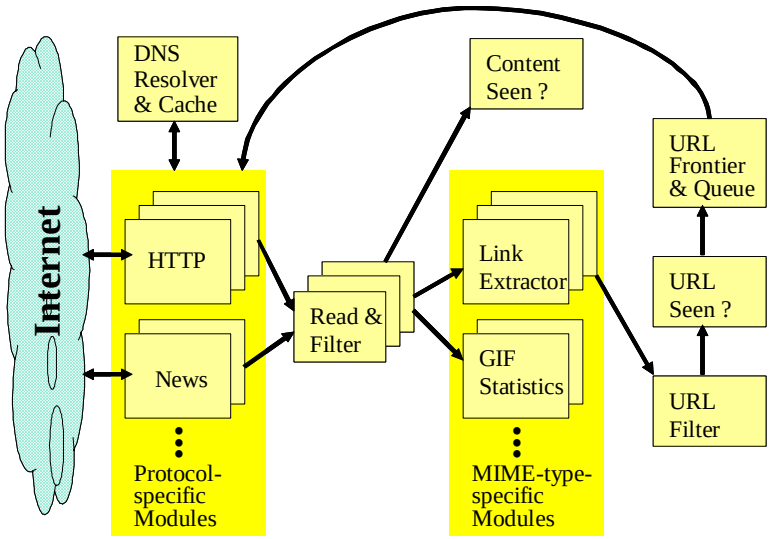
- **URL Queue Server** zur Verwaltung einer Priority-Queue, in der zu Beginn Start-URLs stehen und später die aus Links von besuchten Webseiten extrahierten URLs von Seiten, die selbst noch nicht besucht wurden. Die Priorisierung der URLs hängt von vielen Parametern ab, z.B. von der Zeit des letzten Besuchs, vom Domain-Namen, von Eigenschaften der entsprechenden Webseite (z.B. dem Eingangsgrad, sofern dieser durch frühere Crawls geschätzt werden kann), usw.
- **Crawler (Robot, Spider)** zum Zugriff auf Web-Seiten, in der Regel mit dem HTTP-Protokoll (Hypertext Transfer Protocol) unter Beachtung verschiedener Nebenbedingungen (z.B. Einschränkungen des Typs zu besuchender Web-Seiten, Beachtung der durch das Robot Exclusion Protocol gegebenen „Etikette“, usw.).
- **Repository Server** zur Verwaltung der durch den Crawler geladenen Dokumente (z.B. in einer Tabelle *DocumentRepository* in einer Datenbank).
- **Indexer (inkl. Parser, Stemmer)** zur Analyse von (überwiegend HTML-) Dokumenten und zum Aufbau eines *DocumentIndex* mit Anlegen von Einträgen in *Lexicon* (Wörterbuch für alle Terme) und *Anchors* (Anker von Hyperlinks).
- **URL Resolver** zur Umsetzung von URLs in interne DocIds.

- **Link Analyzer** zur Analyse von Hyperlinkstrukturen und entsprechenden Berechnung von Autoritätsmaßen (siehe Kapitel 4).
- **Query Processor** zur Auswertung von Anfragen, typischerweise einfach Mengen von Keywords**,** durch Nachschlagen im Index und Berechnung einer Rangliste von Treffern mit Hilfe statistischer Maße (siehe Abschnitte 2.3 und 2.4).

Diese Komponenten verwenden die folgenden Datenstrukturen, die z.B. als Tabellen in einer Datenbank realisiert sein könnten. Was genau „Tabellen" einer Datenbank sind, wird in Kapitel 5 erklärt; im Moment genügt eine intuitive Vorstellung für das Verständnis. Große Suchmaschinen wie Google oder Inktomi verwenden eigene, auf Files basierende, Datenstrukturen, die über viele Platten und Rechner verteilt sind.

- **DocumentRepository** ***(DocId, DocContent)*** mit dem vollständigen Text aller (HTML-) Seiten, ggf. in komprimierter Form.
- **Lexicon** ***(TermId, Term)*** mit allen Wortstämmen, die als Indexterme verwendet werden.
- **DocumentIndex** ***(DocId, TermId, Weight, ...)*** mit allen Vorkommen von Termen in Dokumenten sowie entsprechenden Gewichten, die auf Wortstatistiken beruhen (siehe Abschnitt 2.3), optimiert für den schnellen Zugriff über DocIds.
- **TermIndex** ***(TermId, DocId, Weight, ...)*** mit allen Dokumenten zu allen Termen, optimiert für den schnellen Zugriff über TermIds (sog. *invertierte Listen* im IR-Jargon).
- **Anchors** ***(SourceDocId, TargetDocId, AnchorText)*** mit allen bekannten Hyperlinks.
- **URLIndex** ***(URL, DocId)*** zur Umsetzung von URLs auf interne DocIds und umgekehrt.

Die folgende Abbildung illustriert die Arbeitsweise eines Web-Crawlers. Die Architektur ist bezüglich der verwendeten Protokolle (HTTP, News, SOAP für Web Services, etc.) und der Formattypen von Web-Seiten (HTML, PDF, XML, etc.) erweiterbar. Alle Komponenten können im Multithreading-Modus betrieben werden, also mit mehreren parallelen Instantiierungen einer Komponente. Aus Effizienzgründen haben Web-Crawler typischerweise eine eigene Komponente zur Übersetzung von Domain-Namen in IP-Adressen und versuchen, aufwändigere Aufrufe von DNS-Servern (Domain Name Service) zu minimieren. Ebenfalls aus Effizienzgründen ist es sehr wichtig, beim Crawling Seiten zu erkennen, die im aktuellen Crawl bereits besucht wurden. Dazu müssen einerseits besuchte URLs schnell erkannt werden (z.B. mittels Nachschlagen in einer Hashtabelle), und andererseits müssen auch Inhalte mit früher besuchten Seiten verglichen werden (z.B. durch Vergleich mit einer kompakten „Fingerprint"-Signatur), weil etliche Seiten über verschiedene Pfade erreichbar, unter mehreren URLs registriert und vielfach kopiert sind.



Große Web-Suchmaschinen wie Google verwalten in ihrem Index mehr als 4 Milliarden Web-Seiten mit mehr als 10 Millionen Termen. Die Rohdaten umfassen eine Größenordnung von 20 Terabytes, der Index alleine etwa 4 Terabytes. Daten und Index sind über mehr als 10 000 PCs verteilt, partitioniert und aus Verfügbarkeits- und Lastabalancierungsgründen stark repliziert. Eine solche Suchmaschine muss mehr als 200 Millionen Queries pro Tag bedienen, wobei Benutzer eine Antwortzeit von nicht mehr als 1 oder 2 Sekunden erwarten. Der Crawler muss mehr als 1000 Seiten pro Sekunde verarbeiten, und alle Seiten sollten spätestens alle 30 Tage erneut besucht bzw. geprüft werden.

2.3 Vektorraummodell für IR mit Ranking

Im Vektorraummodell nach G. Salton werden Dokumente als Vektoren in einem Feature-Raum repräsentiert. Features, also Vektorraumdimensionen, entsprechen den Termen, die in dem Korpus vorkommen, also den Wörtern nach Stoppwortelimination, Stammformreduktion und ggf. weiteren Abstraktionsschritten (siehe Abschnitt 2.1). Die Komponenten des Vektors eines Dokuments sind entweder binär und signalisieren dann schlicht das Vorkommen eines bestimmten Terms im Dokument oder reelle Zahlen aus dem Intervall [0,1] und signalisieren dann die relative Wichtigkeit oder Häufigkeit des Terms im Dokument. Diese Termgewichte werden nach bestimmten Heuristiken berechnet (siehe unten). Bei einer Feature-Menge F hat der entsprechende Vektorraum die Dimensionalität |F|, und die Vektoren der Form (0...0 1 0 ... 0), bei denen genau ein Feature den Wert 1 hat und alle anderen den Wert 0, bilden eine Basis des Vektorraums.

Anfragen sind in diesem Modell ebenfalls Vektoren im selben Featureraum. Die in der Anfrage vorkommenden Terme werden durch eine 1-Komponente im Vektor repräsentiert, alle anderen Komponenten sind 0. Wenn durch die Reihenfolge der Terme in der Anfrage implizit Gewichte ausgedrückt sein sollen (also die wichtigsten Suchterme vorne stehen müssen), können die Komponenten des Anfragevektors auch geeignet auf das Intervall [0,1] abgebildet werden. Wenn negative Terme angegeben sind, also Wörter, die in den Trefferdokumenten (möglichst) nicht vorkommen sollen, kann man auch das Intervall [-1,1] verwenden. In der Praxis werden solche Negativkriterien aber nicht im Ranking selbst behandelt, sondern als zusätzlicher Filterschritt auf der Rangliste der Kandidaten gemäß der positiven Suchterme realisiert.

Im Vektorraummodell können nun Ähnlichkeiten zwischen Dokumenten und Anfragen (oder auch zwischen verschiedenen Dokumenten) berechnet werden. Diese *Ähnlichkeitswerte*, die oft auch *Relevanz-Scores* oder *Retrieval-Status-Values (RSVs)* genannt werden, sind die Grundlage des Ran-

king: das Resultat ist eine nach Ähnlichkeit zur Anfrage absteigend sortierte Liste von Dokumenten. Das (zumindest als Basis für weitere Variationen) am häufigsten verwendetste Ähnlichkeitsmaß ist das *Cosinus-Maß*:

Für eine Featuremenge F, eine Dokumentmenge D, ein Dokument $d \in D \subseteq [0,1]^{|F|}$ und eine Anfrage $q \in [0,1]^{|F|}$ ist die Ähnlichkeit von d und q: $sim(d,q) = \frac{\sum_{j=1}^{|F|} d_j \, q_j}{\sqrt{\sum_{j=1}^{|F|} d_j^2} \sqrt{\sum_{j=1}^{|F|} q_j^2}}$.

Die Termgewichte der Dokumentvektoren werden in der Regel aufgrund von sog. tf*idf-Formeln bestimmt. Dabei ist tf(t,d) die Häufigkeit von Term t in Dokument d (engl. term frequency = tf), und idf(t) ist der Kehrwert der Häufigkeit des Terms t im gesamten Korpus (engl. inverse document frequency = idf). Die Idee dahinter ist, dass das Gewicht von t in d mit seiner Häufigkeit in d wachsen soll, aber mit der Häufigkeit von t im gesamten Korpus fallen soll. Lokal häufige Terme werden also als signifikant für den Inhalt eines Dokuments angesehen, während global häufige Terme eher inflationären Charakter und daher nur noch geringe Aussagekraft über den Inhalt eines Dokuments haben. In der Praxis werden beide Größen – tf und idf – ggf. noch normalisiert (z.B. auf Werte zwischen 0 und 1), und der idf-Wert wird in der Regel logarithmisch gedämpft, da er sonst das Produkt dominieren würde. Eine typische Variante der tf*idf-Formel für das Termgewicht w_{ij} des i-ten Terms im j-ten Dokument ist z.B.:

$$w_{ij} := \frac{tf_{ij}}{max_k \, tf_{kj}} \; log \, \frac{N}{df_i}, \text{ wobei}$$

tf_{ij} die Häufigkeit von Term i in Dokument j,

N die Anzahl der Dokumente im Korpus D und

df_i die Häufigkeit von Term i im gesamten Korpus sind.

Wenn man nicht das Cosinus-Maß als Ähnlichkeitsfunktion verwendet, sonderen andere Metriken, kann auch eine Normalisierung der w_{ij}-Werte auf die Dokumentvektorlänge 1 Sinn machen:

$$\omega_{ij} := w_{ij} \, / \sqrt{\sum_k w_{kj}^2}$$

In Web-Suchmaschinen werden bei den Termgewichten außerdem die Position und Formatierung von Wörtern (z.B. in Überschriften, im Fettdruck oder in einem Hyperlink-Anker) berücksichtigt.

Relevanz-Feedback

Nachdem der Benutzer die ersten r Dokumente der Rangliste einer Query inspiziert hat, kann er durch Markieren der relevanten und irrelevanten Dokumente dem IRS Feedback geben, aufgrund dessen in einer weiteren Verfeinerungs- bzw. Präzisierungs-Query explizite Gewichte in der Query gesetzt werden. Im einfachsten Fall kann dies so realisiert werden, dass der Benutzer den besten Treffer auswählt und eine Query startet, die genau die Deskriptoren dieses Dokuments mit seinen entsprechenden Gewichten enthält.

Die bekannteste Methode für Relevanz-Feedback ist das *Rocchio-Verfahren*. Seien D^+ und D^- Resultatdokumente einer Query q, die der Benutzer positiv bzw. negativ markiert. Dann wird daraus die verfeinerte Query q' mit Gewichten $\alpha, \beta, \gamma \in [0,1]$ (meist mit $\alpha > \beta > \gamma$) wie folgt gebildet:

$$q' = \alpha q + \frac{\beta}{|D^+|} \sum_{d \in D^+} d - \frac{\gamma}{|D^-|} \sum_{d \in D^-} d$$

Alternativ könnten - in einer längeren Recherche-Sitzung auch benutzer- bzw. sitzungsspezifische Dokumentgewichte geführt und je nach Feedback angepasst werden, was aber wesentlich aufwendiger zu implementieren ist.

2.4 Anfrageausführung mit Ranking

Die einfachste, aber mit Abstand wichtigste und häufigste, Form von Anfragen sind Mengen von Keywords $q = t_1 ... t_z$ mit Termen t_i. Im Vektorraummodell sind dies Vektoren, bei denen die Komponenten für $t_1 ... t_z$ Gewicht 1 und alle anderen Komponenten Gewicht 0 haben. Da Treffer mit hohem Score nicht notwendigerweise alle z Terme enthalten müssen, ergibt sich folgender, auf den Indexlisten für $t_1 ... t_z$ arbeitender Algorithmus.

Naiver Algorithmus:

```
candidate-docs := ∅;
for i=1 to z do {
    candidate-docs := candidate-docs ∪ index-lookup(ti) };
for each dj ∈ candidate-docs do {compute score(q,dj)};
sort candidate-docs by score(q,dj) descending;
return k docs from candidate-docs with highest scores;
```

Der Algorithmus ist naiv, weil er einfach alle möglichen Treffer ermittelt und erst danach Scores berechnet, um am Ende nur die besten k (z.B. k=10) Treffer auszugeben. Bei sehr langen Indexlisten – mit Zehn- oder Hunderttausenden von DocIds – und bei sehr vielen Anfragetermen (z.B. bei vom System erweiterten Anfragen oder Anfragen, die aus markierten Dokumenten konstruiert werden) ist dieser Aufwand unakzeptabel.

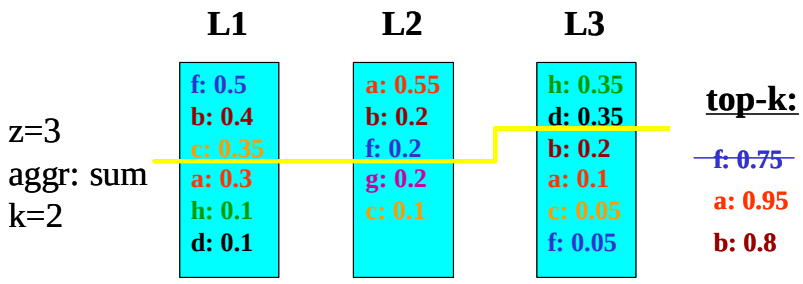
Ein guter Algorithmus sollte es vermeiden, die Indexlisten komplett zu lesen, und der Aufwand sollte in k beschränkt sein. In der IR-Literatur sind zu diesem Zweck zahlreiche Heuristiken vorgeschlagen worden. Ein neuerer, vielseitig einsetzbarer Algorithmus, der garantiert die Top-k-Treffer berechnet und in seiner Effizienz asymptotisch optimal ist, ist der *Threshold-Algorithmus (TA)* von Fagin et al. Dieses Verfahren arbeitet auf z Indexlisten $L_1, ..., L_z$, die jeweils die Treffer zu Anfragetermen $t_1 ... t_z$ enthalten. Zu jeder Indexliste L_i gibt es lokale Scores s_i, und es wird angenommen, dass die Liste nach absteigenden s_i-Werten sortiert ist. Ferner wird angenommen, dass der Gesamt-Score s eines Trefferdokuments eine monotone Aggregationsfunktion aggr seiner lokalen Scores ist, d.h. für alle Dokumente d', d'' gilt:

$s_1(d') \leq s_1(d'') \wedge ... \wedge s_z(d') \leq s_z(d'') \Rightarrow s(d')=aggr(s_1(d'), ..., s_z(d')) \leq aggr(s_1(d''), ..., s_z(d''))=s(d'')$. Letzeres ist z.B. erfüllt, wenn s eine gewichtete Summe der s_i ist, aber auch max und zahlreiche andere Funktionen erfüllen das Monotoniekriterium.

TA traversiert alle z Indexlisten L_1, ..., L_z in verschränkter Form (Round-Robin), liest also erst den besten Treffer von L_1, dann den von L_2, usw., bis hin zum besten Treffer von L_z und dann weiter mit dem zweitbesten von L_1, usw. Die Listen werden also primär mit sortierten Zugriffen (*Sorted Access*) verarbeitet. Zu jedem auf diese Weise gelesenen Dokument d schaut TA in allen anderen Listen per wahlfreiem Zugriff (*Random Access*) die lokalen Scores nach und berechnet dann direkt den Gesamt-Score von d. Die besten k bislang gefundenen Dokumente werden in einer Top-k-Liste festgehalten. In jedem Schritt merkt sich TA außerdem den lokalen Score des zuletzt gelesenen Eintrags der Liste L_i in einer Variablen $high_i$, die eine obere Schranke für den bestmöglichen lokalen Score von noch nicht gelesenen Dokumenten ist. Das Verfahren terminiert, bricht also die Index-Scans ab, wenn der aufgrund der $high_i$-Werte bestmögliche Gesamt-Score eines noch unbekannten Dokuments, $aggr(high_1, ..., high_z)$, nicht besser ist als der schlechteste Gesamt-Score unter den aktuellen Top-k. In der Literatur wurde gezeigt, dass TA bei n Einträgen pro Indexliste L_i mit hoher Wahrscheinlichkeit höchstens $O(n^{m-1/m} \cdot k^{1/m})$ Schritte benötigt und typischerweise sehr viel weniger.

Algorithmus TA:
scan all lists L_i (i=1..z) in parallel:
  consider dj at position pos_i in Li;
  $high_i := s_i(dj)$;
  if dj $\notin$ top-k then {
    look up $s_\nu(dj)$ in all lists L_ν with $\nu \neq i$; // random access
    compute $s(dj) := aggr \{s_\nu(dj) \mid \nu=1..z\}$;
    if s(dj) > min score among top-k then
      add dj to top-k and remove min-score d from top-k; };
  if min score among top-k $\geq aggr \{high_\nu \mid \nu=1..z\}$ then exit;

Beispiel:



Wenn TA alle Listenelemente oberhalb der gelben Linie gelesen hat, ist der Schwellwert für den Abbruch 0.35+0.2+0.35=0.9. Da dieser Wert höher ist als der schlechteste Gesamt-Score unter den aktuellen Top-k, nämlich 0.8 für Dokument b, muss TA noch ein weiteres Listenelement lesen, und zwar den Eintrag b:0.2 aus Liste L3. Damit wird der Schwellwert neu berechnet, und zwar zu 0.35+0.2+0.2=0.75, und der Algorithmus terminiert.

Da wahlfreie Zugriffe auf den Indexlisten in vielen Anwendungssituationen sehr viel teurer sind als die sortierten Zugriffe, sollten sie minimiert werden. Eine Variante von TA, die ganz ohne wahlfreie Zugriffe auskommt, ist *TA-sorted* (TA-NRA für No Random Access). Der Algorithmus führt über jedes gelesene Dokument dj Buch, indem er sich folgende Werte merkt:

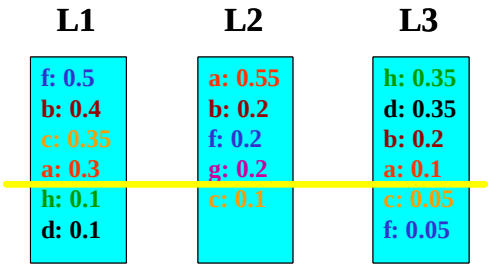
- E(dj): die Menge der für den Gesamt-Score von dj ausgewerteten Listen (für die also der lokale Score bekannt ist),
- worstscore(dj) = aggr{x1, ..., xz) mit xi := si(q,dj) für i∈E(dj), 0 für i ∉E(dj)
- bestscore(dj) = aggr{x1, ..., xz) mit xi := si(q,dj) für i∈E(dj), $high_i$ für i ∉E(dj);

TA-sorted kennt in jedem Schritt die aufgrund ihrer worstscore-Werte aktuell besten Top-k-Dokumente und merkt sich alle Kandidaten d, für die bestscore(d) besser ist als der worstscore des schlechtesten Dokuments unter den Top-k. Unter den Kandidaten wird auch ein virtuelles Dokument d mit E(d)=∅ geführt. Kandidaten, deren bestscore unter den worstscore des schlechtesten Top-k-Dokuments fällt, können eliminiert werden. Der Algorithmus terminiert, wenn die Kandidatenliste leer geworden ist.

Algorithmus TA-sorted:
scan index lists in parallel:
  consider dj at position pos_i in Li;
  E(dj) := E(dj) ∪ {i}; $high_i$:= si(q,dj);
  bestscore(dj) := aggr{x1, ..., xz)
    with xi := si(q,dj) for i∈E(dj), $high_i$ for i ∉E(dj);
  worstscore(dj) := aggr{x1, ..., xz)
    with xi := si(q,dj) for i∈E(dj), 0 for i ∉E(dj);
  top-k := k docs with largest worstscore;
  if min worstscore among top-k ≥ bestscore{d | d not in top-k}
  then exit;

Beispiel:

m=3
aggr: sum
k=2



top-k:
a: 0.95
b: 0.8

candidates:
~~f: 0.7 + ? ≤ 0.7 + 0.1~~
h: 0.35 + ? ≤ 0.35 + 0.5
~~c: 0.35 + ? ≤ 0.35 + 0.3~~
d: 0.35 + ? ≤ 0.35 + 0.5
~~g: 0.2 + ? ≤ 0.2 + 0.4~~

Die Abbildung gibt den Zustand von TA-sorted an, wenn der Algorithmus alle Listenelemente oberhalb der gelben Linie gelesen hat. Zu diesem Zeitpunkt könnten sich noch h und d für die Top-k qualifizieren. Im nächsten Schritt liest der Algorithmus den L1-Eintrag h:0.1. Dadurch ändern sich der worstscore und der bestscore von h zu 0.35+0.1+? = 0.45+? ≤ 0.45+0.2=0.65, so dass h eliminiert werden kann. Im selben Schritt ergibt sich für d die Änderung 0.35+? ≤ 0.35+0.3=0.65, so dass auch d sich nicht mehr für die Top-k qualifizieren kann. Die Kandidatenliste ist dann leer, und TA-sorted terminiert.

2.5 Grundlagen aus der Linearen Algebra

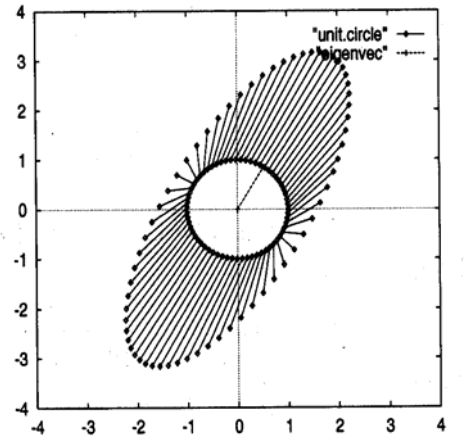
Eine Menge S von Elementen eines Vektorraums heißt **linear unabhängig**, wenn kein Vektor $x \in S$ als Linearkombination anderer Vektoren aus S darstellbar ist, es also keine $y_1, ..., y_k \in S-\{x\}$ gibt, so dass
$x = a_1 y_1 + ... + a_k y_k$ mit reellen Zahlen $a_1, ..., a_k$.
Der **Rang** einer Matrix A ist die maximale Anzahl linear unabhängiger Zeilenvektoren oder linear unabhängiger Spaltenvektoren. Eine **Basis** einer n×n-Matrix A ist eine Menge S linear unabhängiger Zeilenvektoren (oder Spaltenvektoren), so dass alle Zeilen (bzw. Spalten) von A als Linearkombinationen von Vektoren aus S darstellbar sind.
Eine Basis S von A heißt **Orthonormalbasis** wenn für alle x, y ∈S gilt:

$$\|x\|_2 := \sqrt{\sum_{i=1}^{n} x_i^2} = 1 = \|y\|_2 \quad und \quad x \cdot y = 0$$

Sei A eine reellwertige n×n-Matrix, x ein reellwertiger n×1-Vektor und λ ein reellwertiger Skalar. Lösungen x und λ der Gleichung $A \times x = \lambda x$ heißen **Eigenvektor** und **Eigenwert** von A. Eigenvektoren von A sind Vektoren, deren Richtung bei der durch A beschriebenen affinen Abbildung erhalten bleibt.

Beispiel:

Die Matrix $A = \begin{pmatrix} 2 & 1 \\ 1 & 3 \end{pmatrix}$ beschreibt eine affine Abbildung von Vektoren der Ebene: $x \mapsto Ax$. Sie bildet z.B. die Punkte des Einheitskreises auf eine gedreht Ellipse ab. Die Eigenvektoren sind x1 = $(0.52\ 0.85)^T$ zum Eigenwert λ1 = 3.62 und x2 = $(0.85\ -0.52)^T$ zum Eigenwert λ2 = 1.38. Die folgende Abbildung illustriert dieses Beispiel; die Eigenvektoren entsprechen den Richtungen der Ellipsenachsen.



Die Eigenwerte von A sind Nullstellen des **charakteristischen Polynoms** f(λ) von A:
$f(\lambda) = |A - \lambda I| = 0$

mit der Determinante $|A| = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \left|A^{(ij)}\right|$, wobei $A^{(ij)}$ eine Matrix bezeichnet, die aus A durch Streichen der i-ten Zeile und j-ten Spalte abgeleitet ist.

Eine reellwerte n×n-Matrix A heißt **symmetrisch,** wenn $a_{ij}=a_{ji}$ für alle i, j. A heißt **positiv definit**, wenn für alle n×1-Vektoren $x \neq \mathcal{O}$: $x^T \times A \times x > 0$.

Satz:
Wenn A symmetrisch ist, dann sind alle Eigenwerte von A reell. Wenn A symmetrisch und positiv definit ist, dann sind alle Eigenwerte positiv.

Spektralsatz (Hauptachsentransformation, Principal Component Analysis, PCA, Karhunen-Loeve-Transformation):
Sei A eine symmetrische n×n-Matrix mit Eigenwerten $\lambda_1, ..., \lambda_n$ und Eigenvektoren $x_1, ..., x_n$, so dass $\|x_i\|_2 = 1$ für alle i. Die Eigenvektoren bilden eine Orthonormalbasis von A.
Dann gilt: $D = Q^T \times A \times Q$, wobei D eine Diagonalmatrix ist mit den Diagonalelementen $\lambda_1, ..., \lambda_n$ und Q aus den Spaltenvektoren $x_1, ..., x_n$ besteht. Die Vektoren $x_1, ..., x_n$ heißen in diesem Kontext Hauptachsen (Principal Components).
Man kann leicht zeigen, dass daraus $A = Q \times D \times Q^T$ folgt.

2.6 Latent Semantic Indexing (LSI)

Das einfache Vektorraum-Modell ist in (mindestens) zweierlei Hinsicht zu kritisieren:

- Da es u.U. starke Korrelationen im Vorkommen verschiedener Features in den Dokumenten gibt, ist die Dimensionalität des Feature-Vektorraums eigentlich unnötig hoch.
  Beispiel:
  Wenn "Web" und "Internet" nahezu immer zusammen vorkommen, braucht man nur eines dieser beiden Features zu indexieren.
- Dasselbe Feature kann je nach vorkommender Kombination mit anderen Features eine andere Semantik haben, so daß solche Korrelation explizit berücksichtigt sein sollten.
  Beispiel:
  Wenn "Java" und "Library" zusammen auftreten, ist die Bedeutung von "Java" vermutlich die Programmiersprache Java; das Dokument behandelt, also im weiteren Sinne das Thema Internet. Wenn "Java", "Kona Blend" und "Mokka" zusammen auftreten, handelt das Dokument mit hoher Wahrscheinlichkeit von Kaffee. Wenn schließlich "Java", "Sumatra" und "Borneo" zusammen auftreten, betrifft das Dokument vermutlich das Land Indonesien.

Die Idee von LSI ist, solche Korrelationen auszunutzen, um von den Featurekombinationen auf Themen (Topics) bzw. Konzepte (Concepts) zu schließen, die den Inhalt von Dokumenten charakterisieren. Die Anzahl verschiedener Themen ist typischerweise deutlich kleiner als die der Features. Wenn man das Vektorraum-Modell auf Themen anwendet statt auf Features, erzeugen die Themen einen Vektorraum mit deutlich niedrigerer Dimensionalität. In den Themen kommt also die "latente Semantik" der Dokumente klarer zum Ausdruck.

Gegeben seien:
- eine Featuremenge F mit $|F| = m$
- eine Dokumentmenge D mit $|D|=n$ und $D \subseteq [0,1]^m$
- eine Query $q \in [0,1]^m$

D kann also als m×n-Matrix A mit Werten aus dem Intervall [0,1] interpretiert werden und q als m×1-(Spalten)Vektor.

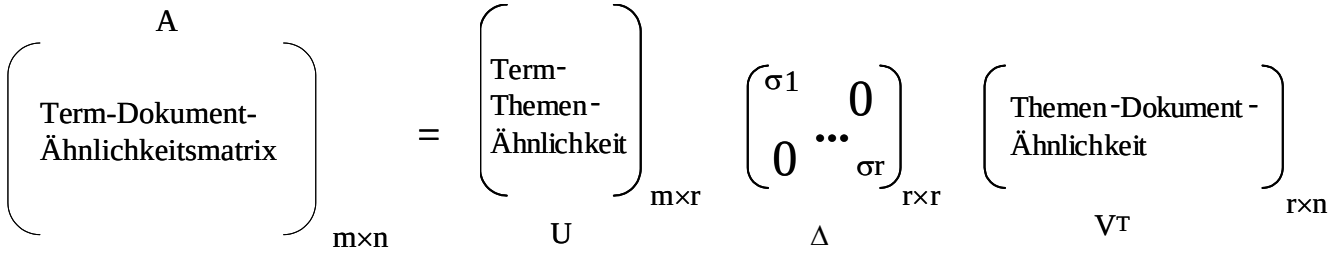
Gesucht wird:
eine "möglichst gute" Abbildung von D und q in einen Themen-Vektorraum
mit Dimensionalität $k << m$

Lösung:
Die Lösung besteht aus einer auf der sogenannten Singulärwertdekomposition von A beruhenden Transformation von A und q in einen automatisch "erzeugten" Vektorraum niedrigerer Dimensionalität. Dabei werden die folgenden mathematischen Eigenschaften der Singulärwertdekomposition ausgenutzt, die den Spektralsatz der Linearen Algebra verallgemeinern.

Satz:
Jede reellwertige n×m-Matrix A mit Rang r kann zerlegt werden in die Form

$$A = U \times \Delta \times V^T$$

mit einer n×r-Matrix U mit orthonormalen Spaltenvektoren, einer r×r-Diagonalmatrix D und einer m×r-Matrix V mit orthonormalen Spaltenvektoren.
Diese Zerlegung heißt *Singulärwertdekomposition* (engl.: Singular Value Decomposition, kurz: SVD) und ist unter der Annahme, daß die Elemente von Δ geordnet sind, eindeutig bestimmt.



Satz:
In der Singulärwertdekomposition $A = U \times \Delta \times V^T$ der Matrix A sind U, Δ und V wie folgt bestimmt:
- Δ besteht aus den Singulärwerten von A, d.h. den positiven Wurzeln der Eigenwerte von $A^T \times A$,
- die Spaltenvektoren von U sind die Eigenvektoren von $A \times A^T$,
- die Zeilenvektoren von V sind die Eigenvektoren von $A^T \times A$.

Dabei kann $A \times A^T$ als Feature-Feature-Ähnlichkeitsmatrix interpretiert werden und $A^T \times A$ als Dokument-Dokument-Ähnlichkeitsmatrix. U ist als *Feature-Thema-Ähnlichkeitsmatrix* und V als *Dokument-Thema-Ähnlichkeitsmatrix* zu interpretieren.

Abbildung von Dokumentvektoren in den Themenraum:

Dokumente d und Queries q, beide als m×1-(Spalten)Vektoren repräsentiert, werden durch Multiplikation mit U^T in den Themenraum abgebildet:

$d \mapsto U^T \times d =: d'$ und $q \mapsto U^T \times q =: q'$.

Die Gesamtheit aller Dokumente, die Spalten von A, wird wie folgt abgebildet: $A \mapsto U^T \times A =: A'$. Dies ist äquivalent zu $A \mapsto U^T \times A = U^T \times \left(U \times \Delta \times V^T\right) = \Delta \times V^T = A'$. Die j-te Spalte von ΔV^T entspricht also dem in den Themenraum transformierten j-ten Dokument.

Die Ähnlichkeit zwischen d_j und q bzw. die Relevanz von d_j für q wird dann z.B. durch das Skalarprodukt $d_j{}'^T \times q'$ = $((\Delta V^T)_{*j})^T \times$ q' bestimmt (wobei M_{*j} die j-te Spalte der Matrix M bezeichnet).

Indexierung von A und Ausführung von Queries q:

Als Index zu verwalten ist die Dokument-Thema-Ähnlichkeitsmatrix ΔV^T sowie - quasi als Domänenwissen - die Term-Thema-Ähnlichkeitsmatrix U. Zur Verwendung von ΔV^T als Index werden in der Literatur auch Variationen betrachtet, insbesondere auch die Variante, bei der einfach V^T als Index verwendet wird und Δ selbst gar nicht berücksichtigt wird. Der Einfachheit halber verwenden wir im folgenden diese simplifizierte Variante. U kann man weitgehend statisch berechnen, sobald man eine repräsentative, große Dokumentenkollektion hat. Ein neu eingefügtes Dokument d (d.h. ein m×1-Spaltenvektor) muß dann zur Indexierung in den Themen-Vektorraum abgebildet werden, und zwar durch die Transformation d' = U^T × D, und der resultierende r×1-Spaltenvektor d' wird zum Index hinzugefügt, d.h. V^T wird um eine Spalte erweitert (*Folding-in*).

Eine Query q, die als m×1-Spaltenvektor aufgefasst werden kann, wird dann zunächst in eine Query q' in den Themen-Vektorraum transformiert, und zwar durch q' = U^T × q.

Dann wird q' aufgrund des Index V^T evaluiert (mit der verwendeten Ähnlichkeitsfunktion des r-dimensionalen Themen-Vektorraums), und die dadurch erzeugte Rangliste von Dokumenten ist das Anfrageresultat. Im einfachsten Fall würde z.B. das Cosinus-Maß, das Skalarprodukt oder die Euklidische Distanz zwischen q' und den Spalten von V^T (also den Dokumenten) verwendet.

Beispiel:

m=5 (Bush, Schröder, Korea, Klose, Völler), n=7

$$A = \begin{pmatrix} 1&2&1&5&0&0&0 \\ 1&2&1&5&0&0&0 \\ 1&2&1&5&0&0&0 \\ 0&0&0&0&2&3&1 \\ 0&0&0&0&2&3&1 \end{pmatrix} = \begin{pmatrix} 0.58 & 0.00 \\ 0.58 & 0.00 \\ 0.58 & 0.00 \\ 0.00 & 0.71 \\ 0.00 & 0.71 \end{pmatrix} \times \begin{pmatrix} 9.64 & 0.00 \\ 0.00 & 5.29 \end{pmatrix} \times \begin{pmatrix} 0.18 & 0.36 & 0.18 & 0.90 & 0.00 & 0.00 & 0.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 0.53 & 0.80 & 0.27 \end{pmatrix}$$

U Δ V^T

Die Anfrage q = $(0\ 0\ 1\ 0\ 0)^T$ wird in
q‘ = U^T q = $(0.58\ \ 0.00)^T$ transformiert und gegen V^T evaluiert.
Ein neues Dokument d8 = $(1\ 1\ 0\ 0\ 0)^T$ wird in
d8‘ = U^T d8 = $(1.16\ \ 0.00)^T$ transformiert und an V^T angefügt.

SVD als Regressionsverfahren und für approximatives LSI

Wenn der Rang r von A nicht hinreichend klein ist, also die Anzahl der “Themen" zu groß ist, kann man die Dimensionalität des Themen-Vektorraums künstlich auf k < r beschränken, indem man nur die k größten Singulärwerte von A und die zugehörigen Eigenvektoren betrachtet. Dadurch berücksichtigt man die “ausgeprägtesten" Themen und “stärksten" Zusammenhänge zwischen Features und Themen, was durch den folgenden Satz formal untermauert wird.

Satz:
Sei A eine m×n-Matrix mit Rang r und sei $A_k = U_k \times D_k \times V_k^T$, wobei die k×k-Diagonalmatrix D_k die k größten Singulärwerte von A enthält und die m×k-Matrix U_k und die k×n-Matrix V_k^T aus den zugehörigen Eigenvektoren der Singulärwertdekomposition von A bestehen.
Unter allen m×n-Matrizen C mit einem Rang, der nicht größer als k ist, ist A_k diejenige Matrix, die den Fehler $\|A-C\|_F^2 = \sum_{i=1}^{m}\sum_{j=1}^{n}(A_{ij}-C_{ij})^2$ minimiert (die Frobenius-Norm).

Zur Illustration:
Die SVD kann als affine Transformation der Dokumentvektoren aufgefaßt werden (also als Kombination von Drehung und Streckung/Stauchung). Bei der Beschränkung auf die k größten Singulärwerte erfolgt quasi eine Projektion auf die k aussagekräftigsten Dimensionen (diejenigen mit der größten Varianz der Datenpunktkoordinaten).
SVD verallgemeinert die Methode der kleinsten Quadrate zur linearen Regression. Im einfachsten Fall betrachtet man m Punkte (x_1, y_1), ..., (x_m, y_m) im R^2 und versucht eine lineare Funktion f(x)=ax+b zu finden, die den quadratischen Fehler $\sum_{i=1..m}(y_i - f(x_i))^2 = \sum_{i=1..m}(y_i - (ax_i + b))^2$ minimiert. Hierbei sind die Werte für alle x_i und y_i gegeben, sozusagen als Messwerte, und die Koeffizienten a und b sind unbekannt. Durch partielle Differenzierung und Nullstellenbestimmung der 1. Ableitungen kann man a und b berechnen, wie es schon Gauß vor mehr als 150 Jahren getan hat. In Matrixschreibweise ist dies mit A=(a b), $X=\begin{pmatrix} x_1 & \dots & x_m \\ 1 & \dots & 1 \end{pmatrix}$ und $Y=(y_1 \dots y_m)$ äquivalent dazu, $\|AX-Y\|_F^2$ zu minieren. Dies zeigt die enge Verwandtschaft zur SVD, die ihrerseits die lineare Regression auf m Punkte des R^n und die Bestimmung einer k-dimensionalen Hyperebene mit kleinestem quadratischem Fehler verallgemeinert.

Bei der Indexierung von Dokumenten und der Ausführung von Queries im - nunmehr k-dimensionalen Themenraum - spielen dann einfach U_k und V_k die Rolle von U und V.

Beispiel:
m=6 terms t1: bak(e,ing) t2: recipe(s) t3: bread
t4: cake t5: pastr(y,ies) t6: pie
n=5 documents
d1: How to bake bread without recipes
d2: The classic art of Viennese Pastry
d3: Numerical recipes: the art of scientific computing
d4: Breads, pastries, pies and cakes: quantity baking recipes
d5: Pastry: a book of best French recipes

$$A = \begin{pmatrix} 0.5774 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.5774 & 0.0000 & 1.0000 & 0.4082 & 0.7071 \\ 0.5774 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \\ 0.0000 & 1.0000 & 0.0000 & 0.4082 & 0.7071 \\ 0.0000 & 0.0000 & 0.0000 & 0.4082 & 0.0000 \end{pmatrix}$$

$$A = \begin{pmatrix} 0.2670 & -0.2567 & 0.5308 & -0.2847 \\ 0.7479 & -0.3981 & -0.5249 & 0.0816 \\ 0.2670 & -0.2567 & 0.5308 & -0.2847 \\ 0.1182 & -0.0127 & 0.2774 & 0.6394 \\ 0.5198 & 0.8423 & 0.0838 & -0.1158 \\ 0.1182 & -0.0127 & 0.2774 & 0.6394 \end{pmatrix} \quad \mathrm{U}$$

$$\times \begin{pmatrix} 1.6950 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 1.1158 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.8403 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.4195 \end{pmatrix} \quad \Delta$$

$$\times \begin{pmatrix} 0.4366 & 0.3067 & 0.4412 & 0.4909 & 0.5288 \\ -0.4717 & 0.7549 & -0.3568 & -0.0346 & 0.2815 \\ 0.3688 & 0.0998 & -0.6247 & 0.5711 & -0.3712 \\ -0.6715 & -0.2760 & 0.1945 & 0.6571 & -0.0577 \end{pmatrix} \quad \mathrm{V}^{\mathrm{T}}$$

$$A_3 = \begin{pmatrix} 0.4971 & -0.0330 & 0.0232 & 0.4867 & -0.0069 \\ 0.6003 & 0.0094 & 0.9933 & 0.3858 & 0.7091 \\ 0.4971 & -0.0330 & 0.0232 & 0.4867 & -0.0069 \\ 0.1801 & 0.0740 & -0.0522 & 0.2320 & 0.0155 \\ -0.0326 & 0.9866 & 0.0094 & 0.4402 & 0.7043 \\ 0.1801 & 0.0740 & -0.0522 & 0.2320 & 0.0155 \end{pmatrix} = U_3 \times \Delta_3 \times V_3^T$$

Anfrage q: baking bread → $q = (1\ 0\ 1\ 0\ 0\ 0)^T$
Transformation in den Themenraum mit k=3 → $q' = U_k^T \times q \approx (0.5340\ \ -0.5134\ \ 1.0616)^T$
Skalarprodukt-Ähnlichkeit im Themenraum mit k=3:
sim (q, d1) = $(V_k^T)_{*1}^T \times q' \approx 0.86$ sim (q, d2) = $(V_k^T)_{*2}^T \times q' \approx -0.12$
sim (q, d3) = $(V_k^T)_{*3}^T \times q' \approx -0.24$ usw.

Folding-in eines neuen Dokuments d6:
algorithmic recipes for the computation of pie → $d6 = (0\ \ 0.7071\ \ 0\ \ 0\ \ 0\ \ 0.7071)^T$
Transformation in den Themenraum mit k=3 → $d6' = U_k^T \times d6 \approx (0.5\ \ -0.28\ \ -0.15)^T$
d6‘ als neue Spalte an V_k^T anhängen

Zusammenfassende Bewertung von LSI

LSI ist ein elegantes, mathematisch wohlfundiertes Modell, das in Benchmarks auf homogenen Korpora wie z.B. Patentsammlungen, Wirtschaftsnachrichten oder Bibliotheken sehr gute Ergebnisse bzgl. Präzision und Ausbeute liefert. Dabei wird die approximative (Regressions-) Variante mit Werten von k in der Größenordnung von 100 verwendet. Diese Variante liefert deutlich bessere Ergebnisse als die mit dem vollen Rang r, da sie gewissermaßen Rauschen unterdrückt und Korrelationen kompakter zu Themen zusammenfasst.

LSI kann sogar für multilinguales IR verwendet werden, wenn man eine hinreichend große Startmenge von Dokumenten hat, die in mehreren Sprachen verfügbar sind. Dazu werden die mehrsprachigen Varianten desselben Dokument jeweils zu einem einzigen virtuellen Dokument konkateniert, und die SVD wird durch diese Startmenge berechnet. Dadurch kommen die starken Korrelationen zwischen den Wörter desselben Begriffs in mehreren Sprachen implizit im selben Thema zum Ausdruck. Weitere Dokumente können dann auch nur in einer Teilmenge der unterstützten Sprachen

(z.B. einer einzigen Sprache) verfügbar sein und werden per Folding-in in den Index eingefügt. Anfragen in einer beliebigen der unterstützten Sprachen werden auf den latenten Themenraum abgebildet und können somit auch Treffer in anderen Sprachen finden.

Für Web-Retrieval hat sich LSI nicht bewährt, da man es dort mit einem hochgradig heterogenen, terminologisch extrem streuenden Korpus zu tun hat. Hier ist es sehr schwer, einen brauchbaren Wert für die Zahl k der berücksichtigten Singulärwerte zu finden.

Ergänzende Literatur:

A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, S. Raghavan: Searching the Web, ACM Transactions on Internet Technology, Vol.1 No.1, 2001

Text REtrieval Conference, http://trec.nist.gov/

S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine, WWW Conference, 1998

A.Heydon, M. Najork: Mercator: A Scalable, Extensible Web Crawler, WWW Conference, 1999

Ronald Fagin, Amnon Lotem, Moni Naor: Optimal Aggregation Algorithms for Middleware, Journal of Computer and System Sciences Vol. 66, 2003

M.W. Berry, S.T. Dumais, G.W. O'Brien: Using Linear Algebra for Intelligent Information Retrieval, SIAM Review Vol.37 No.4, 1995

W.H. Press: Numerical Recipes in C, Cambridge University Press, 1993, available online at http://www.nr.com/

G.H. Golub, C.F. Van Loan: Matrix Computations, John Hopkins University Press, 1996

Christos H. Papadimitriou, Prabhakar Raghavan, Hisao Tamaki, Santosh Vempala: Latent Semantic Indexing: A Probabilistic Analysis, PODS Conference, 1998

Yossi Azar, Amos Fiat, Anna R. Karlin, Frank McSherry, Jared Saia: Spectral Analysis of Data, ACM Symposium on Theory of Computing (STOC), 2001

Kapitel 3: Automatische Klassifikation von Dokumenten

3.1 Einfache Klassifikatoren

In bestimmten Anwendungen möchte man Dokumente automatisch klassifizieren, also aufgrund ihrer Featurevektoren bestimmten Themen (Klassen, Kategorien) zuordnen. Wenn man von einigen Trainingsdokumenten d1, ..., dn die Klassenzuordnung $c(di) \in \{C1, ..., Ck\}$ a priori kennt, weil diese Dokumente intellektuell klassifiziert worden sind, kann man weitere Dokumente mit a priori unbekannter Klasse durch statistische Ähnlichkeit mit den Trainingsdaten der verschiedenen Klassen mit einer bestimmten Wahrscheinlichkeit einer Klasse zuordnen. Beispielsweise kann man Dokumente, in denen die Wörter Theorem und Homomorphismus mit hohem (tf*idf-) Gewicht auftreten, mit hoher Wahrscheinlichkeit der Klasse Mathematik zuordnen, während bei Dokumenten mit hoch gewichteten Termen Tor, Elfmeter und Schiedsrichter vieles für die Klasse Sport spricht. Verfahren dieser Art gehören zur Familie des *Supervised Learning*, also der *Lernverfahren mit Trainingsdaten*. Wenn man keinerlei Trainingsdaten hat, kann man Verfahren des *Unsupervised Learning* anwenden, insbesondere die statistischen Verfahren der *Cluster-Analyse*. Im folgenden wird nur Klassifikation mit Trainingsdaten betrachtet.

Gegeben sind n Trainingsdokumente d1, ..., $dn \in [0,1]^m$ über einem m-dimensionalen Featureraum mit bekannten Klassen $c(di) \in \{C1, ..., Ck\}$ aus einer Menge von k verschiedenen Themen. Ein Klassifikator ist eine Funktion $c: [0,1]^m \rightarrow \{C1, ..., Ck\}$.

Automatische Klassifikation ist in IRS-Anwendungen auf vielfältige Weise nützlich:
- *Filtern:* teste eintreffende Dokumente (z.B. Mail, News), ob sie in eine interessante Klasse fallen
- *Übersicht:* organisiere Query-/Crawler-Resultate, Verzeichnisse, Feeds, etc.
- *Query-Expansion:* ordne Query einer Klasse zu und ergänze dementsprechende Suchterme
- *Relevanz-Feedback:* klassifiziere Treffer und lasse Benutzer relevante Klassen identifizieren, um bessere Query zu generieren
- *Query-Effizienz:* beschränke (Index-)Suche auf relevante Klasse(n)

Gütemaße für Klassifikatoren

Zur Bewertung der Güte eines Klassifikators für eine Klasse C stellt man folgende Kontingenztafel auf:

	#Dokumente d mit c(d)=C	#Dokumente d mit c(d)≠C
#Dokumente $\in$ C	a	c
#Dokumente $\notin$ C	b	d

Dann sind die folgenden Gütemaße wie folgt definiert:

Präzision (Precision) = a / (a+b)
Ausbeute (Ausbeute) = a / (a+c)
Genauigkeit (Accuracy) = (a+d) / (a+b+c+d)

Fehler (Error) = 1 – Genauigkeit

Für einen Klassifikator mit mehr als 2 Klassen, berechnet Durchschnittswerte über alle Klassen. Dabei unterscheidet man Makrodurchschnitte (Macro Averages), bei denen direkt die Werte des jeweiligen Gütemaßes gemittelt werden (z.B. die Präzisionswerte), und Mikrodurchschnitte (Micro Averages), bei denen für das jeweilige Gütemaß die Zähler und Nenner separat über alle Klassen aufsummiert werden.

Zur experimentellen Bestimmung der Güte eines Klassifikators werden Benchmark-Daten mit bekannten Klassen verwendet. Man unterteilt die Daten in p Partitionen, trainiert auf einer Partition und testet die Klassifikatorgüte auf den anderen p-1 Partitionen. Diese Schema wird durch Wahl der Trainingspartition systematisch variiert, und der Mittelwert für alle p Möglichkeiten ist die experimentelle Güte. Dieses Prinzip nennt man Kreuzvalidierung (Cross Validation). Ein Sonderfall ist die Leave-one-out-Validierung, bei der man 2 unterschiedliche große Partitionen wählt, eine Trainingspartition, die alle Dokumente außer einem enthält, und eine Testpartition, die genau ein Dokument enthält.

kNN-Klassifikator

Der einfachste Klassifikator ist das sog. k-Nearest-Neighbor-Verfahren, kurz kNN. Es bestimmt zu einem zu klassifizierenden Dokument zunächst die k nächsten Nachbarn unter den Trainingsdaten gemäß einer Ähnlichkeitsfunktion über dem zugrundeliegenden Featureraum, z.B. der Cosinus-Ähnlichkeit. Dann ermittelt das Verfahren die Klassen dieser k nächsten Nachbarn und ordnet schließlich das neue Dokument der am häufigsten auftretenden Klasse zu. Beim letzten Schritt können die Klassenhäufigkeiten mit den Abständen der entsprechenden Trainingsdokumente zu dem neuen Dokument gewichtet werden.

Ordne d derjenigen Klasse Cj zu für die

$$f(\vec{d},C_j) = \sum_{\vec{v}\in kNN(\vec{d})} sim(\vec{d},\vec{v}) * \begin{cases} 1 & falls\ \vec{v}\in C_j \\ 0 & sonst \end{cases}$$

maximal ist.
Falls man nur binär klassifiziert, also nur testen will, ob ein Dokument zu einem gegebenen Thema passt oder nicht, ordnet man d zu, wenn $f(\vec{d},C_j)$ einen bestimmten Schwellwert δ überschreitet (z.B. δ=0.5).

Klassifikator nach Rocchio

Schritt 1:
Repräsentiere die Trainingsdokumente einer Klasse Cj
- mit tf*idf-Vektorkomponenten – durch den **Prototypvektor**:

$$\vec{c}_j := \alpha \frac{1}{|C_j|} \sum_{\vec{d}\in C_j} \frac{\vec{d}}{\|\vec{d}\|} - \beta \frac{1}{|D-C_j|} \sum_{\vec{d}\in D-C_j} \frac{\vec{d}}{\|\vec{d}\|}$$

mit geeigneten Koeffizienten α und β (z.B. α=16, β=4).

Schritt 2:
Ordne ein neues Dokument d derjenigen Klasse C_j zu, für
die Cosinus-Ähnlichkeit $\cos(d,c_j)$ maximal ist.

3.2 Grundlagen aus der Wahrscheinlichkeitsrechnung

Ein **Wahrscheinlichkeitsraum** ist ein Tripel (Ω, E, P) mit
- einer Menge W elementarer Ereignisse,
- einer Familie E von Teilmengen von Ω mit Ω∈E, die unter
  ∪, ∩ und - mit abzählbar vielen Operanden abgeschlossen ist
  (bei endlichem Ω ist in der Regel $E=2^{\Omega}$), und
- einem Wahrscheinlichkeitsmaß P: E → [0,1] mit P[Ω]=1 und $P[\cup_i Ai] = \Sigma_i P[Ai]$
  für abzählbar viele, paarweise disjunkte Ai

Eigenschaften von P:
P[A] + P[¬A] = 1
P[∅] = 0
P[A ∪ B] = P[A] + P[B] – P[A ∩ B]
P[Ω] = 1

Eine **Zufallsvariable** X über einem Wahrscheinlichkeitsraum (W, E, P) ist eine Funktion X: Ω → M mit M ⊆ R, so daß {e | X(e) ≤ x} ∈E für alle x ∈M.

F_X: M → [0,1] mit $F_X(x) = P[X \le x]$ heißt **Verteilungsfunktion** von X;
bei abzählbarer Menge M heißt f_X: M → [0,1] mit $f_X(x) = P[X = x]$ **Dichtefunktion** von X,
ansonsten ist $f_X(x)$ durch $F'_X(x)$ gegeben.

Einige wichtige Wahrscheinlichkeitsverteilungen sind:

- Diskrete Gleichverteilung über M={x | x∈N ∧ a ≤ x ≤ b} ⊆ N mit der Dichte
  $$P[X = k] = f_X(k) = \frac{1}{b-a+1} \quad \text{für } a \le k \le b,\ 0 \text{ sonst}$$
- Diskrete Bernoulli-Verteilung über M={0,1} mit der Dichte $f_X(k) = \begin{cases} p & \text{für } k = 1 \\ 1-p & \text{für } k = 0 \end{cases}$
- Diskrete Binomialverteilung über M={x | x∈N ∧ 0 ≤ x ≤ n} ⊆ N mit der Dichte
  $$P[X = k] = f_X(k) = \binom{n}{k} p^k (1-p)^{n-k}$$
- Diskrete Poisson-Verteilung über den natürlichen Zahlen mit der Dichte
  $$P[X = k] = f_X(k) = e^{-\lambda} \frac{\lambda^k}{k!}$$
- Diskrete geometrische Verteilung über den natürlichen Zahlen mit der Dichte
  $$P[X = k] = f_X(k) = (1-p)^k\ p$$
- Kontinuierliche Gleichverteilung über M=[a,b] ⊆ R für a,b∈R mit der Dichte
- $$f_X(x) = \frac{1}{b-a} \quad \text{für } a \le x \le b,\ 0 \text{ sonst}$$
- Kontinuierliche Exponentialverteilung über den nichtnegativen reellen Zahlen mit der Dichte
  $$f_X(x) = \lambda e^{-\lambda x} \quad \text{für } x \ge 0,\ 0 \text{ sonst}$$

- Kontinuierliche Pareto-Verteilung über M=={x | x∈R ∧ x>b} mit der Dichte

  $f_X(x) = \frac{a}{b}\left(\frac{b}{x}\right)^{a+1}$ *für* $x > b$, 0 *sonst*

- Kontinuierliche Normalverteilung über den reellen Zahlen mit der Dichte

  $f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Dabei sind a, b, p, λ, μ, σ (nichtnegative) Parameter.

Erwartungswert und Momente:
Für eine diskrete bzw. kontinuierliche Zufallsvariable X mit Dichte f(x) ist der Erwartungswert E[X] definiert durch $\sum_{x\in M} x \cdot f(x)$ bzw. $\int_{x\in M} x \cdot f(x)\,dx$. Das i-te Moment ist $E[X^i] = \sum_{x\in M} x^i \cdot f(x)$ bzw. $\int_{x\in M} x^i \cdot f(x)\,dx$. Die Varianz Var[X] ist $E[(X-E[X])^2] = E[X^2] - E[X]^2$.

Zwei Ereignisse A, B eines W.raums heißen **unabhängig**, wenn gilt $P[A \cap B] = P[A]\,P[B]$.
Die **bedingte Wahrscheinlichkeit P[A | B]** von A unter der Bedingung (Hypothese) B ist definiert als: $P[A \mid B] = \frac{P[A \cap B]}{P[B]}$

Satz von der totalen Wahrscheinlichkeit:

Für eine Partitionierung von W in Ereignisse B1, ..., Bn gilt: $P[A] = \sum_{i=1}^{n} P[A \mid Bi]\,P[Bi]$

Satz von Bayes:

Für Ereignisse A, B eines W.raums gilt: $P[A \mid B] = \frac{P[B \mid A] \cdot P[A]}{P[B]}$

Bedingte Wahrscheinlichkeiten der Form P[A|B] werden oft als A-Posteriori-Wahrscheinlichkeiten (nach Eintreten von B) bezeichnet; die unbedingte Wahrscheinlichkeit P[A] heißt dann A-Priori-Wahrscheinlichkeit.

Beispiele:
Beim Würfeln mit 1 Würfel ist P[X=5 | X ist ungerade] = (1/6) / (1/2) = 1/3.
Beim Würfeln mit 2 Würfeln ist P[X und Y zeigen einen Pasch | X+Y ≥] = (2/36) / (4/36) = 1/2.
Auf einem binären Übertragungskanal treten Bitfehler mit folgenden Wahrscheinlichkeiten auf:
P[0 empfangen | 0 gesendet] = 0.8,
P[1 empfangen | 0 gesendet] = 0.2,
P[0 empfangen | 1 gesendet] = 0.3,
P[1 empfangen | 1 gesendet] = 0.7,
und die Häufigkeiten von 0 und 1 seien wie folgt gegeben:
P[0 gesendet] = 0.4, P[1 gesendet] = 0.6.
Dann ist die Wahrscheinlichkeit, dass eine empfangene 1 auch wirklich eine 1 ist:
P[1 gesendet | 1 empfangen] =

$$\frac{P[\text{1 empfangen} \mid \text{1 gesendet}] \cdot P[\text{1 gesendet}]}{P[\text{1 empfangen}]} = \frac{P[\text{1 empfangen} \mid \text{1 gesendet}] \cdot P[\text{1 gesendet}]}{\sum_{x\in\{0,1\}} P[\text{1 empfangen} \mid x \text{ gesendet}] \cdot P[x \text{ gesendet}]} = 0.42/0.5 = 0.84.$$

3.3 Naive-Bayes-Klassifikator

Bei diesem Verfahren schätzt man die bedingte Wahrscheinlichkeit, dass ein Dokument zur Klasse Cj gehört unter der Vorbedingung, dass es einen bestimmten Featurevektor hat. Durch den Satz von Bayes lässt sich dies zurückführen auf die Wahrscheinlichkeit, dass das Dokument diesen Featurevektor hat, wenn es zur Klasse Cj gehört. Diese Wahrscheinlichkeit lässt sich anhand der Trainingsdaten schätzen.

Einfache Variante: binäre Features

Bei dieser Variante wird nur das Vorkommen oder Nichtvorkommen eines Terms im Dokument berücksichtigt; die Häufigkeit bleibt unberücksichtigt. Der Featurevektor des Dokuments di ist also ein binärer Vektor (der Dimension |F|=m), den wir zur besseren Abgrenzung mit Xi bezeichnen.

Schätze $P[\ d \in c_k \,|\, d\ \ hat\ \ \vec{X}\ ] = \dfrac{P[\ d\ \ hat\ \ \vec{X} \,|\, d \in c_k\ ]\, P[\ d \in c_k\ ]}{P[\ d\ \ hat\ \ \vec{X}\ ]}$

$\sim P[\ X \,|\, d \in c_k\ ]\, P[\ d \in c_k\ ]$

$= \Pi_{i=1}^{m}\, P[\ X_i \,|\, d \in c_k\ ]\, P[\ d \in c_k\ ]$ unter der Annahme, dass die Features paarweise unabhängig sind

bzw. der Linked-Independence-Annahme

$$\frac{P[\ X \,|\, d \in c_k\ ]}{P[\ X \,|\, d \notin c_k\ ]} = \Pi_i \frac{P[\ X_i \,|\, d \in c_k\ ]}{P[\ X_i \,|\, d \notin c_k\ ]}$$

$= \Pi_{i=1}^{m}\, p_{ik}^{\ Xi}\, (1 - p_{ik})^{1-Xi}\, p_k$ mit empirisch - anhand der Trainingsdaten - zu schätzenden

$p_{ik}=P[X_i=1|c_k]$, $p_k=P[c_k]$

Die Unabhängigkeitsannahme für Features ist eigentlich realitätsfern; sie erklärt das Attribut „naiv“ im Namen des Verfahrens. Da es bei der Klassifikation aber nur auf eine relative Schätzung der Zugehörigkeitswahrscheinlichkeiten zu den verschiedenen Klassen ankommt, funktioniert das Verfahren trotz dieser groben Annahme erstaunlich gut.

Bessere Variante: Bag-of-Words-Modell

Hier werden die Häufigkeiten der Terme im Dokument berücksichtigt, nicht jedoch deren Positionen zueinander (daher Bag-of-Words). Man postuliert ein generierendes Modell, nach dem Dokumente erzeugt werden, und schätzt die Parameter dieses Modells anhand der Trainingsdaten. Eine simple Variante nimmt dazu an, dass man für jeden Term separat „würfelt“ – mit einem zweiseitigen Würfel bzw. einer Münze, ob er auf Wortposition 1, 2, usw. erscheint; dies führt auf eine Binomialverteilung für die Termhäufigkeit. Eine präzisere Variante berücksichtigt die Nebenbedingung, dass die Summe aller Termhäufigkeit gleich der Dokumentlänge sein muss (nach Elimination von Stoppwörtern). Man „würfelt“ dabei für jede Wortposition – quasi mit einem m-seitigen Würfel -, welcher Term dort erscheint. Dies führt auf eine sog. Multinomialverteilung. Die Wahrscheinlichkeit der verschiedenen Würfelseiten sind die Parameter dieser Verteilung und werden anhand der Trainingsdaten geschätzt.

Schätze $P[\, d \in c_k \,|\, d \text{ hat } \vec{f}\,] \sim P[\, \vec{f} \,|\, d \in c_k\,]\, P[\, d \in c_k\,]$ mit Termhäufigkeitsvektor f

$= \Pi_{i=1}^{m} P[\, f_i \,|\, d \in c_k\,]\, P[\, d \in c_k\,]$ bei Featureunabhängigkeit

$= \Pi_{i=1}^{m} \binom{length(d)}{f_i} p_{ik}^{\,f_i} (1 - p_{ik})^{length(d) - f_i} \;\; p_k$ mit Binomialverteilung

bzw. präziser:

$= \binom{length(d)}{f_1\, f_2 \ldots f_m} p_{1k}^{\,f_1}\, p_{2k}^{\,f_2} \ldots p_{mk}^{\,f_m} \;\; p_k$ mit Multinomialverteilung und der Restriktion $\sum_{i=1}^{m} f_i = length(d)$

mit den Multinomialkoeffizienten $\binom{n}{k_1\, k_2 \ldots k_m} := \frac{n!}{k_1!\, k_2! \ldots k_m!}$

Beispiel für Naive-Bayes-Klassifikation mit Bag-of-Words-Modell

3 Klassen: c1 – Algebra, c2 – Analysis, c3 – Stochastik
8 Terme, 6 Trainingsdokumente d1, ..., d6: je 2 in jeder Klasse

$\Rightarrow$ p1=2/6, p2=2/6, p3=2/6

	Gruppe	Homomorphismus	Vektor	Integral	Limes	Varianz	Wahrscheinlichkeit	Würfel
	f1	f2	f3	f4	f5	f6	f7	f8
d1:	3	2	0	0	0	0	0	1
d2:	1	2	3	0	0	0	0	0
d3:	0	0	0	3	3	0	0	0
d4:	0	0	1	2	2	0	1	0
d5:	0	0	0	1	1	2	2	0
d6:	1	0	1	0	0	0	2	2

	Algebra	Analysis	Stochastik
	k=1	k=2	k=3
p1k	4/12	0	1/12
p2k	4/12	0	0
p3k	3/12	1/12	1/12
p4k	0	5/12	1/12
p5k	0	5/12	1/12
p6k	0	0	2/12
p7k	0	1/12	4/12
p8k	1/12	0	2/12

Klassifikation von d7: (0 0 1 2 0 0 3 0)

$P[\, \vec{f} \,|\, d \in c_k\,]\, P[\, d \in c_k\,] = \binom{length(d)}{f_1\, f_2 \ldots f_m} p_{1k}^{\,f_1}\, p_{2k}^{\,f_2} \ldots p_{mk}^{\,f_m} \;\; p_k$

für k=1 (Algebra): $= \binom{6}{1\;2\;3} \left(\frac{3}{12}\right)^1 0^2\, 0^3 \;\; \frac{2}{6} = 0$

für k=2 (Analysis): $= \binom{6}{1\;2\;3} \left(\frac{1}{12}\right)^1 \left(\frac{5}{12}\right)^2 \left(\frac{1}{12}\right)^3 \;\; \frac{2}{6} = 20 * \frac{25}{12^6}$

für k=3 (Stochastik): $= \binom{6}{1\;2\;3} \left(\frac{1}{12}\right)^1 \left(\frac{1}{12}\right)^2 \left(\frac{4}{12}\right)^3 \;\; \frac{2}{6} = 20 * \frac{64}{12^6}$

Resultat: Ordne d7 der Klasse C3 (Stochastik) zu

Verbesserte Parameterschätzung mit Glättung (Smoothing)

Aufgrund der Trainingsdaten werden manche der pik-Parameter mit dem Wert 0 geschätzt, so dass bestimmte Klassenzuordnungen gar nicht mehr in Frage kommen. Dies schneidet den Klassifikator zu stark auf die – mit einer gewissen Willkür behafteten – Trainingsdaten zu; diesen Effekt nennt man Overfitting. Zur Abhilfe schätzt man alle Parameter mit einem von 0 verschiedenen Wert, indem man 0-Werte durch kleine Werte ersetzt und dafür alle anderen Werte geringfügig reduziert. Die einfachste und bekannteste Technik für solche Parameterglättungen ist das Laplace-Smoothing (das sich mathematisch als Korrektur eines sog. Maximum-Likelihood-Schätzers mit einer A-Priori-Gleichverteilung für den zu schätzenden Parameter ergibt). Dabei setzt man für für ein elementares Ereignis, das unter n Versuchen mit m möglichen Ausgängen j-mal beobachtet wurde, den Schätzwert für die Ereigniswahrscheinlichkeit p auf p = (j+1) / (n+m).

3.4 Feature-Selektion

Bei der Klassifikation von Textdokumenten kann die Dimensionalität des Featureraums sehr groß sein, und man möchte aus Effizienzgründen lieber mit einem Raum niedrigerer Dimensionalität arbeiten. Außerdem möchte man das inhärente Rauschen im Featureraum unterdrücken, also nur die wichtigsten, für die jeweiligen Klassen charakteristischen, Terme berücksichtigen. Dies kann man mit Hilfe informationstheoretischer Maße realisieren, z.B. der **relativen Entropie** zwischen Termen und Klassen, die in der Literatur auch als **Mutual-Information-Maß (MI-Maß)** bezeichnet wird:

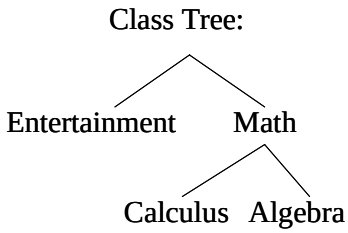
$$MI(X_i, c_j) = \sum_{X \in \{X_i, \bar{X}_i\}} \sum_{C \in \{c_j, \bar{c}_j\}} P[X \wedge C] \log \frac{P[X \wedge C]}{P[X]\,P[C]}$$ mit binären Variablen X_i=1, wenn der entsprechende Term in einem zufälligen Dokument vorkommt, und 0 sonst.

Dies ist ein Sonderfall der sog. **Kullback-Leibler-Distanz**, ein Maß für die Unterschiedlichkeit zweier Wahrscheinlichkeitsverteilungen, insbesondere zwischen einer zweidimensionalen Verteilung von Term und Klasse und einer Verteilung, bei der Term und Klasse unabhängig voneinander sind. Unabhängigkeit würde bedeuten, dass der Term in allen Klassen gleichwahrscheinlich ist, so dass er dann für die Klassifikation wenig geeignet wäre. Man wählt daher bei der Feature-Selektion gerade diejenigen m' << m Terme für eine Klasse aus, deren Kreuzentropie bzw. Kullback-Leibler-Divergenz am größten ist. Man beachte, dass diese Auswahl pro Klasse getroffen werden kann. Wenn man für alle Klassen denselben reduzierten Featureraum verwenden will, wählt man diejenigen Features mit den größten Werten der mit den Klassenhäufigkeiten gewichteten MI-Maße:

$$MI(X_i) = \sum_{j=1}^{k} P[c_j]\, MI(X_i, c_j)$$

Beispiel:

	film	hit	chart	theorem	limit	integral	group	vector
	f1	f2	f3	f4	f5	f6	f7	f8
d1:	1	1	0	0	0	0	0	0
d2:	0	1	1	0	0	0	1	0
d3:	1	0	1	0	0	0	0	0
d4:	0	1	1	0	0	0	0	0
d5:	0	0	0	1	1	1	0	0
d6:	0	0	0	1	0	1	0	0
d7:	0	0	0	0	1	0	0	0
d8:	0	0	0	1	0	1	0	0
d9:	0	0	0	0	0	0	1	1
d10:	0	0	0	1	0	0	1	1
d11:	0	0	0	1	0	1	0	1
d12:	0	0	1	1	1	0	1	0



training docs:
d1, d2, d3, d4
→ Entertainment
d5, d6, d7, d8
→ Calculus
d9, d10, d11, d12
→ Algebra

Die Güte des Terms „chart“ als Diskriminator zwischen den Klassen Entertainment und Math ist:
MI(chart, Entertainment) = 3/12 log (3/12 / (4*4/144)) + 1/12 log (1/12 / (4*8/144)) +
1/12 log (1/12 / (8*4/144)) + 7/12 log (7/12 / (8*8/144))

Ergänzende Literatur

- David D. Lewis: Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. ECML Conference, 1998
- Kamal Nigam, Andrew McCallum, Sebastian Thrun, Tom M. Mitchell: Text Classification from Labeled and Unlabeled Documents using EM. Machine Learning Vol. 39 No. 2/3, 2000
- Yiming Yang, Xin Liu: A Re-Examination of Text Categorization Methods. SIGIR Conf. 1999
- Yiming Yang, Jan O. Pedersen: A Comparative Study on Feature Selection in Text Categorization. ICML Conference, 1997
- Arnold O. Allen: Probability, Statistics, and Queueing Theory with Computer Science Applications, Academic Press, 1990
- M. Greiner, G. Tinhofer: Stochastik für Studienanfänger der Informatik, Hanser-Verlag, 1996
- Thomas M. Cover, Joy A. Thomas: Elements of Information Theory, Wiley&Sons, 1991

Kapitel 4: Linkanalyse für Autoritäts-Ranking

Zusätzlich zu ihrer inhaltlichen Relevanz können Dokumente auch nach ihrer Autorität, also nach Umfang, Klarheit und Signifikanz der in einem Dokument enthaltenen Information, bewertet werden, und entsprechende Autoritätsmaße können in das Ranking der Resultatsdokumente einer Query einfließen. Durch Kombination von Relevanz- und Autoritätsbewertung ist es möglich, die Präzision von Suchresultaten deutlich zu steigern, indem unter allen Treffern einer Anfrage diejenigen Dokumente mit hoher Autorität möglichst weit vorne in der Rangliste platziert werden.

Eine naheliegende Idee, Autorität von Web-Dokumenten zu quantifizieren, besteht darin, die eingehenden Hyperlinks einer Web-Seite als Zitate durch andere Web-Nutzer zu interpretieren. Einfach nur die Anzahl eingehender Links als Autoritätsmaß zu verwenden, greift jedoch zu kurz, da Links je nach Ursprung unterschiedliches Gewicht haben könnten und ein brauchbarer Ansatz resistent sein sollte gegen Link-Manipulationen, Zitier-Cliquen, u.ä.

4.1 Grundlagen aus der Stochastik

Ein **stochastischer Prozeß** ist eine Familie von Zufallsvariablen $\{X(t) \mid t \in T\}$.
T heißt Parameterraum, und der Definitionsbereich M der X(t) heißt Zustandsraum. T und M können diskret oder kontinuierlich sein.
Ein stochastischer Prozeß heißt **Markov-Prozeß**, wenn für beliebige $t_1, ..., t_{n+1}$ aus dem Parameterraum und für beliebige $x_1, ..., x_{n+1}$ aus dem Zustandsraum gilt:
Ein Markov-Prozeß mit diskretem Zustandsraum heißt **Markov-Kette**. O.B.d.A. werden die natürlichen Zahlen als Zustandsraum gewählt. Als Notation für Markov-Ketten mit diskretem Parameterraum schreiben wir: X_n statt $X(t_n)$ mit n = 0, 1, 2, ...

Die Markov-Kette Xn mit diskretem Parameterraum heißt
homogen, wenn die Übergangswahrscheinlichkeiten $p_{ij} := P[X_{n+1} = j \mid X_n = i]$ unabhängig von n sind
irreduzibel, wenn jeder Zustand von jedem Zustand mit positiver Wahrscheinlichkeit erreichbar ist:
$\sum_{n=1}^{\infty} P[X_n = j \mid X_0 = i] > 0$
aperiodisch, wenn alle Zustände i die Periode 1 haben, wobei die Periode von i der ggT aller Werte n ist, für die gilt: $P[X_n = i \wedge X_k \neq i \text{ für } k = 1,...,n-1 \mid X_0 = i] > 0$

Die Markov-Kette Xn mit diskretem Parameterraum heißt
positiv rekurrent, wenn für jeden Zustand i die Rückkehrwahrscheinlichkeit gleich 1 ist und die mittlere Rekurrenzzeit endlich: $\sum_{n=1}^{\infty} P[X_n = i \wedge X_k \neq i \text{ for } k = 1,...,n-1 \mid X_0 = i] = 1$ und

$\sum_{n=1}^{\infty} n\, P[X_n = i \wedge X_k \neq i \text{ for } k = 1,...,n-1 \mid X_0 = i] < \infty$. Sie heißt
ergodisch, wenn sie homogen, irreduzibel, aperiodisch und positiv rekurrent ist.

Für die **n-Schritt-Transitionswahrscheinlichkeiten** gilt:
$p_{ij}^{(n)} := P[X_n = j \mid X_0 = i] = \sum_k p_{ik}^{(n-1)} p_{kj}$ mit $p_{ij}^{(1)} := p_{ik}$ bzw. allgemeiner: $p_{ij}^{(n)} = \sum_k p_{ik}^{(n-t)} p_{kj}^{(t)}$ für $1 \leq t \leq n-1$

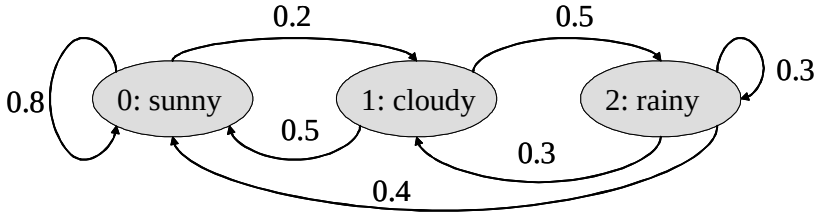
und in Matrixnotation: $P^{(n)} = P^n$.

Für die **Zustandswahrscheinlichkeiten nach n Schritten** gilt:
$\pi_j^{(n)} := P[X_n = j] = \sum_i \pi_i^{(0)} p_{ij}^{(n)}$ mit initialen Zustandswahrscheinlichkeiten $\pi_i^{(0)}$
und in Matrixnotation: $\Pi^{(n)} = \Pi^{(0)} P^{(n)}$ mit einem 1×n-Zeilenvektor Π.
Diese Gleichung nennt man die Chapman-Kolmogorov-Gleichung für Markov-Ketten.

Satz:
Jede homogene, irreduzible, aperiodische Markov-Kette mit endlich vielen Zuständen ist positiv rekurrent und ergodisch.
Für jede ergodische Markov-Kette existieren **stationäre Zustandswahrscheinlichkeiten** $\pi_j := \lim_{n\to\infty} \pi_j^{(n)}$**.** Diese sind unabhängig von $P^{(0)}$ und durch das folgende lineare Gleichungssystem bestimmt: $\pi_j = \sum_i \pi_i p_{ij}$ *für alle* j (Gleichgewichtsgleichungen) und $\sum_j \pi_j = 1$
bzw. in Matrix-Notation: $\Pi = \Pi P$ und $\Pi \vec{1} = 1$ mit einem n×1-Spaltenvektor $\vec{1}$.

Π ist also ein Eigenvektor der Matrix P zum dominanten (d.h. betragsgrößten) Eigenwert 1.
Man kann die Lösung für Π auf zwei Arten berechnen: entweder durch Lösen des Gleichungssystems oder approximativ durch Potenziteration (Power Iteration): $\Pi^{(0)} = (1/n \ldots 1/n)^T$ und $(\Pi^{(i)})^T = (\Pi^{(i-1)})^T P$.

Beispiel:



$\pi0 = 0.8\,\pi0 + 0.5\,\pi1 + 0.4\,\pi2$
$\pi1 = 0.2\,\pi0 + 0.3\,\pi2$
$\pi2 = 0.5\,\pi1 + 0.3\,\pi2$
$\pi0 + \pi1 + \pi2 = 1$

$\Rightarrow \pi0 = 330/474 \approx 0.696$
$\pi1 = 84/474 \approx 0.177$
$\pi2 = 10/79 \approx 0.126$

4.2 Autoritäts-Ranking nach der Methode von Page und Brin (Page-Rank)

Das Web (oder ein Intranet) wird bei dieser Methode, die in der Suchmaschine Google verwendet wird, als gerichteter Graph G = (V,E) gesehen mit Web-Seiten als Knotenmenge V, |V|=n, und Hyperlinks als Kantenmenge E. Für die folgenden Berechnungen wird angenommen, dass der Graph komplett a priori bekannt ist und seine Adjazenzmatrix A, eine n×n-Matrix mit Aij = 1 falls (i,j)∈E, 0 sonst, auf einem Server gespeichert ist. Google baut diesen Graph aufgrund der Ergebnisse seines Crawlers auf; diese sind bei Google mehr als 1 Milliarde Knoten.

Die Kernidee dieser Methode ist, dass die *Autorität (Authority-Score, Authority-Rank)* r(q) einer Web-Seite q proportional zur Summe der Autoritätsbewertungen der Vorgänger von q ist – vorausgesetzt, alle Vorgänger hätten dieselbe Anzahl ausgehender Links. Bei Vorgängern mit vielen ausgehenden Kanten würde man das Autoritätsmaß nur anteilsmäßig auf die Zieldokumente der Kanten umlegen. Diese Überlegung führt auf die folgende erste Arbeitsdefinition:

$$r(q) = k \sum_{(p,q)\in E} r(p) / outdegree(p)$$

mit einer Konstanten k und der Anzahl *outdegree(p)* von p ausgehender Kanten.

Es zeigt sich jedoch, dass diese Definition noch nicht tragfähig ist, da unklar ist, wie man mehreren Zusammenhangskomponenten oder gar isolierten Knoten umgehen sollte. Daher weist die Lösung von Page und Brin jeder Web-Seite unabhängig von ihren Vorgängern eine Mindestautorität ε/n zu und berechnet die (1-ε) gewichtete Restautorität nach dem o.a. Ansatz.

Definition:

Die Autorität der Web-Seite q im Web-Graphen G=(V,E) ist gegeben durch

$$r(q) = \varepsilon / n + (1-\varepsilon) \sum_{(p,q)\in E} r(p) / outdegree(p) .$$

Dabei ist ε ein Kalibrierungsparameter, für den lt. Page und Brin $0 < \varepsilon \leq 0.25$ gelten sollte.

Satz:

Mit einer modifizierten Matrix A' mit A'_{ij} = 1/outdegree(j) falls $(j,i)\in E$ und 0 sonst, gilt

$\vec{r} = \vec{\varepsilon} / n + (1-\varepsilon)A'\vec{r}$ und äquivalent dazu $\vec{r} = \left(\frac{\vec{\varepsilon}}{n}\vec{1}^T + (1-\varepsilon)A' \right)\vec{r}$.

Dabei ist $\vec{r}$ ein n×1-Spaltenvektor, und $\vec{\varepsilon}$ und $\vec{1}$ sind n×1-Spaltenvektoren, die komplett mit dem Wert ε bzw. 1 besetzt sind.

Man sieht somit, dass der Spaltenvektor $\vec{r}$ der Autoritätswerte Eigenvektor einer modifizierten Transitionsmatrix ist. Eine approximative Berechnung durch Potenziteration ist:

1) $\vec{r}^{(0)} = \vec{1} / n$
2) Wiederhole bis sich die größten Werte von $\vec{r}$ kaum noch ändern:
   $\vec{r}^{(i+1)} = \vec{\varepsilon} / n + (1-\varepsilon)A'\vec{r}^{(i)}$

Die besten Autoritäten sind dann diejenigen Komponenten von $\vec{r}$ mit den größten Werten.

Diese Methode funktioniert in der Praxis verblüffend gut. Google z.B. rechnet ca. 100 Iterationen und speichert dann die Web-Seiten-spezifischen Autoritätswerte in seinem Index. Bei Anfragen berechnet Google pro Trefferseite einen anfragespezifischen Relevanz-Score und kombiniert diesen in einer gewichteten Summe mit dem vorberechneten Autoritäts-Score. Die Resultatsliste ist der Präfix der nach dieser gewichteten Summe absteigend sortierten Liste. Die relativen Gewichte von Relevanz und Autorität sind per Trial-and-Error anhand typischer Google-Queries ermittelt.

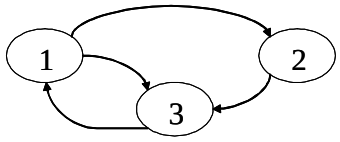
Auch die Methode in der Praxis sehr gut funktioniert, ist vor allem der Ad-hoc-Charakter der oben eingeführten Gewichtungen, insbesondere auch der etwas willkürlichen Einführung von ε, wissenschaftlich unbefriedigend. Eine alternative Herleitung derselben Lösung verwendet einen sog. Random Walk auf dem Web-Graphen und mathematische Resultate über Markov-Ketten.

Autoritätsbewertung mittels Random Walk

Das Verfahren von Page und Brin modelliert einen Random Walk über den Web-Graphen, bei dem man mit Wahrscheinlichkeit (1-ε) von der aktuell besuchten Web-Seite zu einem der Nachfolger wandert und mit Wahrscheinlichkeit ε einen „Random Jump“ zu irgendeiner Web-Seite macht. Im ersten Fall wird die als nächstes besuchte Seite gemäß einer Gleichverteilung unter den Nachfolgern der aktuellen Seite ausgewählt; im zweiten Fall wird eine Seite unter allen Web-Seiten gemäß einer Gleichverteilung ausgewählt.

Für den so definierten Random Walk kann man eine Markov-Kette angeben, die beweisbar ergodisch ist. Der Autoritäts-Score einer Seite nach dem Verfahren von Page und Brin ist die **stationäre Besuchswahrscheinlichkeit der Web-Seite** in dieser Markov-Kette.

Beispiel:



$\varepsilon = 0.2$ $\quad P = \begin{pmatrix} 0.0 & 0.5 & 0.5 \\ 0.1 & 0.0 & 0.9 \\ 0.9 & 0.1 & 0.0 \end{pmatrix}$

$$\Pi^{(0)} \approx \begin{pmatrix} 0.333 \\ 0.333 \\ 0.333 \end{pmatrix}^T \Rightarrow \Pi^{(1)} \approx \begin{pmatrix} 0.333 \\ 0.200 \\ 0.466 \end{pmatrix}^T \Rightarrow \Pi^{(2)} \approx \begin{pmatrix} 0.439 \\ 0.212 \\ 0.346 \end{pmatrix}^T \Rightarrow \Pi^{(3)} \approx \begin{pmatrix} 0.332 \\ 0.253 \\ 0.401 \end{pmatrix}^T$$

$$\Rightarrow \Pi^{(4)} \approx \begin{pmatrix} 0.385 \\ 0.176 \\ 0.527 \end{pmatrix}^T \Rightarrow \Pi^{(5)} \approx \begin{pmatrix} 0.491 \\ 0.244 \\ 0.350 \end{pmatrix}^T$$

$\pi1 = 0.1\ \pi2 + 0.9\ \pi3$
$\pi2 = 0.5\ \pi1 + 0.1\ \pi3$
$\pi3 = 0.5\ \pi1 + 0.9\ \pi2$
$\pi1 + \pi2 + \pi3 = 1$ $\quad \Rightarrow \pi1 \approx 0.3776,\ \pi2 \approx 0.2282,\ \pi3 \approx 0.3942$

4.3 Autoritäts-Ranking nach der HITS-Methode von Kleinberg

Bei dem HITS-Verfahren nach Kleinberg (Hyperlink-Induced Topic Search) analysiert man die Linkstruktur eines relativ kleinen Untergraphen, der aufgrund seiner thematischen Relevanz bestimmt wird. Man berechnet zunächst aufgrund einer klassischen Relevanzbewertung eine Menge von Wurzelseiten, z.B. die Top 100 einer Anfrage bei Google oder AltaVista, und fügt zu dieser alle Nachfolger und möglichst viele Vorgänger hinzu sowie alle Kanten zwischen den betrachteten Seiten; der Graph G=(V,E), |V|=n, der so enstandene Menge von – z.B. einigen Tausend – Seiten V bildet die Basis der Linkanalyse. Anders als beim Verfahren nach Page und Brin ist dieser Graph anfragespezifisch.

Die Arbeitshypothese des HITS-Verfahrens ist, dass es in einem solchen Graph außer guten *Autoritäten (Authorities)* auch gute *Referenzen (Hubs)* gibt:Linksammlungen, die Verweise auf die besten Autoritäten eines bestimmten Themas enthalten. Das Verfahren berechnet für jede Seite p im Graphen sowohl ein *Autoritätsgewicht (Authority-Score)* x_p als auch ein *Referenzgewicht (Hub-Score).* y_p. Zwischen Autoritäten und Referenzen gibt es eine wechselseitige Rekursion: eine Autorität ist umso besser, hat also höheres Referenzgewicht, je besser die Autoritäten sind, auf die sie verweist, und eine Autorität ist umso besser, hat also höheres Autoritätsgewicht, je besser die Referenzen sind, die auf sie verweisen. Wenn man postuliert, dass dieser Zusammenhang zwischen Autoritätsgewichten x_p und Referenzgewichten y_p linear ist, kommt man auf folgende Gleichungen:

$$x_q = \sum_{(p,q)\in E} y_p \quad \text{und} \quad y_p = \sum_{(p,q)\in E} x_q$$

bzw. in Matrix-Notation:

$\vec{x} = A^T \vec{y}$ und $\vec{y} = A\vec{x}$, wobei A die Adjazenzmatrix von G ist.

Setzt man die rechte Seite der y-Gleichung in die rechte Seite der x-Gleichung ein (und analog für y), so erhält man

$$\vec{x} := A^T \vec{y} := A^T A \vec{x} \quad \text{und} \quad \vec{y} := A\vec{x} := AA^T \vec{y},$$

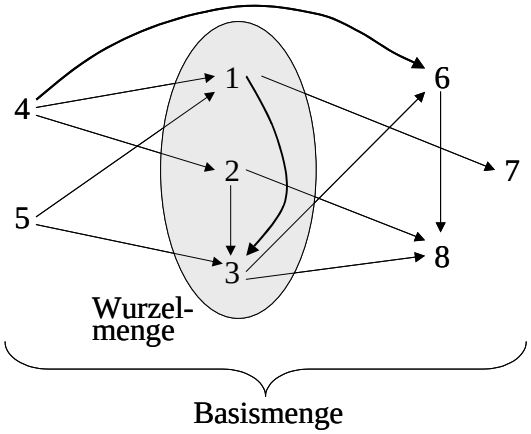
und wir erkennen, dass die Vektoren x und y Eigenvektoren der Matrizen A^TA bzw. AA^T sind.

Die Matrix $M^{(auth)} := A^TA$ kann als Cocitation-Matrix interpretiert werden: $M^{(auth)}_{ij}$ ist die Anzahl der Web-Seiten, die auf i und auf j verweisen. Die Matrix $M^{(hub)} := AA^T$ kann als Bibliographic-Coupling-Matrix interpretiert werden: $M^{(hub)}_{ij}$ ist die Anzahl der Web-Seiten, auf die sowohl i als auch j verweisen

Die wechselseitige Rekursion kann iterativ berechnet werden, indem man sowohl den x- als auch den y-Vektor zunächst mit dem Wert 1/n in allen Komponenten initialisiert und dann abwechselnd x und y neu berechnet, bis sich die größten Komponenten nur noch geringfügig ändern. In jedem Iterationsschritt werden die erhaltenen x- und y-Vektoren auf die Länge 1 normalisiert. Dieses Iterationsverfahren konvergiert (unter bestimmten Bedingungen, die in der Regel gegeben sind) gegen die Eigenvektoren von A^TA und AA^T, die zum betragsgrößten Eigenwert gehören.

Der HITS-Algorithmus sieht insgesamt also folgendermaßen aus:
1) Bestimme hinreichend viele (z.B. 50-200) „Wurzelseiten“
   per Relevanz-Ranking (z.B. mittels tf*idf-Ranking)
2) Füge alle Nachfolger von Wurzelseiten hinzu
3) Füge für jede Wurzelseite max. d Vorgänger hinzu (durch Zufallsauswahl unter allen Vorgängern)
4) Erzeuge den Graph G=(V,E), |V|=n, für die so erhaltene Basismenge
5) Initialisiere alle Komponenten von $x^{(0)}$ mit 1/n und die Komponenten von $y^{(0)}$ mit 1/n
6) Solange noch keine hinreichende Konvergenz erreicht ist
   (oder für eine feste Anzahl von Iterationen) wiederhole
   $x^{(i)} := A^T y^{(i-1)}$ und $y^{(i)} := A x^{(i-1)}$
7) Gib Seiten nach absteigend sortierten Authority-Scores aus
   (z.B. die 10 größten Komponenten von x)

Illustration der Schritte 1 bis 4:



Das HITS-Verfahren hat eine gewisse Anfälligkeit gegenüber Themendrifts: auch wenn die Wurzelmenge nur für das ursprünglich gegebene Thema (die Anfrage) relevant ist, könnte der Algorithmus auf sehr starke Autoritäten und Referenzen zu einem anderen Thema stoßen, so dass das Endresultat durch das andere Thema dominiert sein könnte. Um dies zu verhindern, kann man – in einer erweiteten Variante – in jedem Iterationsschritt thematische Relevanzwerte (z.B. tf*idf-basierte Ähnlichkeiten zur ursprünglichen Anfrage) als multiplikative Gewichte der Vorgänger- bzw. Nachfolger-Gewichte einbauen.

Das HITS-Verfahren kann auch benutzt werden, um zu einer gegebenen Web-Seite ähnliche Web-Seiten zu ermitteln; dabei bezieht sich Ähnlichkeit auf die Linkstruktur, z.B. wären zwei Seiten sehr ähnlich, wenn sie genau dieselben Vorgänger und Nachfolger hätten. Für diese Art der Ähnlichkeitsanalyse bestimmt man zu einer gegebenen Web-Seite zunächst alle Nachfolger, alle oder eine beschränkte Anzahl der Vorgänger sowie (eine beschränkte Zahl von) Vorgänger(n) der Nachfolger und Nachfolger(n) der Vorgänger. Auf dieser Basismenge führt man dann den HITS-Algorithmus aus, und die ermittelten Autoritätsgewichte liefern eine Rangliste ähnlicher Seiten.

4.4 Themenspezifisches Page-Rank-Verfahren

Alle Verfahren zur Autoritätsanalyse verursachen signifikanten Berechnungsaufwand, so dass sie bei großen Web-Suchmaschine nicht für jede Anfrage neu berechnet werden können. Bei Google etwa werden Page-Rank-Werte nur gelegentlich berechnet und im Index abgespeichert. Autorität ist also in diesem Fall Query- und damit Benutzer-unabhängig. Eine Methode, Autorität in einer für den jeweiligen Benutzer spezifischen Form auszunutzen, ist das themenspezifische Page-Rank-Verfahren. Dabei werden Anfragen durch Klassifikation auf eine vorgegebene Menge von Themen wie z.B. Sport, Politik, Elektronik, Internet, etc. abgebildet. Für jedes Thema gibt es eine intellektuell zusammengestellte Menge von spezifischen Autoritäten, und ein themenspezifischer Page-Rank-Vektor wird dadurch berechnet, dass man die Random Jumps so modifiziert, dass die Autoritäten des jeweiligen Themas mit höherer Wahrscheinlichkeit angesprungen werden.

Sei T_k die Menge der vorgegebenen Autoritäten für das k-te Thema. Die Page-Rank-Gleichung für Thema k lautet dann:

$$\vec{r}_k = \varepsilon \vec{p}_k + (1-\varepsilon) A' \vec{r}_k$$

mit einem Zielvektor $\vec{p}_k$ für Random Jumps, dessen i-te Komponente $1/|T_k|$ ist, wenn i in T_k liegt und 0 sonst (und dieser Ansatz ließe sich noch weiter verallgemeinern, um die relativen Autoritätswerte der Seiten in T_k zu berücksichtigen). A'_{ij} ist 1/outdegree(i), wenn es einen Link von Seite i zur Seite j gibt, und 0 sonst.

Für jedes Thema werden die themenspezifischen Page-Rank-Werte aller Web-Seiten vorausberechnet. Zum Zeitpunkt der Ausführung einer Query q werden folgende Schritte ausgeführt:

1) Berechne Wahrscheinlichkeiten bzw. normalisierte Konfidenzwerte w_k, dass q zur Klasse c_k gehört. (Dies kann man z.B. mit einem Naive-Bayes-Klassifikator tun.)
2) Setze den Autoritätswert einer für q interessanten, potentiellen Trefferseite d auf $\sum_k w_k \, r_k(d)$.

Alternativ könnte man auch Benutzer aufgrund von Profilen oder expliziter Registrierung einem oder mehreren Themen zuordnen und daraus Gewichte w_k ableiten.

Wenn man bei der **Personalisierung** des Suchmaschinenverhaltens noch weitergehen will, könnte man jeden Benutzer selbst einen persönlichen Random-Jump-Vektor $\vec{p}_k$ anlegen lassen. Auf den ersten Blick sieht das nach einem monströsen Verwaltungs-Overhead aus, der für eine Websuchmaschine mit Millionen von Benutzern nicht mehr leistbar ist. Wenn jedoch die Gesamtheit der Random-Jump-Ziele aller Benutzer eine nicht zu große Menge T bildet, kann man jeden der persönlichen Page-Rank-Vektoren als Linearkombination von elementaren Page-Rank-Vektoren berechnen. Dabei ist ein elementarer Page-Rank-Vektor der Vektor der Autoritätswerte aller Webseiten, der sich aus der Gleichung $\vec{r}_i = \varepsilon \vec{e}_i + (1-\varepsilon) A' \vec{r}_i$ ergibt mit dem Basisvektor $\vec{e}_i$, der als i-te Komponente den Wert 1 hat und in allen anderen Komponenten den Wert 0. Bei diesem Random Walk gibt es also nur ein einziges Random-Jump-Ziel, die Seite i. Hat man nun $\vec{r}_i$ für jede Seite i aus T berechnet, kann man für jeden Benutzer mit Random-Jump-Profil $\vec{p} = \sum_{i \in T} \alpha_i \vec{e}_i$ - mit Gewichten $\alpha_i \geq 0$ - den persönlichen Page-Rank-Vektor $\vec{r}$ als konvexe Linearkombination der $\vec{r}_i$ berechnen: $\vec{r} = \sum_{i \in T} \alpha_i \vec{r}_i$.

Ergänzende Literatur

- S. Brin, L. Page: The Anatomy of a Large-Scale Hypertextual Web Search Engine, WWW Conference 1998.
- J.M. Kleinberg: Authoritative Sources in a Hyperlinked Environment, Journal of the ACM Vol.46 No.5, 1999
- C. Ding, X. He, P. Husbands, H. Zha, H. Simon: PageRank, HITS, and a Unified Framework for Link Analysis, SIAM Int. Conf. on Data Mining, 2003.
- A. Borodin, J. S. Rosenthal, G. O. Roberts, P. Tsaparas: Finding Authorities and Hubs From Link Structures on the World Wide Web, WWW Conference 2001
- Taher Haveliwala: Topic-Sensitive PageRank: A Context-Sensitive Ranking Algorithm for Web Search, IEEE Transactions on Knowledge and Data Engineering Vol. 15 No. 4, 2003

...

Kapitel 5: Relationenmodell und algebraorientierte Anfragesprachen

5.1 Grundbegriffe

Das Relationenmodell beruht auf Arbeiten von E.F. Codd und anderen um 1970, für die Codd 1981 mit dem Turing Award ausgezeichnet wurde.

Grundprinzip:
Alle Daten werden in Form von mathematischen Relationen (Teilmengen eines kartesischen Produkts von Wertemengen) repräsentiert.

Definition:
Gegeben sei eine Menge von Wertebereichen primitiver Datentypen {D1, ..., Dm}, die als "*Domains*" bezeichnet werden.
Eine *Relation* R ist ein Paar R = (s,v) mit

- einem *Schema* s = {A1, ..., An}, das aus einer Menge von *Attributen* (genauer: Attributnamen) besteht und für jedes Attribut Ai einen Domain dom(Ai) $\in$ {D1, ..., Dm} festlegt, und
- einer *Ausprägung* (auch *Wert* genannt) v $\subseteq$ dom(A1) $\times$ dom(A2) $\times$ ... $\times$ dom(An).

Schema und Ausprägung von R werden auch mit *sch(R)* und *val(R)* bezeichnet.

Bemerkungen:
- Die Elemente der Ausprägung einer Relation heißen *Tupel*. Für den Wert eines Tupels t bezüglich eines Attributs Ai schreibt man t.Ai (seltener auch t[Ai] oder Ai(t)).
- Die Tupel einer Relation bilden eine (ungeordnete) Menge. Es gibt also keine Reihenfolge unter den Tupeln, und je zwei Tupel müssen sich in mindestens einem Attribut unterscheiden.
- Die Attribute einer Relation bilden eine (ungeordnete) Menge; es gibt also keine Reihenfolge unter den Attributen.
- Als Domains von Attributen sind nur primitive Wertebereiche zugelassen, d.h. Wertebereiche elementarer Datentypen (im wesentlichen INTEGER, REAL, BOOLEAN plus Spezialtypen wie DATE, MONEY, etc.), die Wertebereiche von Bereichstypen und Aufzählungstypen sowie der Wertebereich der Zeichenketten (ARRAY OF CHAR). Diese Einschränkung wird auch als *1. Normalform* des Relationenmodells bezeichnet.
- Eine relationale Datenbank ist eine Menge von Relationen. Die Menge der betreffenden Relationenschemata wird als Datenbankschema bezeichnet.
  Häufige Schreibweise für Relationenschemata:
  R(A1, ..., An)
  Kunden (KNr, Name, Stadt, Saldo, Rabatt)
  Produkte (PNr, Bez, Gewicht, Preis, Lagerort, Vorrat)
  Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge, Summe, Status)

Informelle Sprechweise:

Relation	=	Tabelle
Schema	=	Tabellenkopf (Überschrift der Tabelle)
Attribut	=	Spalte der Tabelle (engl.: column, field; Feld)
Tupel	=	Zeile der Tabelle (engl.: row, record; Datensatz)

Beispieldatenbank:

Kunden

KNr	Name	Stadt	Saldo	Rabatt
1	Lauer	Merzig	- 1080.00	0.10
2	Schneider	Homburg	- 800.00	0.20
3	Kirsch	Homburg	0.00	0.10
4	Schulz	Merzig	0.00	0.10
5	Becker	Dillingen	0.00	0.05
6	Meier	Saarlouis	- 3800.00	0.05

Produkte

PNr	Bez	Gewicht	Preis	Lagerort	Vorrat
1	Papier	2.000	20.00	Homburg	10000
2	Platte	1.000	2500.00	Saarbrücken	400
3	Drucker	5.000	2000.00	Merzig	200
4	Bildschirm	5.000	3000.00	Merzig	80
5	Disketten	0.500	20.00	Homburg	5000
6	Maus	0.250	100.00	Homburg	200
7	Speicher	0.100	200.00	Saarbrücken	2000

Bestellungen

BestNr	Monat	Tag	KNr	PNr	Menge	Summe	Status
1	7	16	1	1	100	1800.00	bezahlt
2	7	21	1	1	100	1800.00	bezahlt
3	9	30	1	2	4	9000.00	bezahlt
4	9	30	1	3	1	1800.00	bezahlt
5	9	30	1	4	10	27000.00	bezahlt
6	10	15	1	5	50	900.00	bezahlt
7	10	28	1	6	2	180.00	geliefert
8	11	2	1	7	5	900.00	neu
9	10	26	2	1	100	1600.00	bezahlt
10	11	2	2	5	50	800.00	neu
11	9	28	3	5	50	900.00	bezahlt
12	10	28	3	7	10	1800.00	bezahlt
13	4	15	4	1	50	900.00	bezahlt
14	5	31	6	1	200	3800.00	bezahlt
15	6	30	6	7	10	1900.00	geliefert
16	7	31	6	1	100	1900.00	geliefert

Konsequenz der 1. Normalform

Studenten

Name	Fach	...	Systemkenntnisse
Meier	Informatik		{Oracle, mySQL, PHP}
Schmidt	Informatik		{Java, Oracle}
Kunz	Informatik		{Oracle}
Müller	Mathematik		∅

ist nicht zulässig!

Korrekte Repräsentation im Relationenmodell:
Studenten

Name	Fach	...	Systemkenntnis
Meier	Informatik		Oracle
Meier	Informatik		mySQL
Meier	Informatik		PHP
Schmidt	Informatik		Java
Schmidt	Informatik		Oracle
Kunz	Informatik		Oracle
Müller	Mathematik		---

oder besser:
Studenten

Name	Fachbereich	...
Meier	Informatik	
Schmidt	Informatik	
Kunz	Informatik	
Müller	Mathematik	

Kenntnisse

Name	Systemkenntnis
Meier	Oracle
Meier	mySQL
Meier	PHP
Schmidt	Java
Schmidt	Oracle
Kunz	Oracle

Inhärente Integritätsbedingungen des Relationenmodells

Definitionen:
Sei R eine Relation. Eine Attributmenge $K \subseteq sch(R)$ heißt *Schlüsselkandidat*, wenn zu jedem Zeitpunkt für je zwei Tupel $t1, t2 \in val(R)$ gelten muß:

$t1.K = t2.K \Rightarrow t1 = t2$

und wenn es keine echte Teilmenge von K gibt, die diese Eigenschaft hat.
(Dabei bedeutet $t1.K = t2.K$ präzise: $\forall A \in K: t1.A = t2.A$.)
Bemerkung: Jede Relation hat mindestens einen Schlüsselkandidaten.
Ein Attribut einer Relation R, das in mindestens einem Schlüsselkandidaten vorkommt, heißt *Schlüsselattribut*.
Der *Primärschlüssel* (engl.: primary key) einer Relation ist ein Schlüsselkandidat, der explizit ausgewählt wird.
Eine Attributmenge $F \subseteq sch(S)$ einer Relation S ist ein *Fremdschlüssel* (engl.: foreign key) in S, wenn es eine Relation R gibt, in der F Primärschlüssel ist.
Ein Attribut A eines Tupels t hat einen *Nullwert*, wenn der Wert t.A undefiniert oder unbekannt ist. (Gedanklich werden alle Domains um einen solchen Nullwert erweitert.)
Beispiele:

- Vorrat für Produkte, die nicht gelagert werden, sondern bei Bedarf extern gekauft werden.
- Note von Studenten, die ihre Prüfung noch nicht abgelegt haben.
- Informatikkenntnisse von Studenten zu Beginn des Studiums.

Schreibweise:
In einem Relationenschema wird der Primärschlüssel unterstrichen; weitere Schlüsselkandidaten werden oft gestrichelt unterstrichen.
Das Relationenmodell (genauer: eine Implementierung des Relationenmodells) garantiert die folgenden Integritätsbedingungen:

Primärschlüsselbedingung ("Entity Integrity"):

Für jede Relation muß ein Primärschlüssel festgelegt werden. Der Primärschlüssel eines Tupels darf nie den Nullwert annehmen (auf keinem einzigen der zum Primärschlüssel gehörigen Attribute).
Beispiel:

Kunden (<u>KNr</u>, Name, Stadt, Saldo, Rabatt)
Produkte (<u>PNr</u>, Bez, Gewicht, Preis, Lagerort, Vorrat)
Bestellungen (<u>BestNr</u>, Monat, Tag, KNr, PNr, Menge, Summe, Status)

Fremdschlüsselbedingung ("Referential Integrity"):

Für jeden Wert eines Fremdschlüssels in einer Relation R muß in den referenzierten Relationen jeweils ein Tupel mit demselben Wert als Primärschlüssel existieren, oder der Wert des Fremdschlüssels muß der Nullwert sein.
Beispiel für Nullwert als Fremdschlüssel:

Produkte (PNr, ..., Lagerort)
Lager (Lagerort, Adresse, Verwalter, ...)

5.2 Relationenalgebra

Die Relationenalgebra bildet ein Fundament für Datenbankanfragesprachen (Query Languages), indem sie Grundoperationen für Relationen definiert.

- Eine Operation hat eine oder mehrere Relationen als Operanden und stets wiederum eine Relation als Ergebnis.
  $\rightarrow$ Abgeschlossenheit der Relationenalgebra.
- Das Schema eines Operationsresultats kann von den Schemata der Operanden verschieden sein.

Mengenoperationen:

Für zwei Relationen R, S mit sch(R) = sch(S) sind die üblichen Mengenoperationen definiert:

- Vereinigung (Union) $R \cup S$:
  $sch(R \cup S) = sch(R)$
  $val(R \cup S) = \{t \mid t \in val(R) \vee t \in val(S)\}$
- Durchschnitt (Intersection) $R \cap S$:
  $sch(R \cap S) = sch(R)$
  $val(R \cap S) = \{t \mid t \in val(R) \wedge t \in val(S)\}$
- Differenz (Difference) $R - S$:
  $sch(R - S) = sch(R)$
  $val(R - S) = \{t \mid t \in val(R) \wedge t \notin val(S)\}$

Selektion σ (Filterung, Auswahl von Zeilen einer Tabelle):

Sei F eine Boolesche Formel über einfachen Vergleichsbedingungen zwischen zwei Attributen einer Relation oder einem Attribut und einer Konstanten. Das Resultat einer Selektion $\sigma[F](R)$ auf einer Relation R (auch $\sigma_F(R)$ geschrieben) ist wie folgt definiert:

$sch(\sigma[F](R)) = sch(R)$
$val(\sigma[F](R)) = \{t \mid t \in R \wedge F(t)\}$ wobei F(t) bedeutet, daß t die Bedingung F erfüllt.

Die Menge der möglichen Filterformeln F ist wie folgt präzise definiert:

1) Für Attribute A, B von R mit dom(A)=dom(B), Konstanten $c \in dom(A)$ und Vergleichsoperationen $\theta \in \{=, \neq, <, >, \leq, \geq\}$ sind
   $A \theta B$ und $A \theta c$ zulässige Filterbedingungen.
2) Falls F1 und F2 zulässige Filterbedingungen sind, dann sind auch
   $F1 \wedge F2$, $F1 \vee F2$, $\neg F1$ und (F1) zulässig.
3) Nur die aufgrund von 1) und 2) erzeugten Filterbedingungen sind zulässig.

Projektion π (Auswahl von Spalten einer Tabelle):

Sei $A \subseteq sch(R)$ eine Teilmenge der Attribute einer Relation R. Das Resultat einer Projektion $\pi[A](R)$ auf der Relation R (auch $\pi_A(R)$ geschrieben) ist wie folgt definiert:

$sch(\pi[A](R)) = A$
$val(\pi[A](R)) = \{t \mid \exists\, r \in val(R): t.A = r.A\}$ bzw. ausführlicher
$val(\pi[A](R)) = \{t \mid \exists\, r \in val(R): t.A1 = r.A1 \wedge ... \; t.An = r.An$ für $A=\{A1, ..., An\}\}$

Achtung: Die Projektion beinhaltet eine Duplikateliminierung.

Beispiele für die Selektion:

1) Finde alle Homburger Kunden.
→ σ[Stadt='Homburg'] (Kunden)
→ Resultat:

KNr	Name	Stadt	Saldo	Rabatt
2	Schneider	Homburg	- 800.00	0.20
3	Kirsch	Homburg	0.00	0.10

2) Finde alle Homburger Kunden, die einen Rabatt von mindestens 15 % haben.
→ σ[Stadt='Homburg' ∧ Rabatt >= 0.15] (Kunden)
→ Resultat:

KNr	Name	Stadt	Saldo	Rabatt
2	Schneider	Homburg	- 800.00	0.20

Beispiele für die Projektion:

3) Gib alle Produktbezeichnungen aus.
→ π[Bez] (Produkte)
→ Resultat:

Bez
Papier
Platte
Drucker
Bildschirm
Disketten
Maus
Speicher

4) Gib alle Lagerorte von Produkten aus.
→ π[Lagerort] (Produkte)
→ Resultat:

Lagerort
Homburg
Saarbrücken
Merzig

(Natural) Join |×|

(Natürlicher Verbund, Verbindung zweier Relationen über gleiche Attributnamen und gleiche Attributwerte der Tupel):
Seien R, S Relationen mit Schemata A=sch(R) und B=sch(S). Das Resultat des Joins R |×| S ist wie folgt definiert:

$sch(R \mid\times\mid S) = sch(R) \cup sch(S)$

$val(R \mid\times\mid S) = \{t \mid \exists\, r \in val(R)\ \exists\, s \in val(S): t.A = r.A \wedge t.B = s.B\}$

Ein Sonderfall des Joins, nämlich für $sch(R) \cap sch(S) = \varnothing$, erhält man das kartesische Produkt $R \times S$.

Beispiele für den Join:

1) Führe alle Bestellungen zusammen mit den dazugehörigen Produktdaten auf.
→ Bestellungen | × | Produkt
→ Resultat:

Best Nr	Monat	Tag	KNr	PNr	Menge	Summe	Status	Bez	Gewicht	Preis	Lagerort	Vorrat
1	7	16	1	1	100	1800.00	bezahlt	Papier	2.000	20.00	Homburg	10000
2	7	21	1	1	100	1800.00	bezahlt	Papier	2.000	20.00	Homburg	10000
3	9	30	1	2	4	9000.00	bezahlt	Platte	1.000	2500.00	Saar-brücken	400
. . .												

2) R | × | S mit:
R S

R1	J
a	1
b	2
c	2
d	3

S1	J
w	2
x	2
y	3
z	4

→ Resultat:

R1	J	S1
b	2	w
b	2	x
c	2	w
c	2	x
d	3	y

Achtung: Im allgemeinen kann $\pi\,[sch(R)]\,(R \mid x \mid S) \neq R$ sein.

Zuweisung:

Seien R und S zwei Relationen mit sch(R)={A1, ..., An} und sch(S)={B1, ..., Bn}, so daß für alle i gilt: dom(Ai)=dom(Bi). Die Zuweisung *R := S* bedeutet, daß sich die Ausprägung von R wie folgt ändert: val(R) = val(S). Ausführlicher schreibt man ggf. auch R(A1, ..., An) := S(B1, ..., Bn). Allgemeiner kann man für Ausdrücke E1, ..., En, die über B1, ..., Bn und k-stelligen skalaren Operatoren ψ: $dom(B_{i1}) \times \ldots \times dom(B_{ik}) \rightarrow D$ mit skalarem Resultat im Wertebereich W Zuweisungen der Form R(A1, ..., An) := S(E1, ..., En) vornehmen, wenn der Wertebereich von Ei mit dom(Ai) übereinstimmt. Dabei ist val(R) = {t | es gibt $s \in$ val(S) und t.Ai = Ei(s) für alle i}.
Die Zuweisung ist nur eine Hilfsoperation der Relationenalgebra. Sie ist nützlich als Notation für Zwischenresultate, für Umbenennungen von Attributnamen und um Änderungsoperationen ausdrücken zu können.

Beispiele für komplexere Anfragen:

1) Finden Sie die Namen der Kunden mit negativem Saldo.
   → π[Name] (s[Saldo<0.0] (Kunden))

2) Finden Sie die Namen der Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
   → π[Name] (σ[Status<>'bezahlt' ∧ Monat<10] (Bestellungen) | × | Kunden)
   oder mit Zwischenresultaten:
   B := σ[Status<>'bezahlt' ∧ Monat<10] (Bestellungen)
   BK := B | × | Kunden
   π[Name] (BK)

3) Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg geliefert bekommen haben.
   → π[Name] (σ[Monat >= 9 ∧ Status<>'neu'] (Bestellungen)
   | × | σ[Lagerort='Homburg'] (Produkte)
   | × | σ[Stadt='Homburg'] (Kunden))
   oder mit Zwischenresultaten:
   B := σ[Monat >= 9 ∧ Status<>'neu'] (Bestellungen)
   P := σ[Lagerort='Homburg' (Produkte)
   K := σ[Stadt='Homburg'] (Kunden)
   BP := B | × | P
   BPK := BP | × | K
   π[Name] (BPK)

4) Finden Sie (alle Attribute bzw. die Kundennummern der) Kunden, von denen mindestens eine Bestellung registriert ist.
   → π[sch(Kunden)] (Kunden |×| Bestellungen) bzw.
   π[KNr] (Kunden |×| Bestellungen) oder π[KNr] (Bestellungen)

5) Finden Sie die Kundennummern der Kunden, von denen keine Bestellung registriert ist.
   → π[KNr] (Kunden) - π[KNr] (Kunden | × | Bestellungen)
   oder:
   π[KNr] (Kunden) - π[KNr] (Bestellungen)

6) Finden Sie die Namen der Kunden, von denen keine Bestellung registriert ist.
   → π[Name] (π[KNr,Name] (Kunden) - π[KNr,Name] (Kunden |×| Bestellungen))
   oder:
   π[Name] (Kunden |×| (π[KNr](Kunden) - π[KNr](Bestellungen))

Division ÷ :

Seien R, S Relationen mit Schemata A=sch(R) und B=sch(S), so daß $B \subset A$. Mit Q=A-B sei die Menge der Attribute bezeichnet, die in R vorkommen, nicht aber in S. Das Resultat der Division R ÷ S ist wie folgt definiert:

$sch(R \div S) = A - B$

$val(R \div S) = \{t \mid \forall s \in val(S)\ \exists r \in val(R): r.B = s.B \wedge t = r.Q\}$

Intuitive Bedeutung:
Ein Tupel t ist in R ÷ S genau dann, wenn für alle S-Tupel s ein zusammengesetztes Tupel <t,s> in R enthalten ist.

Bemerkungen:
- Die Division ist die einfachste Art, Anfragen der Form "... für alle ..." auszudrücken.
- Ein Sonderfall der Division ist $R \div \varnothing = \pi[sch(R) - sch(\varnothing)](R)$.

Beispiel für die Division:
Finden Sie die Kundennummern derjenigen Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.

→ π[KNr, PNr] (Bestellungen) ÷ π[PNr] (Produkte)

π[KNr, PNr] (Bestellungen)

KNr	PNr
1	1
1	2
1	3
1	4
1	5
1	6
1	7
2	1
2	5
3	5
3	7
4	1
6	1
6	7

π[PNr] (Produkte)

PNr
1
2
3
4
5
6
7

→ Resultat:

KNr
1

Satz:
Seien R(A,B,C), T(A,B) und S(C) Relationen, so daß $R = T \times S$. Dann gilt: $T = R \div S$.

Kartesisches Produkt × :

Seien R, S zwei Relationen mit Schemata A=sch(R) und B=sch(S). Sei A' ein Schema, bei dem alle Attributnamen Ai, die auch in sch(S) vorkommen, unbenannt sind in R.Ai, und sei B' ein Schema, bei dem alle Attributnamen Ai, die auch in sch(R) vorkommen, umbenannt sind in S.Ai. Das Resultat des kartesischen Produkts R × S ist wie folgt definiert:

$sch(R \times S) = A' \cup B'$

$val(R \times S) = \{t \mid \exists\, r \in val(R)\ \ \exists\, s \in val(S):\ \ t.A' = r.A \text{ und } t.B' = s.B\}$

Intuitive Bedeutung:
R × S enthält alle möglichen Kombinationen von R-Tupeln und S-Tupeln.

Beispiel für das kartesische Produkt:
R × S mit:

R

R1	**J**
a	1
b	2
c	2

S

S1	**J**
w	2
x	2

→ Resultat:

R1	**R.J**	**S1**	**S.J**
a	1	w	2
b	2	w	2
c	2	w	2
a	1	x	2
b	2	x	2
c	2	x	2

Rückführung des Join auf das kartesische Produkt:
Seien R und S zwei Relationen mit Schemata A=sch(R) und B=sch(S), und sei J=sch(R) ∩ sch(S). Es gilt (bis auf Umbenennungen von Attributen):

$R \mid\times\mid S = \pi[A \cup B - \{S.J\}]\ (\ \sigma[R.J=S.J]\ (\ R \times S\ ))$

Rückführung der Division auf das kartesische Produkt:

Satz:
Seien R und S zwei Relationen mit Schemata A=sch(R) und B=sch(S) mit $A \supset B$, und sei Q=sch(R) - sch(S). Es gilt (bis auf Umbenennungen von Attributen):

$R \div S = \pi[Q](R) - \pi[Q] ((\pi[Q](R) \times S) - R)$

Beispiel:

$R := \pi[KNr, PNr]$ (Bestellungen)	
$S := \pi[PNr]$ (Produkte)	
$T1 := \pi[KNr](R) \times S$	alle überhaupt möglichen Bestellungen
$T2 := T1 - R$	potentiell mögliche, aber nicht erfolgte Bestellungen
$T3 := \pi[KNr]$ (T2)	Kunden, die nicht alle Produkte bestellt haben
$T4 := \pi[KNr]$ (R) - T3	Kunden, die alle Produkte bestellt haben

θ-Join:

Seien R, S zwei Relationen mit Schemata sch(R) und sch(S), so daß sch(R) ∩ sch(S) = ∅. Seien A ⊆ sch(R) und B ⊆ sch(S). Sei ferner θ eine der Vergleichsoperationen =, ≠, <, >, ≤, ≥. Das Resultat des θ-Joins R |×|[A θ B] S (auch R $|\times|_{A\theta B}$ S geschrieben) ist wie folgt definiert:

sch(R |×|[A θ B] S) = sch(R × S) = sch(R) ∪ sch(S)
val(R |×|[A θ B] S) = val(σ[A θ B](R × S))

Wenn θ die Vergleichsoperation = ist, dann heißt der θ-Join auch Equijoin.
In analoger Weise läßt sich der θ-Join zu einem Join mit allgemeiner Boolescher Filterformel erweitern, wobei dieselben Formeln wie bei der Selektion zulässig sind.

Beispiele für den θ-Join:

1) Finde alle Kunden, die an einem Lagerort wohnen.
→ π[KNr, Name, Stadt, ...] (Kunden |×|[Stadt=Lagerort] Produkte)

2) Finde alle bisherigen Bestellungen, die den momentanen Vorrat erschöpfen würden.
→ B(BestNr,Monat,Tag,KNr,B.PNr, ...) := Bestellungen(BestNr, Monat, Tag, KNr, PNr, ...)
π[BestNr, ...] (B | × |[B.PNr=PNr ∧ Menge >= Vorrat] Produkte)

3) Finde alle Paare von Kunden, die in derselben Stadt wohnen.
→ K1(K1.KNr, K1.Name, K1.Stadt, ...) := Kunden(KNr, Name, Stadt, ...)
K2(K2.KNr, K2.Name, K2.Stadt, ...) := Kunden(KNr, Name, Stadt, ...)
π[K1.KNr, K2.KNr] (K1 |×|[K1.Stadt = K2.Stadt] K2)
bzw. besser:
π[K1.KNr, K2.KNr] (K1 |×|[K1.Stadt = K2.Stadt ∧ K1.KNr < K2.KNr] K2)
um jedes (echte) Kundenpaar nur einmal aufzulisten

saloppere Schreibweise häufig:
π[KNr1, KNr2] (Kunden |×|[Stadt1 = Stadt2 ∧ KNr1 < KNr2] Kunden)

Semi-Join und Anti-Join:

Seien R, S zwei Relationen mit Schemata sch(R) und sch(S), so daß sch(R) ∩ sch(S) = J.
Das Resultat des Semi-Joins R |× S und das Resultat des Anti-Joins R |- S sind wie folgt definiert:

sch(R |× S) = sch(R)
val(R |× S) = {r | r ∈ val(R) ∧ ∃ s ∈ val(S): r.J = s.J}
sch(R |- S) = sch(R)
val(R |- S) = {r | r ∈ val(R) ∧ ¬ (∃ s ∈ val(S): r.J = s.J)}

Satz:
R |x S = π[R] (R |x| S) und
R |- S = R - π[R] (R |x| S)

Outer Join

Seien R und S zwei Relationen mit Schemata A=sch(R) und B=sch(S), und sei
J=sch(R) $\cap$ sch(S).
Bezeichne ferner ω den Nullwert.
Das Resultat des Outer Joins R |*| S ist wie folgt definiert:

sch (R |*| S) = sch (R) $\cup$ sch (S)

val (R |*| S) = val (R |×| S) $\cup$
$\{t|\exists\ r \in val\ (R) : t.A=r.A \wedge \neg\ (\exists\ s \in val\ (S) : t.J=s.J) \wedge t.(B-J)=\omega\ \} \cup$
$\{t|\exists\ s \in val\ (S) : t.B=s.B \wedge \neg\ (\exists\ r \in val\ (R) : t.J=r.J) \wedge t.(A-J)=\omega\ \}.$

Beispiel für den Outer Join:

Gib alle Kunden mit 5 % Rabatt zusammen mit ihren Bestellungen aus, und zwar auch, wenn für einen Kunden gar keine Bestellungen vorliegen.

σ [Rabatt = 0.05] (Kunden |*| Bestellungen)

KNr	Name	Stadt	Saldo	Rabatt	BestNr	...
5	Becker	Dillingen	0.00	0.05	ω	...
6	Meier	Saarlouis	-3800.00	0.05	14	...
6	Meier	Saarlouis	-3800.00	0.05	15	...
6	Meier	Saarlouis	-3800.00	0.05	16	...

Rückführung des Outer Join auf Join und kartesisches Produkt:
Seien NR und NS Relationen mit sch(NR)=sch(R)-sch(S) und sch(NS)=sch(S)-sch(R), die jeweils genau ein Tupel enthalten, dessen Attribute alle ω als Wert haben.
Dann ist: R |*| S = (R |×| S) $\cup$
((R - π[sch(R)] (R |×| S)) × NS) $\cup$
(NR × (S - π[sch(S)] (R |×| S)))

Left Outer Join:

sch (R ||*| S) = sch (R) $\cup$ sch (S)

val (R ||*| S) = val (R |×| S) $\cup$
$\{t \mid \exists\ r \in val\ (R) : t.A=r.A \wedge \neg\ (\exists\ s \in val\ (S) : t.J=s.J) \wedge t.(B-J)=\omega\ \}.$

Right Outer Join:

sch (R |*|| S) = sch (R) $\cup$ sch (S)

val (R |*|| S) = val (R |×| S) $\cup$
$\{t \mid \exists\ s \in val\ (S) : t.B=s.B \wedge \neg\ (\exists\ r \in val\ (R) : t.J=r.J) \wedge t.(A-J)=\omega\ \}.$

Äquivalenzregeln, Minimalität und Vollständigkeit der Relationenalgebra

Seien R, S, T Relationen, P, P1, P2 Prädikate, R1, R2, S1 Teilmengen der Attribute von R bzw. S. Es gelten u.a. die folgenden Äquivalenzregeln:

Kommutativitätsregeln:

1) $\pi[R1]\ \sigma[P]\ (R) = \sigma[P]\ \pi[R1]\ (R)$
   falls P nur R1-Attribute enthält
2) $R \mid x \mid S = S \mid x \mid R$

Assoziativitätsregeln:

3) $R \mid x \mid (S \mid x \mid T) = (R \mid x \mid S) \mid x \mid T$

Idempotenzregeln:

4) $\pi[R1] (\pi[R2] (R)) = \pi[R1] (R)$
   falls $R1 \subseteq R2$
5) $\sigma[P1] (\sigma[P2] (R)) = \sigma[P1 \wedge P2] (R)$

Distributivitätsregeln:

6) $\pi[R1] (R \cup S) = \pi[R1](R) \cup \pi[R1](S)$
7) $\sigma[P] (R \cup S) = \sigma[P](R) \cup \sigma[P](S)$
8) $\sigma[P] (R \mid x \mid S) = \sigma[P](R) \mid x \mid S$
   falls P nur R-Attribute enthält
9) $\pi[R1,S1] (R \mid x \mid S) = \pi[R1](R) \mid x \mid \pi[S1](S)$
   falls Joinattribute $\subseteq R1 \cup S1$
10) $R \mid x \mid (S \cup T) = (R \mid x \mid S) \cup (R \mid x \mid T)$

Invertierungsregeln:

11) $\pi[sch(R)] (R \parallel * \mid S) = R$

Diese und weitere Äquivalenzregeln bilden die Grundlage für eine automatische Optimierung von deklarativen Anfragen in Form von Ausdrücken der Relationenalgebra.

Definition:
Die Menge der relationenalgebraischen Ausdrücke über einer Menge von Relationen R1, ..., Rn ist wie folgt definiert:

(i) R1, ..., Rn sind Ausdrücke.

(ii) Wenn R, S, T, Q Ausdrücke sind,
F eine Filterformel über dem Resultatschema von R ist,
A eine Teilmenge des Resultatschemas von R ist,
S und T dasselbe Resultatschema haben und sch(R) $\supset$ sch(Q) gilt,
dann sind
$\sigma[F](R)$, $\pi[A](R)$, R |x| S, R x S, R |*| S, S $\cap$ T, S $\cup$ T, S - T, R $\div$ Q
auch Ausdrücke.

(iii) Nur die gemäß (i) und (ii) erzeugten Ausdrücke sind relationalgebraische Ausdrücke.

Satz:
$\times$, π, σ, $\cup$ und - bilden eine minimale Menge von Operationen, mit denen sich alle Operationen der Relationenalgebra ausdrücken lassen.
Beweis: siehe Vorlesung

Definition:
Eine Anfragesprache heißt *relational vollständig*, wenn sich damit alle Anfragen der (minimalen) Relationenalgebra ausdrücken lassen.

5.3 Erweiterte Relationenalgebra

Die "reine" Relationenalgebra ist auf Mengen von Tupeln definiert. In der Einsatzpraxis von Datenbanken werden aber auch Anfragen auf *Multimengen* (und u.U. auch - sortierten - Listen) benötigt.

Definition:
Eine *Multimenge* (engl. multiset, bag) M über einer Grundmenge G ist eine Abbildung
$M: G \rightarrow N_0$ in die natürlichen Zahlen $N_0 = \{0, 1, 2, ...\}$.

M(x) wird als die Häufigkeit von x in M bezeichnet oder auch als die "Anzahl der Duplikate von x". Multimengen werden häufig in einer Pseudomengennotation geschrieben, bei der die Häufigkeit eines Elements durch entsprechend wiederholtes Aufführen des Elements angegeben wird. Beispielsweise ist M = {♣, ♣, ♥, ♠, ♠, ♠} eine Multimenge über G={♣, ♠, ♥, ♦} mit M(♣)=2, M(♥)=1, M(♠)=3 und M(x)=0 für alle anderen $x \in G$.
Für Multimengen M und M' über derselben Grundmenge G gilt $M \subseteq M'$ genau dann, wenn für alle $x \in G$: $M(x) \leq M'(x)$.

Definition:
Eine *Multirelation* R ist ein Paar R = (s,v) mit

- einem Schema s = {A1, ..., An}, das aus einer Menge von *Attributen* (genauer: Attributnamen) besteht und für jedes Attribut Ai einen Domain $dom(Ai) \in \{D1, ..., Dm\}$ festlegt, und
- einer Ausprägung (auch Wert genannt) v, die eine Multimenge über der Grundmenge dom(A1) x dom(A2) x ... x dom(An) ist.

Schema und Ausprägung von R werden - wie bei Relationen - mit sch(r) und val(R) bezeichnet. Eine Multirelation hat in der Regel *keinen* Primärschlüssel.

Beispiel einer Multirelation:

Name	**Stadt**	**Rabatt**
Lauer	Merzig	0.10
Schneider	Homburg	0.20
Schneider	Homburg	0.20
Schulz	Merzig	0.10
Schulz	Merzig	0.05
Meier	Saarlouis	0.05

Jede Relation ist zugleich auch eine Multirelation (mit Häufigkeit 1 für alle Tupel). Umgekehrt kann jede Multirelation R durch eine Funktion χ wie folgt in eine Relation $\chi(R)$ konvertiert werden: $val(\chi(R)) = \{t \mid R(t)>0\}$.

Multimengenoperationen:

Für zwei Multirelationen R, S mit sch(R) = sch(S) sind die folgenden Multimengenoperationen definiert:

- Vereinigung $R \cup_+ S$:
  $sch(R \cup_+ S) = sch(R)$
  $val(R \cup_+ S) = t \mapsto R(t) + S(t)$
  bzw. $\{t \mid t \in val(R) \vee t \in val(S)$ und die Häufigkeit von t ist $R(t)+S(t)\}$
- Durchschnitt $R \cap_+ S$:
  $sch(R \cap_+ S) = sch(R)$
  $val(R \cap_+ S) = t \mapsto min(R(t), S(t))$
  bzw. $\{t \mid t \in val(R) \wedge t \in val(S)$ und die Häufigkeit von t ist $min(R(t), S(t))\}$
- Differenz $R -_+ S$:
  $sch(R -_+ S) = sch(R)$
  $val(R -_+ S) = t \mapsto R(t) - S(t)$ falls $R(t) \geq S(t)$
  0 sonst
  bzw. $\{t \mid t \in val(R)$ und die Häufigkeit von t ist $R(t)-S(t) \geq 0\}$

Selektion σ_+ (Filterung, Auswahl von Zeilen einer Tabelle):

Sei F eine Boolesche Formel über einfachen Vergleichsbedingungen zwischen zwei Attributen einer Multirelation oder einem Attribut und einer Konstanten. Das Resultat einer Selektion $\sigma_+ [F](R)$ auf einer Multirelation R (auch $\sigma_{+F}(R)$ geschrieben) ist wie folgt definiert:

$sch(\sigma_+ [F](R)) = sch(R)$
$val(\sigma_+ [F](R)) = t \mapsto R(t)$ falls $R(t) \wedge F(t)$
0 sonst
bzw. $\{t \mid t \in R \wedge F(t)$ und die Häufigkeit von t ist $R(t)\}$
wobei F(t) bedeutet, daß t die Bedingung F erfüllt.

Die Menge der möglichen Filterformeln F ist wie bei der Selektion auf Relationen definiert.

Projektion π_+ (Auswahl von Spalten einer Tabelle):

Sei $A \subseteq sch(R)$ eine Teilmenge der Attribute einer Multirelation R. Das Resultat einer Projektion $\pi_+ [A](R)$ auf der Multirelation R (auch $\pi_{+A}(R)$ geschrieben) ist wie folgt definiert:

$sch(\pi_+ [A](R)) = A$
$val(\pi_+ [A](R)) = t \mapsto \Sigma \{R(r) \mid r \in val(R)$ und $r.A = t.A\}$
bzw. $\{t \mid \exists r \in val(R)$: $t.A = r.A$ und die Häufigkeit von t ist $|\{d \in val(R) \mid d.A = t.A\}|$ $\}$

Achtung: Die Projektion auf Multirelationen beinhaltet *keine* Duplikateliminierung.

Zuweisung (Umbenennung von Attributen):

Seien R und S zwei Multirelationen mit sch(R)={A1, ..., An} und sch(S)={B1, ..., Bn}, so daß für alle i gilt: dom(Ai)=dom(Bi). Die Zuweisung *R := S* bedeutet, daß sich die Ausprägung von R wie folgt ändert: val(R) = val(S). Ausführlicher schreibt man auch R(A1, ..., An) := S(B1, ..., Bn) bzw. R(A1, ..., An) := S(E1, ..., En) mit Ausdrücken E1, ..., En, die wie bei der Zuweisung für Relationen gebildet werden. Im letzteren Fall ist val(R) = {t | es gibt $s \in S$ und $t.Ai = Ei(s)$ für alle i}.

Kartesisches Produkt $\times_+$ (Kombination von Zeilen zweier Tabellen):
Seien R, S zwei Multirelationen mit Schemata A=sch(R) und B=sch(S). Sei A' ein Schema, bei dem alle Attributnamen Ai, die auch in sch(S) vorkommen, unbenannt sind in R.Ai, und sei B' ein Schema, bei dem alle Attributnamen Ai, die auch in sch(R) vorkommen, umbenannt sind in S.Ai. Das Resultat des kartesischen Produkts $R \times_+ S$ ist wie folgt definiert:

$sch(R \times_+ S) = A' \cup B'$

$val(R \times_+ S) = t \mapsto R(s)*S(s)$ falls $t.A' = r.A$ und $t.B' = s.B$
0 sonst

bzw. $\{t \mid \exists r \in val(R)\ \exists s \in val(S)$: $t.A' = r.A$ und $t.B' = s.B$
und die Häufigkeit von t ist $R(r)*S(s)\}$

Aggregation α_+ (Zusammenfassen von Zeilen einer Tabelle):
Sei $A \in sch(R)$ und sei f eine Funktion, die Multimengen über dom(A) in einen Wertebereich W abbildet (z.B. max, min, sum, count, median). Das Resultat einer Aggregation $\alpha_+ [A,f](R)$ auf der Multirelation R (auch $\alpha_{+A,f}(R)$ geschrieben) ist wie folgt definiert:

$sch(\alpha_+ [A,f](R)) = A'$ mit $dom(A')=W$
$val(\alpha_+ [A,f](R)) = f(\pi_+[A](R))$

Aggregationsfunktionen f bilden eine Multimenge auf einen einzelnen Wert ab.

Gruppierung γ_+ (Zusammenfassen von Äquivalenzklassen der Zeilen einer Tabelle):
Die Wertegleichheit auf einer Multimenge M ist eine Äquivalenzrelation ~ auf den Elementen von M mit Häufigkeit ≥ 1, für die gilt $x \sim y$ genau dann, wenn $x = y$.
Sei $X \subseteq sch(R)$, $A \in sch(R)$, und sei f eine Funktion, die Multimengen über dom(A) in einen Wertebereich W abbildet. Das Resultat einer Gruppierung $\gamma_+ [X,A,f](R)$ auf der Multirelation R (auch $\gamma_{+X,A,f}(R)$ geschrieben) ist eine Multimenge, die wie folgt definiert ist:

$sch(\gamma_+ [X,A,f](R)) = X \cup \{A'\}$ mit $dom(A')=W$
$val(\gamma_+ [X,A,f](R)) = \{ t \mid$ es gibt eine Äquivalenzklasse G von $\pi_+[X](R)$ unter der
Wertegleichheit und t.X ist der Wert der Tupel in G
und $t.A' = f(\pi_+[A](G))\ \}$

Die hier zugrundeliegenden Äquivalenzklassen heißen auch "(Wertegleichheits-) Gruppen".

Beispielanfragen auf Multirelationen:

Produkte:

PNr	Bez	Gewicht	Preis	Lagerort	Vorrat
1	Papier	2.000	20.00	Homburg	10000
2	Platte	1.000	2500.00	Saarbrücken	400
3	Drucker	5.000	2000.00	Merzig	200
4	Bildschirm	5.000	3000.00	Merzig	80
5	Disketten	0.500	20.00	Homburg	5000
6	Maus	0.250	100.00	Homburg	200
7	Speicher	0.100	200.00	Saarbrücken	2000

1) Bestimme die (Anzahl der) Produkte unter 50 DM sowie deren Lagerorte und Preise:
σ_+ [Preis < 50.00] (π_+ [Lagerort, Preis] (Produkte))

Resultat:

Lagerort	**Preis**
Homburg	20.00
Homburg	20.00

2) Bestimme die Gesamtstückzahl (aller Produkte) über alle Lager:
Resultat (Gesamtvorrat) := α_+ [Vorrat, sum](Produkte)

Resultat:

Gesamtvorrat
17880

3) Bestimme für jedes Lager die Gesamtstückzahl aller dort gelagerten Produkte:
Resultat (Lagerort, Gesamtvorrat) := γ_+ [{Lagerort}, Vorrat, sum](Produkte)

Resultat:

Lagerort	**Gesamtvorrat**
Homburg	15200
Saarbrücken	2400
Merzig	280

4) Bestimme für jedes Lager die Gesamtkapitalbindung (Stückzahl * Preis) aller dort gelagerten Produkte:
P1 := π_+ [Lagerort, Preis, Vorrat] (Produkte)
P2 (Lagerort, Wert) := P1(Lagerort, Preis*Vorrat)
Resultat (Lagerort, Kapitalbindung) := γ_+ [{Lagerort}, Wert, sum](P2)

Resultat:

Lagerort	**Kapitalbindung**
Homburg	320 000
Saarbrücken	1 400 000
Merzig	640 000

Weitere Erweiterungen der Relationenalgebra

Orthogonal zu der Erweiterung auf Multimengen und Listen oder auch auf geschachtelte Relationen (sog. NF^2-Relationen: Non First Normal Form), bei denen Tupelkomponenten selbst ganze Relationen sein dürfen, kann man die Relationenalgebra auch noch um mächtigere Operatoren ergänzen, die nicht auf die Standardoperatoren zurückführbar sind. Die wichtigste Operation dieser Kategorie ist die transitive Hülle von binären (d.h. zweistelligen) Relationen.

Transitive Hülle R^+ einer binären Relation R:
Sei R(A, B) eine binäre Relation mit dom(A)=dom(B). Das Resultat der transitiven Hülle R^+ ist wie folgt definiert:

$sch(R^+) = sch(R)$

$val(R^+)$ ist die kleinste Menge, für die gilt:

(i) für jedes $r \in R$ gilt $r \in R^+$ und

(ii) für $t \in R^+$ und $r \in R$ mit t.B=r.A gilt $(t.A, r.B) \in R^+$

R^+ ist die kleinste transitiv abgeschlossene Menge, die R enthält.
R^+ ist die (bzgl. $\subseteq$) kleinste Lösung V der Fixpunktgleichung $V = R \cup (V \,|\times|\, [B=A]\, R)$.

Wenn man R als Kanten eines gerichteten Graphen interpretiert, dom(A)=dom(B) also Knotenmengen sind, ist R^+ die Menge aller Knotenpaare, die über einen Weg verbunden sind.

Beispiel:
Flugverbindungen := π[Abflugort, Zielort] (Flüge)

Flüge

FlugNr	Abflugort	Zielort	...
LH58	Frankfurt	Chicago	
AA371	Chicago	Phoenix	
DA77	Phoenix	Yuma	
AA70	Frankfurt	Dallas	
AA351	Dallas	Phoenix	
UA111	Chicago	Dallas	

Flugverbindungen

Abflugort	Zielort
Frankfurt	Chicago
Frankfurt	Dallas
Frankfurt	Phoenix
Frankfurt	Yuma
...	...

Ergänzende Literatur zu Kapitel 5

Jeffrey D. Ullman: Principles of Database and Knowledge Base Systems Vol. 1, Computer Science Press, 1988

David Maier: The Theory of Relational Databases, Computer Science Press, 1983

Serge Abiteboul, Richard Hull, Victor Vianu: Foundations of Databases, Addison-Wesley, 1995

Giedrius Slivinskas, Christian S. Jensen, Richard T. Snodgrass: A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. IEEE Transactions on Knowledge and Data Engineering Vol. 13 No.1, 2001

Kapitel 6:
Logikorientierte Anfragesprachen

6.1 Grundlagen aus der Logik

Logik ist eine wichtige Grundlage, um über die Semantik von Datenbankanfragesprachen (und später auch über die Semantik von Softwareentwurfssprachen) präzise reden zu können. In einer Logik werden zunächst einmal *Formeln* als syntaktische Gebilde definiert. Auf Formeln kann man dann *Ableitungs-, Beweis- oder Deduktionsregeln* definieren, mit denen man Formeln aus einer gegebenen Formelgrundmenge mechanisch herleiten kann; wir schreiben

$F_1,...,F_n \vdash G$ oder $\frac{F_1,...,F_n}{G}$ für die Ableitung von Formel G aus den Formeln F_1, ..., F_n.

Eine Menge von (schematisierten) Ableitungsregeln nennt man einen *Ableitungs-, Beweis- oder Deduktionskalkül*.
Dieser syntaktischen (*beweistheoretischen*) Seite einer Logik steht die (*modellorientierte*) *Semantik* von Formeln gegenüber. Dazu werden bestimmte Komponenten der Formeln, insbesondere Aussagen-, Funktions- und Prädikatsymbole, in einer (mathematischen) Struktur interpretiert (z.B. den natürlichen Zahlen mit Funktionen wie Addition, Multiplikation, usw. und Prädikaten wie IstPrimzahl oder SindTeilerfremd); auf dieser Basis erhalten ganze Formeln einen Wahrheitswert in der der *Interpretation* zugrundeliegenden Struktur. Wenn eine gegebene Formelmenge G unter einer bestimmten Interpretation ψ in einer Struktur S wahr ist, nennt man die Struktur ein *Modell* der Formelmenge, und wir schreiben $\psi \models G$ oder $\models_\psi G$.

Zwischen syntaktischer Ableitung und semantischer Interpretation besteht typischerweise ein Zusammenhang: idealerweise wünscht man sich, daß für eine gegebene Grundformelmenge G und eine Struktur S, die ein Modell von G ist, jede mit einem Deduktionskalkül ableitbare Formel in S wahr ist (*Korrektheit des Kalküls*) und umgekehrt jede in S wahre Formel aus G syntaktisch ableitbar ist (*Vollständigkeit des Kalküls*). Für bestimmte Strukturen gibt es korrekte und vollständige Kalküle, für andere nicht. Beispielsweise besagt der berühmte Satz von Gödel, daß es für die Struktur der arithmetischen Gleichungen über den natürlichen Zahlen mit den Funktionen Addition und Multiplikation und darauf aufgebauten prädikatenlogischen Formeln keinen vollständigen Ableitungskalkül geben kann.

6.1.1 Aussagenlogik

Definition:
Eine atomare Formel der Aussagenlogik ist eine Aussagenkonstante True oder False oder eine Aussagevariable der Form Ai (i=1, 2, ...).
Die Menge der Formeln der Aussagenlogik ist induktiv wie folgt definiert:

(i) Jede atomare Formel ist eine Formel.

(ii) Wenn F und G Formeln sind, dann auch (F), $\neg$F, F$\wedge$G, F$\vee$G sowie F$\Rightarrow$G als Kurzschreibweise für $\neg$F$\vee$G und F$\Leftrightarrow$G als Kurzschreibweise für (F$\wedge$G)$\vee$($\neg$F$\wedge\neg$G) Formeln. Statt $\neg$F schreibt man häufig auch $\overline{F}$.

Die Formeln der Aussagenlogik sind also Aussageformen über Aussagevariablen.
Um die syntaktische Analyse von Ausdrücken eindeutig zu machen, kann man entweder vollständige Klammerung verlangen (d.h. in (ii) nur (F), ($\neg$F), (F$\wedge$G), (F$\vee$G), (F$\Rightarrow$G), (F$\Leftrightarrow$G) zulassen) oder Präzedenzen festlegen, daß $\neg$ stärker bindet als $\wedge$ oder $\vee$ und diese wiederum stärker als $\Rightarrow$ oder $\Leftrightarrow$.

Definition:
Sei F eine Formel der Aussagenlogik mit atomarer Formelmenge D. Eine *passende Struktur* S zu F ist eine Menge P von Aussagen (die jeweils wahr oder falsch sind), so daß jede Aussagenvariable in D einer dieser Aussagen zugeordnet werden kann. P enthält ggf. die immer wahre Aussage 1 und die immer falsche Aussage 0. Eine *Interpretation* von F ist eine Abbildung ψ: D $\rightarrow$ P, für die gilt ψ(True)=1, ψ(False)=0, ψ(Ai)=1 bzw. 0, falls ψ(Ai) in S wahr bzw. falsch ist.
ψ wird wie wie folgt auf beliebige Formeln F, G, usw. über D fortgesetzt:
(i) ψ((F)) = ψ(F); (ii) ψ(F$\wedge$G)= 1 falls ψ(F)= ψ(G)=1, 0 sonst;
(iii) ψ(F$\vee$G)= 1 falls ψ(F)= 1 oder ψ(G)=1, 0 sonst; (iv) ψ($\neg$F)=1 falls ψ(F)=0, 0 sonst.

Eine Interpretation einer aussagenlogischen Formel ist also im wesentlichen eine Belegung der Aussagevariablen mit Wahrheitswerten.

Definition:
Sei F eine Formel und ψ eine Interpretation von F.
Wenn ψ(F)=1 ist, heißt ψ *Modell* von F. Wir schreiben dann ψ |= F (oder $|=_\psi$ F).
F heißt *erfüllbar*, falls F mindestens ein Modell hat, ansonsten *unerfüllbar*.
F heißt *(allgemein-)gültig* oder *Tautologie*, falls jede Interpretation in einer zu F passenden Struktur ein Modell von F ist.

Satz:
F ist Tautologie genau dann, wenn $\neg$F unerfüllbar ist.

Definition:
G heißt *Folgerung* (Konsequenz) von F1, ..., Fk, geschrieben F1, ..., Fk |= G, wenn für jede Interpretation ψ in einer passenden Struktur gilt: falls ψ ein Modell von F1, ..., Fk ist, dann ist ψ auch ein Modell von G. F und G heißen (semantisch) *äquivalent*, geschrieben F $\equiv$ G, falls für jede Interpretation ψ gilt: ψ(F)= ψ(G).

Satz:
G ist Folgerung von F1, ..., Fk genau dann, wenn (F1 $\wedge$ F2 $\wedge$ ... $\wedge$ Fk) $\Rightarrow$ G eine Tautologie ist.
F und G sind äquivalent genau dann, wenn F Folgerung von G ist und umgekehrt.

Beispiele:

1) Formeln über Variablen T, P:

F1: $T \Rightarrow \neg P$, F2: $P \wedge \neg T$ bzw. zusammengefaßt

F: $(T \Rightarrow \neg P) \wedge (P \wedge \neg T)$

Interpretation: $\psi(T)$ = „7 ist durch 2 teilbar“ (falsch), $\psi(P)$ = „7 ist eine Primzahl“ (wahr)

ψ ist ein Modell von F1 und F2 bzw. von F.

2) Formeln über Variablen S, R, A:

F1: $S \Rightarrow \neg R$, F2: $\neg R \Rightarrow S$, F3: $A \Rightarrow \neg S$, F4: A bzw. zusammengefaßt

F: $(S \Rightarrow \neg R) \wedge (\neg R \Rightarrow S) \wedge (A \Rightarrow \neg S) \wedge (A)$

Interpretation:

$\psi(S)$ = „die Sonne scheint“ (falsch) , $\psi(R)$ = „es regnet“ (wahr), $\psi(A)$ = „es ist April“ (wahr).

ψ ist ein Modell von F1, F2, F3, F4 bzw. von F.

3) $\neg(\neg F) \Leftrightarrow F$ ist eine Tautologie;

$(X \wedge Y) \vee (\neg X \wedge Z)$ ist erfüllbar (z.B. mit X: „dieser Raum ist nichtleer“, Y: „das Licht ist an“ und Z: „das Licht ist aus“), aber keine Tautologie;

$F \wedge \neg F$ ist unerfüllbar.

Ein **Deduktionskalkül** ist eine endliche Menge von Ableitungsregeln der Form $P \vdash K$, wobei P eine endliche Formelmenge, die Menge der Prämissen (oder Hypothesen), ist und K eine Formel, die Konklusion oder Schlussfolgerung. Ein einfacher **Deduktionskalkül für die Aussagenlogik** besteht (z.B.) aus 5 Regeln:

(i)		$\vdash F \Rightarrow (F \Rightarrow F)$	
(ii)		$\vdash (F \Rightarrow (G \Rightarrow H)) \Rightarrow ((F \Rightarrow G) \Rightarrow (F \Rightarrow H))$	
(iii)		$\vdash (\neg F \Rightarrow \neg G) \Rightarrow (G \Rightarrow F)$	
(iv)	$F \Leftrightarrow G, H$	$\vdash H[F/G]$	(Ersetzungsregel, Substitution)
(v)	$F \Rightarrow G, F$	$\vdash G$	(Abtrennungsregel, Modus Ponens)

wobei H[F/G] die Formel ist, die man aus H durch syntaktische Substitution von F durch G erhält.

Definition:

Sei H eine endliche Formelmenge – die Hypothesenmenge (z.B. Axiome einer spezifischen Struktur). Die Formel F heißt aus H ableitbar, in Zeichen: $H \vdash F$, wenn es eine endliche Sequenz F0, F1, ..., Fn von Formeln gibt mit Fn=F, so daß für alle Fi gilt: Fi ist Element von H, Fi ist eine der prämissenlosen Formeln (i) bis (iii) des Deduktionskalküls oder Fi ist aus F0, ..., F(i-1) mittels Regel (iv) oder (v) des Deduktionskalküls abgeleitet.

Für $H=\varnothing$ heißt F Theorem des Deduktionskalküls.

Satz:

Der angegebene Deduktionskalkül mit den Regeln (i) bis (v) ist korrekt und vollständig, d.h.

$H \vdash F$ genau dann, wenn $H \models F$

(d.h. $H \Rightarrow F$ ist eine Tautologie bzw. F ist wahr in jeder Struktur, in der H wahr ist)

und – als Sonderfall mit $H=\varnothing$: $\vdash F$ genau dann, wenn $\models F$

(d.h. F ist eine Tautologie bzw. F ist in jeder passenden Struktur wahr)

Beweis: durch Induktion über den Aufbau von F und entsprechende Fallunterscheidungen

Definition:
Eine (endliche) Formelmenge H heißt Axiomensystem für eine Formelmenge M (z.B. alle wahren Theoreme einer mathemat. Struktur), wenn: $\{\psi \mid \psi$ ist Modell von H$\} = \{\psi \mid \psi$ ist Modell von M$\}$.

Alternative Deduktionskalküle könnten z.B. die Brute-Force-Methode verwenden, Wahrheitstafeln für $H \Rightarrow F$ aufzustellen (mit allen Kombinationen möglicher Wahrheitswerte für die Aussagenvariablen und alle Teilformeln), oder eine geeignete Teilmenge der folgenden (aus dem angegebenen Kalkül ableitbaren und so beweisbaren) Äquivalenzen und der Deduktionsregeln (i) bis (v) verwenden:

$\neg\neg F \Leftrightarrow F$	(Doppelnegation)
$F \wedge F \Leftrightarrow F$	(Idempotenz)
$F \vee F \Leftrightarrow F$	
$F \wedge G \Leftrightarrow G \wedge F$	(Kommutativität)
$F \vee G \Leftrightarrow G \vee F$	
$F \wedge (G \wedge H) \Leftrightarrow (F \wedge G) \wedge H$	(Assoziativität)
$F \vee (G \vee H) \Leftrightarrow (F \vee G) \vee H$	
$F \wedge (F \vee G) \Leftrightarrow F$	(Absorption)
$F \vee (F \wedge G) \Leftrightarrow F$	
$F \wedge (G \vee H) \Leftrightarrow (F \wedge G) \vee (F \wedge H)$	(Distributivität)
$F \vee (G \wedge H) \Leftrightarrow (F \vee G) \wedge (F \vee H)$	
$\neg (F \wedge G) \Leftrightarrow (\neg F \vee \neg G)$	(Gesetz von de Morgan)
$\neg (F \vee G) \Leftrightarrow (\neg F \wedge \neg G)$	
$F \vee 1 \Leftrightarrow 1$	(Tautologieregeln)
$F \wedge 1 \Leftrightarrow F$	
$F \vee 0 \Leftrightarrow F$	(Unerfüllbarkeitsregeln)
$F \wedge 0 \Leftrightarrow 0$	
$F \vee (\neg F) \Leftrightarrow 1$	(Tertium non datur)
$F \wedge (\neg F) \Leftrightarrow 0$	(Kontradiktion)
$((F \Leftrightarrow G) \wedge H) \Leftrightarrow ((F \Leftrightarrow G) \wedge H[F/G])$	(Substitution)

Beispiel:
Auf die Frage „Worin besteht das Geheimnis Ihres Lebens?“ antwortet ein Hundertjähriger:
Wenn ich kein Bier zu einer Mahlzeit trinke, dann habe ich immer Fisch.
Immer wenn ich Fisch und Bier zur selben Mahlzeit habe, verzichte ich auf Eiscreme.
Wenn ich Eiscreme habe oder Bier meide, dann rühre ich Fisch nicht an.

Modellierung:
Formel G: $(\neg B \Rightarrow F) \wedge ((F \wedge B) \Rightarrow \neg E) \wedge ((E \vee \neg B) \Rightarrow \neg F)$

Vereinfachung, sprich Deduktion (mit Unterstreichung als Negation):

$G \Leftrightarrow (B \vee F) \wedge (\underline{F} \wedge \underline{B} \vee \underline{E}) \wedge ((\underline{E} \vee \neg \underline{B}) \vee \underline{F})$	nach Definition der Implikation
$\Leftrightarrow (B \vee F) \wedge (\underline{F} \vee \underline{B} \vee \underline{E}) \wedge ((\underline{E} \wedge B) \vee \underline{F})$	nach dem Gesetz von de Morgan
$\Leftrightarrow (B \vee F) \wedge (\underline{F} \vee \underline{B} \vee \underline{E}) \wedge (\underline{E} \vee \underline{F}) \wedge (B \vee \underline{F})$	nach dem Distributivitätsgesetz
$\Leftrightarrow ((B \vee F) \wedge (B \vee \underline{F})) \wedge ((\underline{F} \vee \underline{B} \vee \underline{E}) \wedge (\underline{E} \vee \underline{F}))$	wegen Kommutativität und Assoziativität
$\Leftrightarrow ((B \vee (F \wedge \underline{F})) \wedge ((\underline{F} \vee \underline{E}) \vee (\underline{B} \wedge 0))$	nach Distributivitätsgesetz und Tautologieregel
$\Leftrightarrow ((B \vee 0) \wedge ((\underline{F} \vee \underline{E}) \vee 0)$	wegen Kontradiktion und Unerfüllbarkeit
$\Leftrightarrow B \wedge (\underline{F} \vee \underline{E})$	wegen Unerfüllbarkeitsregel

Interpretation: Trinke zu jeder Mahlzeit Bier, und iss niemals Fisch mit Eiscreme!

3.2 Prädikatenlogik 1. Ordnung

Motivation: Man möchte kompliziertere Zusammenhänge in einer formalen Logik präzisieren können, die in der Aussagenlogik nicht ausdrückbar sind.
Beispiele:
1) Es gibt unendlich viele Primzahlen:
   $\forall q\ \exists p\ \forall x,y\ (\ p>q \wedge ((x>1 \wedge y>1) \Rightarrow (x \cdot y \neq p))\ )$ bzw.
   $\forall q\ \exists p\ \forall x,y\ (\text{greater}(p,q) \wedge ((\text{greater}(x,1) \wedge \text{greater}(y,1)) \Rightarrow (\neg\ \text{equal}(\text{mult}(x,y),p)))\ )$
2) Das kgV zweier teilerfremder Primzahlen ist ihr Produkt:
   $\forall x,y\ (\text{ggT}(x,y)=1 \Rightarrow \text{kgV}(x,y)=x \cdot y)$
3) Invarianten von Programmen (mit dem Gesamtziel der Programmverifikation)
4) Datenbankanfragen (siehe Kapitel 4) und Datenbankinvarianten (siehe Kapitel 7)

Definition:
Gegeben seien Variablen xi (i=1,2, ...), Prädikatsymbole Pi der Stelligkeit ki (i=1,2, ...), d.h. Prädikatsymbole mit ki Argumenten, und Funktionssymbole fi der Stelligkeit li(i=1,2, ...).
0-stellige Prädikatsymbole sind Aussagen, 0-stellige Funktionssymbole sind Konstanten.
Terme sind induktiv definiert:

(i) Jede Variable ist ein Term.
(ii) Wenn t1, ..., tk Terme sind und fi ein k-stelliges Funktionssymbol ist, dann ist auch fi(t1, ..., tk) ein Term.

Eine *atomare Formel* hat die Form Pi(t1, ..., tk) mit einem k-stelligen Prädikatsymbol Pi und Termen t1, ..., tk.
Formeln der Prädikatenlogik 1. Ordnung sind induktiv definiert:

(i) Jede atomare Formel ist eine Formel.
(ii) Für Formeln F und G sind auch $\neg$F, F$\wedge$G, F$\vee$G und (F) Formeln.
(iii) Für eine Variable xi und eine Formel F sind auch $\forall$xi (F) und $\exists$xi (F) Formeln.
$\forall$ und $\exists$ heißen All- bzw. Existenzquantor, xi ist eine quantifizierte (gebundene) Variable, und F ist der Rumpf der Formel.

Zur eindeutigen Syntaxanalyse kann man entweder vollständige Klammerung verlangen oder Präzedenzen festlegen, so daß Junktoren ($\neg$, $\wedge$, $\vee$) stärker binden als Quantoren und die Präzedenzen der Aussagenlogik gelten. Statt $\forall x\ \forall y\ \forall z$ ... schreiben wir auch kurz $\forall x,y,z$, und Analoges gilt für $\exists$.

Eine Variable x in einer Formel F heißt *gebunden*, wenn x in einer Teilformel der Form $\forall$x (F) oder $\exists$x (F) vorkommt; ansonsten heißt x *frei*. Formeln ohne freie Variablen heißen geschlossen.
Beispiel: In $\forall x\ (P(x,y) \wedge \exists z\ (Q(z,x)))$ sind x und z gebunden, und y ist frei.

Als notationelle Konvention für Formeln verlangen wir, daß keine Variable an mehr als einen Quantor gebunden wird und daß keine Variable sowohl frei als auch gebunden vorkommt. Dies ist durch einfache Umbenennung von Variablen (Bereinigung) immer möglich.
Zur einfacheren Lesbarkeit werden häufig stilistische Konventionen für Bezeichner eingeführt: Variablen werden mit x, y, z, u, v, w, ... bezeichnet, Konstanten mit a, b, c, ..., Funktionssymbole der Stelligkeit ≥ 1 mit f, g, h, ... und Prädikatsymbole mit P, Q, R, S, ...

Semantik von prädikatenlogischen Formeln

Definition:
Gegeben sei eine Menge von Funktions- und Prädikatsymbolen (z.B. alle in einer Formel oder Formelmenge vorkommenden Symbole) mit entsprechenden Stelligkeiten. Eine dazu passende *Struktur* ist ein Tripel S = (U, fun, pred) mit

- einem *Universum* (einer *Trägermenge*) U von Individuen (z.B. ganzen Zahlen oder Zeichenketten),
- einer Menge fun von *Funktionen* fi: $U \times ... \times U \rightarrow U$ der Stelligkeit li, so daß jedes Funktionssymbol der Stelligkeit li eine Funktion derselben Stelligkeit existiert,
- einer Menge pred von *Prädikaten* Pi: $U \times ... \times U \rightarrow \{0,1\}$ der Stelligkeit ki, so daß jedes Prädikatsymbol der Stelligkeit ki ein Prädikat derselben Stelligkeit existiert.

Die *Interpretation* ψ einer Formel F in einer passenden Struktur S = (U, fun, pred) ist eine Abbildung, die das Universum U festlegt und jedem Funktions- und Prädikatsymbol in F eine passende Funktion aus fun bzw. ein passendes Prädikat aus pred zuordnet. Dies ist also eine Abbildung ψ von atomaren Formeln ohne Variablen als Argumenten auf wahre oder falsche Prädikate in der gegebenen Struktur.
ψ wird wie wie folgt auf beliebige Formeln F, G, usw. und Terme über denselben Funktions- und Prädikatsymbolen fortgesetzt:

(i) $\psi((F)) = \psi(F)$; (ii) $\psi(F \wedge G) = 1$ falls $\psi(F) = \psi(G) = 1$, 0 sonst;
(iii) $\psi(F \vee G) = 1$ falls $\psi(F) = 1$ oder $\psi(G) = 1$, 0 sonst; (iv) $\psi(\neg F) = 1$ falls $\psi(F) = 0$, 0 sonst,
(v) $\psi(f(t1, ..., tk)) = \psi(f)(\psi(t1), ..., \psi(tk))$ für ein Funktionssymbol f und Terme t1, ..., tk,
(vi) $\psi(x) = a$ mit einem beliebigen $a \in U$ für eine (freie) Variable x
(vii) $\psi(P(t1, ..., tk)) = \psi(P)(\psi(t1), ..., \psi(tk))$ für ein Prädikatsymbol P und Terme t1, ..., tk,
(viii) $\psi(\forall x (F)) = 1$ falls $\psi(F[x/a]) = 1$ für alle $a \in U$, wobei F[x/a] die Formel ist, die man aus F erhält, wenn man alle Vorkommen von x durch a ersetzt, und
(ix) $\psi(\exists x (F)) = 1$ falls $\psi(F[x/a]) = 1$ für ein beliebiges $a \in U$.

Beispiel 1:
F: $\forall q \exists p \forall x,y$ (greater(p,q) $\wedge$ ((greater(x,1) $\wedge$ greater(y,1)) $\Rightarrow$ ($\neg$ equal(mult(x,y),p))))
Eine passende Struktur ist z.B. S = (N_0, {+, ·}, {>, =}).
Interpretation ψ:
add $\mapsto$ +, mult $\mapsto$ ·, greater $\mapsto$ >, equal $\mapsto$ =, q $\mapsto$ 1, p $\mapsto$ 2, x $\mapsto$ 3, y $\mapsto$ 4
ψ (F) = für jede natürliche Zahl q gibt es eine natürlich Zahl p, so daß für alle Zahlen x und y gilt:
p ist größer als q und falls x und y beide größer als 1 sind, ist x·y von p verschieden
(oder einfacher: zu jeder Zahl gibt es eine Primzahl, die größer ist als die Zahl)

Beispiel 2:
F: K (1, Lauer, Merzig) $\wedge$ K(2, Schneider, Homburg) $\wedge$ P(1, Papier) $\wedge$ B(7, 16, 1, 1, 100)
mit Prädikatsymbolen K, P und B und Konstanten 1, Lauer, usw.
Eine passende Struktur ist z.B. eine Datenbank mit dem folgenden Datenbankschema (d.h. eine Menge von Relationen über passenden kartesischen Produkten des Universums $N_0 \cup \Sigma^*$, der Menge aller natürlichen Zahlen und Zeichenketten über einem Standardalphabet Σ):
Kunden (KNr, Name, Stadt), Produkte (PNr, Bez), Bestellungen (Monat, Tag, KNr, PNr, Menge).
ψ (F) = 1, falls es die entsprechenden vier Tupel in der Datenbank gibt, 0 sonst.

Erweiterungen der Prädikatenlogik 1. Ordnung:

Eine Variation der Prädikatenlogik 1. Ordnung ist die *mehrsortige (typisierte) Variante*, bei der in einer passenden Struktur das Universum aus einer endlichen Menge von *Trägermengen (Sorten)* U1, ..., Um besteht und Funktionen und Prädikate mehrsortig sind mit einer *Signatur*, die neben der Stelligkeit die Sorten der Funktions- und Prädikatargumente festlegt. Eine solche Struktur nennt man auch eine mehrsortige Algebra.

Beispiel: U= {U1, U2} mit U1=R^n, U2=R und

Funktionen Addition: U1 × U1 → U1, Multiplikation: U1 × U2 → U1,
Skalarprodukt: U1 × U1 → U2, Vektorprodukt: U1 × U1 → U1, Nullvektor: → U1 sowie
Prädikaten Orthogonal: U1 × U1 → {0,1} usw.

Der Übergang zu einer mehrsortigen Logik verändert nicht die Ausdruckskraft der Prädikatenlogik 1. Ordnung. Man gewinnt jedoch an Ausdruckskraft, wenn man Prädikate schachteln kann, also Prädikate selbst Prädikate als Argument haben können, oder wenn man nicht nur Individuen, sondern auch Prädikate quantifizieren (also an Quantoren binden) darf. Solche Erweiterungen führen auf die *Prädikatenlogik höherer Ordnung*, die wir hier nicht weiter betrachten.

Ein Beispiel einer Formel höherer Ordnung ist die folgende Spezifikation der transitiven Hülle H einer binären Relation R:

$$\forall x,y\ (\ H(x,y) \Leftrightarrow \forall P\ (\ (R(x,y) \Rightarrow P(x,y)) \wedge \forall u,w,z\ (P(u,w) \wedge P(w,z) \Rightarrow P(u,z)))$$
$$\Rightarrow (H(x,y) \Rightarrow P(x,y))\ )$$

Intuitive Interpretation :
H ist kleinste binäre Relation, die R enthält und transitiv abgeschlossen ist.
Also gilt für jede binäre Relationen P, dass sie, wenn sie R enthält und transitiv abgeschlossen ist, eine Obermenge von H sein muss.

Erfüllbarkeit, Unerfüllbarkeit, Gültigkeit, Modell

Definition:
Sei F eine Formel der Prädikatenlogik 1. Ordnung und ψ eine Interpretation von F.
Wenn $\psi(F)=1$ ist, heißt ψ *Modell* von F. Wir schreiben dann $\psi \models F$ (oder $\models_\psi F$).
F heißt *erfüllbar*, falls F mindestens ein Modell hat, ansonsten *unerfüllbar*.
F heißt *(allgemein-)gültig* oder *Tautologie*, falls jede Interpretation in einer zu F passenden Struktur ein Modell von F ist.

Satz:
F ist Tautologie genau dann, wenn $\neg$F unerfüllbar ist.

Definition:
G heißt *Folgerung* (Konsequenz) von F1, ..., Fk, geschrieben F1, ..., Fk $\models$ G, wenn für jede Interpretation ψ in einer passenden Struktur gilt: falls ψ ein Modell von F1, ..., Fk ist, dann ist ψ auch ein Modell von G. F und G heißen (semantisch) *äquivalent*, geschrieben F $\equiv$ G, falls für jede Interpretation ψ gilt: $\psi(F)= \psi(G)$.

Satz:
G ist Folgerung von F1, ..., Fk genau dann, wenn $(F1 \wedge F2 \wedge ... \wedge Fk) \Rightarrow G$ eine Tautologie ist.
F und G sind äquivalent genau dann, wenn F Folgerung von G ist und umgekehrt.

Definition:
Eine Formelmenge H heißt ein Axiomensystem für eine Formelmenge M (z.B. alle wahren Theoreme einer mathematischen Struktur), falls gilt: $\{\psi \mid \psi$ ist Modell von H$\} = \{\psi \mid \psi$ ist Modell von M$\}$.

Beispiel:
Formeln F1, ..., F5 bzw. Formel $F= F1 \wedge ... \wedge F5$:
$F1 = \forall x,y,z \ (G(f(f(x,y),z), f(x,f(y,z))))$
$F2 = \forall x \ (G(f(x,e),x))$
$F3 = \forall x,y \ (G(x, y) \Leftrightarrow G(y, x))$
$F4 = \forall x \ \forall y \ \forall z \ (G(x,y) \wedge G(y,z) \Rightarrow G(x,z))$
$F5 = \forall x \ \exists y \ (G(f(x,y), e))$

Interpretation A (Modell):
U = Z (Menge der ganzen Zahlen), ψ(G) = Gleichheit, ψ(f) = Addition, ψ(e) = 0, ψ(F)=1 (wahr)
Interpretation B (kein Modell):
U = Z (Menge der ganzen Zahlen), ψ(G) = Gleichheit, ψ(f) = Multiplikation, ψ(e) = 1, ψ(F)=0
Interpretation C (Modell):
U = R (Menge der reellen Zahlen), ψ(G) = Gleichheit, ψ(f) = Multiplikation, ψ(e) = 1, ψ(F)=1
Interpretation D (kein Modell):
$U = N_0$ (Menge der natürlichen Zahlen), $\psi(G) = \geq$, ψ(f) = Addition, ψ(e) = 1, ψ(F)=0
Interpretation E (kein Modell):
$U = \Sigma^*$, ψ(G) = Gleichheit der Länge, ψ(f) = Stringkonkatenation, $\psi(e) = \varepsilon$, ψ(F)=0

Deduktionskalküle

Es gibt Deduktionskalküle für die Prädikatenlogik 1. Ordnung, die korrekt und vollständig sind, z.B. den Resolutionskalkül oder den Tableaukalkül. Diese spielen z.B. in der Logikprogrammierung (z.B. in der Programmiersprache Prolog) oder beim automatischen Theorembeweisen eine Rolle. Es handelt sich jedoch nur um sog. Semientscheidungsverfahren: bei einer semantischen Tautologie als Eingabe terminiert das Ableitungsverfahren mit dem Resultat "gültig" (rein technisch beweist der Resolutionskalkül die Unerfüllbarkeit der negierten Eingabe), bei einer Nichttautologie (deren Negation erfüllbar ist) aber muß das Verfahren nicht terminieren.

Mittels dieser Kalküle (oder auch auf anderem Wege) kann man die Gültigkeit der folgenden - praktisch sehr nützlichen - Umformungsregeln beweisen:

alle aussagenlogischen Äquivalenzen
Wenn $F \Leftrightarrow G$ und F als Teilformel in H vorkommt, dann ist $H \Leftrightarrow H[F/G]$
$\neg \forall(F) \Leftrightarrow \exists x (\neg F)$
$\neg \exists (F) \Leftrightarrow \forall x (\neg F)$
$(\forall x (F)) \wedge G \Leftrightarrow \forall x (F \wedge G)$ falls x in G nicht frei vorkommt
$(\forall x (F)) \vee G \Leftrightarrow \forall x (F \vee G)$ falls x in G nicht frei vorkommt
$(\exists x (F)) \wedge G \Leftrightarrow \exists x (F \wedge G)$ falls x in G nicht frei vorkommt
$(\exists x (F)) \vee G \Leftrightarrow \exists x (F \vee G)$ falls x in G nicht frei vorkommt
$(\forall x (F)) \wedge (\forall y (G)) \Leftrightarrow \forall x (F \wedge G[y/x])$, falls x in G nicht frei vorkommt
$(\exists x (F)) \vee (\exists y (G)) \Leftrightarrow \exists x (F \vee G[y/x])$, falls x in G nicht frei vorkommt
$\forall x (\forall y (F)) \Leftrightarrow \forall y (\forall x (F))$
$\exists x (\exists y (F)) \Leftrightarrow \exists y (\exists x (F))$
$\forall x (F) \Rightarrow F[x/a]$ für alle Konstanten a
$((\forall x (F)) \wedge (F \Rightarrow G)) \Rightarrow (\forall x (G))$

Achtung: Es gelten jedoch *nicht*
$\exists x (\forall y (F)) \Leftrightarrow \forall y (\exists x (F))$
$(\forall x (F)) \vee (\forall y (G)) \Leftrightarrow \forall x (F \vee G[y/x])$
$(\exists x (F)) \wedge (\exists y (G)) \Leftrightarrow \exists x (F \wedge G[y/x])$

Beispiel:

Folgende Zusammenhänge seien wahr:

i) Informatikstudenten können programmieren.
ii) Studenten mit guten Mathematiknoten studieren Informatik.
iii) Studenten, die mit einem Informatiker (jemandem, der Informatik studiert) verwandt sind, haben gute Mathematiknoten.
iv) Alle saarländischen Studenten sind mit Heinz Becker verwandt.
v) Die Verwandtschaftsbeziehung ist symmetrisch.
vi) Heinz Becker hat Informatik studiert.

Modellierung:

Universum: Menge aller Studenten
Prädikate:
KannProgrammieren P(x)
StudiertInformatik I(x)
HatGuteMathenoten M(x)
SindVerwandt V(x,y)
IstSaarländer S(x)
Funktionen und Konstanten:
HeinzBecker hb()
Formeln:

i) $\forall x (I(x) \Rightarrow P(x)) \wedge$
ii) $\forall x (M(x) \Rightarrow I(x)) \wedge$
iii) $\forall x \ \forall y ((V(x,y) \wedge I(y)) \Rightarrow M(x)) \wedge$
iv) $\forall x (S(x) \Rightarrow V(x,hb)) \wedge$
v) $\forall x \ \forall y ((V(x,y) \Leftrightarrow V(y,x)) \wedge$
vi) $I(hb)$

Vereinfachung, sprich Deduktion:

(iii) $\wedge$ (iv) $\wedge$ (vi):

$\forall x \ \forall y (I(hb) \wedge (S(x) \Rightarrow V(x,hb)) \wedge (V(x,y) \wedge I(y)) \Rightarrow M(x)))$
$\vdash \ \forall x \ (I(hb) \wedge (S(x) \Rightarrow V(x,hb)) \wedge (V(x,hb) \wedge I(hb)) \Rightarrow M(x)) \ )$
$\vdash \ \forall x \ (S(x) \Rightarrow M(x)) \ (*)$

(*) $\wedge$ (ii) $\wedge$ (i):

$\forall x \ ((S(x) \Rightarrow M(x)) \ \wedge \ (M(x) \Rightarrow I(x)) \ \wedge \ (I(x) \Rightarrow P(x)))$
$\vdash \ \forall x \ (S(x) \Rightarrow P(x))$

Fazit: alle Saarländer können programmieren!

6.2 Domain-Relationenkalkül (DRK)

Anfragen sind prädikatenlogische Mengenspezifikationen der Form

$\{<x1, ..., xn> \mid F(x1, ..., xn)\}$

wobei F ein prädikatenlogischer Ausdruck über einer Menge von Domain-Variablen ist mit x1, ..., xn als einzigen freien Variablen.

Die Menge der zulässigen prädikatenlogischen Ausdrücke F ist wie folgt präzise definiert:

1) Für Domain-Variablen x1, ..., xn und eine Relation R mit n Attributen ist $<x1,...,xn> \in R$ (auch R(x1, ..., xn) geschrieben) ein zulässiger Ausdruck.
2) Für Domain-Variablen x, y, Konstanten c und Vergleichsoperationen $\theta \in \{=, \neq, <, >, \leq, \geq\}$ sind $x \theta y$ und $x \theta c$ zulässige Ausdrücke.
3) Falls F1 und F2 zulässige Ausdrücke sind, dann sind auch $F1 \wedge F2$, $F1 \vee F2$, $\neg F1$ und (F1) zulässig.
4) Falls F ein zulässiger Ausdruck mit einer freien Domain-Variable x ist, dann sind auch $\exists x: F(x)$ und $\forall x: F(x)$ zulässige Ausdrücke.
5) Falls F ein zulässiger Ausdruck ist, dann ist auch (F) zulässig.
6) Nur die aufgrund von 1) bis 5) erzeugten Ausdrücke sind zulässig.

Semantik des Domain-Relationenkalküls:

Eine *Datenbank* über einem Schema mit Relationen R1(A11, ..., A1n_1), ..., Rm(Am1, ..., Amn$_m$) wird als Formelmenge H aufgefaßt, die für jede Relation Ri ein Prädikatsymbol Ri verwendet und für jedes Tupel <a1, ..., an$_i$> $\in$ val(Ri) eine atomare Formel Ri(a1, ..., an$_i$) enthält. Die Datenbank an sich ist dann ein Modell dieser Formelmenge H. Alternativ kann man H als eine einzige Formel auffassen, die aus der Konjunktion aller dieser atomaren Formeln (über alle Tupel einer Relation und alle Relationen der Datenbank) besteht.

Beispiel:
Für eine Datenbank mit dem Schema
K (KNr, Name, Stadt), P (PNr, Bez), B (Monat, Tag, KNr, PNr, Menge)
und der Ausprägung
val(K) = {<1, Lauer, Merzig>, <2, Schneider, Homburg>},
val(P) = {<1, Papier>}
val(B) = {<7, 16, 1, 1, 100>, <7, 21, 1, 1, 100>, <10, 26, 2, 1, 100>}
ist
H = K(1, Lauer, Merzig) $\wedge$ K(2, Schneider, Homburg) $\wedge$ P(1, Papier) $\wedge$
B(7, 16, 1, 1, 100) $\wedge$ B(10, 26, 2, 1, 100)

Eine Anfrageergebnis für eine Anfrage der Form {<x1, ..., xn> | F(x1, ..., xn)} wird als (potentielle) Folgerung aus der Datenbankformel H bzw. als Beweisziel aufgefasst. Man möchte also zeigen, daß H |= F bzw. H |— F gilt.
Wenn F keine freien Variablen enthält, das Resultat der Anfrage also 1 oder 0 ist, ist die Semantik der Anfrage somit:
1, wenn H |= F und 0 sonst.
Wenn F freie Variablen enthält, was der Regelfall ist, dann ist die Semantik der Anfrage (das Anfrageresultat) die Menge aller möglichen Interpretationen der freien Variablen, also Substitutionen der Variablen durch Individuenkonstanten aus dem Universum U der Datenbankstruktur, so daß F aus H folgt, also:
{<a1, a2, ... an> | a1, a2, ..., an $\in$ U und H |= F[x1/a1, x2/a2, ..., xn/an]}.

Beispiel:
Semantik (Anfrageresultat) von {<k, n> | K(k, n, Merzig)} ("alle Kunden aus Merzig"):
{<1, Lauer>}
Semantik (1 oder 0) von P(1, Speicher) ("gibt es ein Produkt mit PNr 1 und Bez. Speicher"): 0

Beispiele:

1) Finden Sie (die Namen) alle(r) Kunden mit negativem Saldo.
→ {<k,n,st,sa,r> | <k,n,st,sa,r> ∈ Kunden ∧ sa < 0.0} bzw.
→ {<n> | ∃ k,st,sa,r: <k,n,st,sa,r> ∈ Kunden ∧ sa < 0.0}

2) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
→ {<n> | ∃ k,st,sa,r: <k,n,st,sa,r> ∈ Kunden ∧
∃ b,m,t,p,me,su,stat: <b,m,t,k,p,me,su,stat> ∈ Bestellungen ∧
m < 10 ∧ stat ≠ 'bezahlt'}

3) Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg geliefert bekommen haben, jeweils mit der Bezeichnung des entsprechenden Produkts.
→ {<n,bez> | ∃ k,st,sa,r: <k,n,st,sa,r> ∈ Kunden ∧
∃ p,g,pr,l,v: <p,bez,g,pr,l,v> ∈ Produkte ∧
∃ b,m,t,me,su,stat: <b,m,t,k,p,me,su,stat> ∈ Bestellungen ∧
st='Homburg' ∧ mw9 ∧ l='Homburg'}

4) Finden Sie die Kunden, von denen mindestens eine Bestellung registriert ist.
→ {<k,n,st,sa,r> | <k,n,st,sa,r> ∈ Kunden ∧
∃ b,m,t,p,me,su,stat: <b,m,t,k,p,me,su,stat> ∈ Bestellungen}

5) Finden Sie die Kunden, von denen keine Bestellung registriert ist.
→ {<k,n,st,sa,r> | <k,n,st,sa,r> ∈ Kunden ∧
∀ b,m,t,k',p,me,su,stat:
<b,m,t,k',p,me,su,stat> ∈ Bestellungen ⇒ k≠k'}

6) Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.
→ {<k,n,st,sa,r> | <k,n,st,sa,r> ∈ Kunden ∧
∀ p: (∃ bez,g,pr,l,v: <p,bez,g,pr,l,v> ∈ Produkte) ⇒
(∃ b,m,t,me,su,stat: <b,m,t,k,p,me,su,stat> ∈ Bestellungen) }

Eine Anfragesprache auf der Basis des Domain-Relationenkalküls:
Query-by-Example (QBE)
(bzw. MS-Access u.a. als kommerzielle Variante)
Idee:
Bedingungen eines Ausdrucks des Domain-Relationenkalküls werden in Form von "Beispielelementen" in entsprechende Fenster eingetragen, wobei für jedes Relationenschema ein Fenster vorgegeben wird. Gleiche Elemente in verschiedenen Fenstern stehen für "Joinbedingungen".
Beispiel:
Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg (oder Saarbrücken) geliefert bekommen haben, dessen Preis über 100 DM liegt, jeweils mit der Bezeichnung des entsprechenden Produkts.

Kunden

KNr	Name	Stadt	Saldo	Rabatt
_55	P.	Homburg		

Bestellungen

BestNr	Monat	Tag	KNr	PNr	Menge	Summe	Status
	>= 9		_55	_33			

Produkte

PNr	Bez	Gewicht	Preis	Lagerort	Vorrat
_33	P.		> 100.00	Homburg Saarbrücken	

6.2 Tupel-Relationenkalkül (TRK)

Der Domain-Relationenkalkül hat den Vorteil, daß er unmittelbar aus der Prädikatenlogik 1. Ordnung abgeleitet ist und deren Standardnotation weitgehend übernimmt. Er hat aber auch den Nachteil, daß diese Notation bei Datenbankschemata mit vielen Attributen, etwas ausladend wird, weil sehr viele Domain-Variable eingeführt werden müssen. Eine kompaktere Schreibweise stellt der Tupel-Relationenkalkül bereit, bei dem Variablen, wie z.B. t, für ganze Tupel stehen und auf Attribute mit der Notation t.A Bezug genommen wird.
Anfragen sind prädikatenlogische Mengenspezifikation der Form

$\{t \mid F(t)\}$,

wobei F ein prädikatenlogischer Ausdruck über einer Menge von Tupelvariablen ist mit t als einziger freier Variable, oder der Form

$\{<t1.A1, ..., tn.An> \mid F(t1, ..., tn)\}$,

wobei F ein prädikatenlogischer Ausdruck über einer Menge von Tupelvariablen ist mit t1, ..., tn als (nicht notwendigerweise verschiedenen) einzigen freien Variablen und A1, ..., An Attributnamen sind.

Die Menge der zulässigen prädikatenlogischen Ausdrücke F ist wie folgt präzise definiert:

1) Für eine Tupelvariable r und eine Relation R ist $r \in R$ (auch R(r) geschrieben) ein zulässiger Ausdruck.
2) Für Tupelvariable r, s und Attribute A, B mit dom(A)=dom(B), Konstanten $c \in dom(A)$ und Vergleichsoperationen $\theta \in \{=, \neq, <, >, \leq, \geq\}$ sind $r.A\ \theta\ s.B$ und $r.A\ \theta\ c$ zulässige Ausdrücke.
3) Falls F1 und F2 zulässige Ausdrücke sind, dann sind auch $F1 \wedge F2$, $F1 \vee F2$, $\neg F1$ und (F1) zulässig.
4) Falls F ein zulässiger Ausdruck mit einer freien Tupelvariable r ist, dann sind auch $\exists\ r: F(r)$ und $\forall\ r: F(r)$ zulässige Ausdrücke.
5) Falls F ein zulässiger Ausdruck ist, dann ist auch (F) zulässig.
6) Nur die aufgrund von 1) bis 5) erzeugten Ausdrücke sind zulässig.

Semantik des Tupel-Relationenkalküls:

Die Semantik des Tupel-Relationenkalküls ist durch eine Übersetzung in den Domain-Relationenkalkül gegeben. Diese ist wie folgt definiert:
In einer Anfrage der Form $\{t \mid F(t)\}$ oder $\{<t.B1, ..., t.Bn> \mid F(t)\}$ wird jeder Term der Form r.Ai mit einer Tupelvariablen r durch eine Domain-Variable ra_i ersetzt und jeder Term der Form $r \in R$ über einer Relation mit Schema R(A1, ..., Am) durch einen Term $R(ra_1, ..., ra_m)$ mit Domain-Variablen $ra_1, ..., ra_m$.
Die Semantik der TRK-Anfrage ist dann gerade die der durch diese Übersetzung entstehenden DRK-Anfrage.

Beispiele:

1) Finden Sie (die Namen) alle(r) Kunden mit negativem Saldo.
   $\rightarrow \{t \mid t \in Kunden \wedge t.Saldo < 0.0\}$ bzw. $\{t.Name \mid t \in Kunden \wedge t.Saldo < 0.0\}$

2) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
   $\rightarrow \{t.Name \mid t \in Kunden \wedge$
   $\exists\, b: b \in Bestellungen \wedge$
   $t.KNr=b.KNr \wedge b.Monat < 10 \wedge b.Status \neq 'bezahlt'\}$

3) Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg geliefert bekommen haben, jeweils mit der Bezeichnung des entsprechenden Produkts.
   $\rightarrow \{k.Name, p.Bez \mid k \in Kunden \wedge p \in Produkte \wedge$
   $\exists\, b: b \in Bestellungen \wedge$
   $k.Stadt='Homburg' \wedge b.Monat \geq 9 \wedge p.Lagerort='Homburg' \wedge$
   $k.KNr=b.KNr \wedge b.PNr=p.PNr\}$

4) Finden Sie die Kunden, von denen mindestens eine Bestellung registriert ist.
   $\rightarrow \{t \mid t \in Kunden \wedge \exists\, b: b \in Bestellungen \wedge b.KNr=t.KNr\}$

5) Finden Sie die Kunden, von denen keine Bestellung registriert ist.
   $\rightarrow \{t \mid t \in Kunden \wedge \neg(\exists\, b: b \in Bestellungen \wedge b.KNr=t.KNr)\}$ oder
   $\rightarrow \{t \mid t \in Kunden \wedge \forall\, b: b \in Bestellungen \Rightarrow b.KNr \neq t.KNr\}$

6) Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.
   $\rightarrow \{t \mid t \in Kunden \wedge \quad \forall\, p: p \in Produkte \Rightarrow$
   $(\exists\, b: b \in Bestellungen \wedge b.PNr=p.PNr \wedge b.KNr=t.KNr)\}$

Domain-unabhängige Ausdrücke

Problem: Anfragen liefern u.U. unendliche Resultatmengen.

Beispiel: $\{t \mid \neg(t \in R)\}$ für eine endliche Relation R.

Lösungsidee:

Eine Formel des Tupelrelationenkalküls heißt *domain-unabhängig* genau dann, wenn für jede mögliche Ausprägung der Datenbank die Resultatmenge der Formel nur von der Datenbank, nicht aber von den zugrundeliegenden Domains abhängt.

Man sollte nur domain-unabhängige Formeln als Anfragen verwenden.

Präzisierung:

Sei F eine geschlossene Formel des Tupelrelationenkalküls. Eine *Interpretation* von F ist eine Abbildung der Relationsnamen, Konstanten und Variablen in F auf Relationsausprägungen, elementaren Konstanten und Tupeln, die aus einem *Universum* von Werten gebildet werden. Wir sagen, daß F erfüllt ist, wenn die Interpretation von F in dem entsprechenden Universum erfüllt ist (bei kanonischer Interpretation der Junktoren und Quantoren).

Sei F(t) eine Formel des Tupelrelationenkalküls mit einer einzigen freien Variable t (in einer Anfrage $\{t \mid F(t)\}$). Die Interpretation von F(t) ist die Menge aller Tupel, die aus Elementen des gewählten Universums gebildet werden, so daß bei Substitution der Tupel für die freie Variable t die Formel F(t) im Universum erfüllt ist.

Wahl des Universums:

- Variante 1 - *unbeschränkte Interpretation*:
  das Universum besteht aus der Vereinigung aller Domains der Datenbank
- Variante 2 - *beschränkte Interpretation*:
  das Universum besteht aus dem *aktiven Domain* der Datenbank und einer Formel F,
  d.h. der Vereinigung aller Attributwerte der Relationsausprägungen der Datenbank
  und aller Konstanten in F

Beispiele:

Sei R eine Relation mit sch(R)={A}, dom(A)={1,2}, val(R)={<1>}.

Der Ausdruck $\{t \mid \neg (t \in R)\}$ hat
als unbeschränkte Interpretation den Wert {<2>} und
als beschränkte Interpretation den Wert Ø.

Der Ausdruck $\{t \mid \exists\ x : \neg (x=t) \wedge x.A=1\}$ hat
als unbeschränkte Interpretation den Wert {<2>} und
als beschränkte Interpretation den Wert Ø.

Der Ausdruck $\{t \mid t \in R \wedge \forall\ x : x=t\}$ hat
als unbeschränkte Interpretation den Wert Ø und
als beschränkte Interpretation den Wert {<1>}.

Definition:

Eine Formel des Tupelrelationenkalküls heißt *domain-unabhängig* genau dann, wenn ihre unbeschränkte Interpretation für jede mögliche Wahl von Domains, die den aktiven Domain umfassen, mit ihrer beschränkten Interpretation übereinstimmt.

Sichere Ausdrücke

Die "Sicherheit" von Ausdrücken ist eine hinreichende, syntaktisch prüfbare Bedingung für Domain-Unabhängigkeit.

Die Variable x in der Formel F(x) heißt beschränkt, wenn für jede Interpretation von F gilt:
wenn F(x) wahr ist, dann liegt die Interpretation von x im aktiven Domain von F.

Die Variable x in der Formel F(x) heißt unbeschränkt, wenn für jede Interpretation von F gilt:
wenn die Interpretation von x nicht im aktiven Domain von F liegt, muß F(x) wahr sein.

Durch syntaktische Einschränkungen von Formeln können
- freie Variablen beschränkt werden
  (so daß in {t | F(t)} die Formel F(t) nur für t aus dem aktiven Domain erfüllbar ist),
- durch Existenzquantoren gebundene Variablen beschränkt werden
  (so daß in $\exists$ x : G(x) die Formel G(x) nur für x aus dem aktiven Domain erfüllbar ist) und
- durch Allquantoren gebundene Variablen unbeschränkt werden
  (so daß in $\forall$ x : G(x) die Formel G(x) für Interpretationen von x, die nicht im aktiven Domain liegen, immer wahr ist).

Definition:

Sei F(t) eine Formel des Tupelrelationenkalküls mit freier Variable t. Zu jeder Tupelvariablen in F läßt sich ein Schema (d.h. eine Menge Attributen) aus F und dem Datenbankschema ableiten.

Die *Beschränktheit* und die *Unbeschränktheit* der Attribute von Tupelvariablen in einer Formel F sind wie folgt induktiv definiert.

1) In t $\in$ R sind alle Attribute von t beschränkt.
2) In t.A=c mit einer Konstanten c ist t.A beschränkt.
3) In r.A=s.B $\wedge$ F ist r.A beschränkt, wenn s.B in F beschränkt ist,
   und s.B beschränkt, wenn r.A in F beschränkt ist..

4a) In F $\wedge$ G ist t.A genau dann beschränkt, wenn t.A in F oder in G beschränkt ist.
4b) In F $\wedge$ G ist t.A genau dann unbeschränkt, wenn t.A in F und in G unbeschränkt ist.
5a) In F $\vee$ G ist t.A genau dann beschränkt, wenn t.A in F und in G beschränkt ist.
5b) In F $\vee$ G ist t.A genau dann unbeschränkt, wenn t.A in F oder in G unbeschränkt ist.
6a) In $\neg$F ist t.A beschränkt genau dann, wenn t.A in F unbeschränkt ist.
6b) In $\neg$F ist t.A unbeschränkt genau dann, wenn t.A in F beschränkt ist.
7a) In $\exists$ x: F(x) ist t.A. beschränkt genau dann, wenn t.A in F beschränkt ist.
7b) In $\exists$ x: F(x) ist t.A. unbeschränkt genau dann, wenn t.A in F unbeschränkt ist.
8a) In $\forall$ x: F(x) ist t.A beschränkt genau dann, wenn t.A in F beschränkt ist.
8b) In $\forall$ x: F(x) ist t.A unbeschränkt genau dann, wenn t.A in F unbeschränkt ist.
9) Nur die gemäß 1) bis 8) beschränkten bzw. unbeschränkten Attribute von Tupelvariablen sind beschränkt bzw. unbeschränkt.

Eine Formel des Tupelrelationenkalküls ist *sicher* (engl.: safe) genau dann, wenn

a) alle Attribute aller freien Tupelvariablen beschränkt sind,
b) für jede Teilformel der Form $\exists$ x: P(x) alle Attribute von x in P beschränkt sind,
c) für jede Teilformel der Form $\forall$ x: P(x) alle Attribute von x in P unbeschränkt sind.

Satz:

Jede sichere Formel des Tupelrelationenkalküls ist domain-unabhängig.

Beispiele (für das Schema R(A,B) und z.B. die Ausprägung R={<2,2>}):

- $\{t \mid \neg(\, t \in R) \vee t.A=2 \vee t.B=2\}$ ist unsicher.
- $\{t \mid \neg(\, t \in R) \wedge t.A=2\}$ ist unsicher.
- $\{t \mid \neg(\, t \in R) \wedge t.A=2 \wedge t.B=2\}$ ist sicher.

- $\{t \mid \exists\, s: (\, s \in R \wedge s.A=t.A\,)\}$ ist unsicher.
- $\{t \mid \exists\, s: (\, s \in R \vee s=t\,)\}$ ist unsicher.
- $\{t \mid \exists\, s: (\, s \in R \wedge s=t\,)\}$ ist sicher.
- $\{t \mid \neg(\, \exists\, s: (\, s \in R \wedge s=t\,)\,)\}$ ist unsicher.

- $\{t \mid \forall\, s: (\, s \in R \wedge s=t\,)\}$ ist unsicher.
- $\{t \mid \forall\, s: (\, s \in R \Rightarrow s=t\,)\}$ ist unsicher.
- $\{t \mid \forall\, s: (\, s \in R \Rightarrow (\, s=t \wedge t.A=2 \wedge t.B=2\,)\,)\}$ ist unsicher.
- $\{t \mid t \in R \wedge \forall\, s: (\, s \in R \Rightarrow (\, s=t \wedge t.A=2 \wedge t.B=2\,)\,)\}$ ist sicher.
- $\{t \mid \forall\, s: (\, s \in R \wedge (\, s=t \wedge t.A=2 \wedge t.B=2\,)\,)\}$ ist unsicher.
- $\{t \mid \forall\, s: (\, s \in R \Rightarrow \neg(s=t)\,)\}$ ist unsicher.

- $\{t \mid t \in K \wedge \forall\, p: p \in P \Rightarrow (\, \exists\, b: b \in B \wedge b.PNr=p.PNr \wedge b.KNr=t.KNr\,)\}$ ist sicher.

Satz:
Eine Anfrage des Tupelrelationenkalküls ist sicher, wenn
- die Anfrage die Form hat $\{t \mid t \in R \wedge F(t)\}$,
- jede existenzquantifizierte Teilformel die Form hat $\exists\, x: \; x \in R \wedge F(x)$ und
- jede allquantifizierte Teilformel die Form hat $\forall\, x: s \in R \Rightarrow F(x)$.

Für den Domain-Relationenkalkül sind Domain-Unabhängigkeit und Sicherheit analog definiert.

4.3 Mächtigkeit relationaler Anfragesprachen

Anfragesprachen existierender Datenbanksysteme basieren überwiegend entweder auf dem Tupel-Relationenkalkül (z.B. Ingres/Quel) oder dem Domain-Relationenkalkül (z.B. das GUI von MS Access) oder einer Kombination von Relationenalgebra und Tupel-Relationenkalkül (SQL).

Satz:
Jede Anfrage, die sich mit der Relationenalgebra ausdrücken lässt, lässt sich sowohl mit dem sicheren Tupel-Relationenkalkül als auch dem sicheren Domain-Relationenkalkül ausdrücken.
Jede Anfrage, die sich mit dem sicheren Tupel-Relationenkalkül ausdrücken lässt, lässt sich sowohl mit der Relationenalgebra als auch dem sicheren Domain-Relationenkalkül ausdrücken.
Jede Anfrage, die sich mit dem sicheren Domain-Relationenkalkül ausdrücken lässt, lässt sich sowohl mit der Relationenalgebra als auch dem sicheren Tupel-Relationenkalkül ausdrücken.
Beweis: siehe Vorlesung
Bemerkung:
Relationenalgebra, sicherer Tupel-Relationenkalkül und sicherer Domain-Relationenkalkül haben also dieselbe Ausdrucksmächtigkeit.

4.4 Datalog: Deduktionsregeln als Anfragesprache

In DBS werden Informationen traditionellerweise ausschließlich *extensional* - in Form von *Daten (Fakten)* - repräsentiert. In wissensbasierten Systemen und in modernen, um deduktive Fähigkeiten erweiterten DBS können Informationen zusätzlich auch *intensional* - in Form von *Regeln* - repräsentiert werden. Mit Hilfe der Regeln können aus den Fakten weitere Informationen hergeleitet werden.

Beispiel mit rein extensionaler Repräsentation:

Flüge (F)

FlugNr	Abflugort	Zielort	...
LH58	Frankfurt	Chicago	
AA371	Chicago	Phoenix	
DA77	Phoenix	Yuma	
AA70	Frankfurt	Dallas	
AA351	Dallas	Phoenix	
UA111	Chicago	Dallas	

Flugverbindungen (V)

Abflugort	Zielort
Frankfurt	Chicago
Frankfurt	Dallas
Frankfurt	Phoenix
Frankfurt	Yuma
...	...

Beispiel mit intensionaler Repräsentation der Flugverbindungen:

als Fixpunktgleichung in der Relationenalgebra:

$V = F \cup (V |x| [V.Zielort=F.Abflugort] F)$

mit der Semantik, daß V die bzgl. $\subseteq$ kleinste Relation ist, für die Fixpunktgleichung erfüllt ist.

als Menge von (Prolog-artigen) Regeln:

Flugverbindungen (a,z) :- Flüge (a,z)
Flugverbindungen (a,z) :- Flugverbindungen (a,o), Flüge (o,z)

als Formeln des Domain-Relationenkalküls:

$\forall$ a, z: Flüge (a,z) $\Rightarrow$ Flugverbindungen (a,z)
$\forall$ a, z, o: Flugverbindungen (a,o) $\wedge$ Flüge (o,z) $\Rightarrow$ Flugverbindungen (a,z)

bzw.

$\forall$ a, z: $<a,z> \in$ Flüge $\Rightarrow$ $<a,z> \in$ Flugverbindungen
$\forall$ a, z, o: $<a,o> \in$ Flugverbindungen $\wedge$ $<o,z> \in$ Flüge $\Rightarrow$ $<a,z> \in$ Flugverbindungen

Datalog

Definitionen:

Eine *Hornklausel* über einer endlichen Menge von Prädikatsymbolen ist eine logische Formel der Form

$\forall$ X1, ..., Xn:

P1 (Z11, Z12, ..., Z1k_1) $\wedge$ ... $\wedge$Pm (Zm1, Zm2, ..., Zmk_m) $\Rightarrow$ P0 (Z01, Z02, ..., Z0k_0)

mit k_i-stelligen Prädikaten Pi, Variablen X1, ..., Xn und

Konstanten oder Variablen Zij, so daß Zij entweder eine Konstante ist oder eine der Variablen X1, ..., Xn.

Die Formel

P0 (Z01, Z02, ..., Z0k_0)

wird als "Kopf" (engl.: head) der Hornklausel bezeichnet,

die Formel

P1 (Z11, Z12, ..., Z1k_1) $\wedge$ ... $\wedge$Pm (Zm1, Zm2, ..., Zmk_m)

als "Rumpf" (engl.: body).

Gegeben sei eine endliche Menge von Prädikaten.

Ein *Datalog-Programm* besteht aus

- einer Menge von Fakten der Form Pi (V1, V2, ..., Vk)
  mit einem k-stelligen Prädikat Pi und Konstanten V1, ..., Vk und
- einer Menge von Regeln in Form von Hornklauseln über den gegebenen Prädikaten.

Ein *Datalog-Programm mit Negation* besteht aus

- einer Menge von Fakten wie in einem einfachen Datalog-Programm und
- einer Menge von Regeln in Form von Klauseln vom selben Typ wie bei Datalog,
  bei denen jedoch Prädikate im Rumpf negiert sein können.

Eine *Anfrage* (Ziel; engl: goal) für ein Datalog-Programm ist ein Ausdruck der Form P(Z1, ..., Zk) mit einem k-stelligen Prädikat P und Konstanten oder Variablen Z1, ..., Zk, so daß mindestens eines der Zj eine (freie) Variable ist.

Eine Regel r heißt *rekursiv*, wenn ihr Kopfprädikat auch in ihrem Rumpf vorkommt oder - allgemeiner – wenn es Regeln r1, r2, ..., rn gibt mit r1 = rn = r, so daß für alle i ein Rumpfprädikat von ri im Kopf von r(i+1) steht. Man nennt dann auch das Kopfprädikat von r rekursiv.

Eine rekursive Regel heißt *linear*, wenn das Kopfprädikat genau einmal unter den Rumpfprädikaten auftaucht und keines der anderen Rumpfprädikate als Kopfprädikat einer anderen rekursiven Regel auftritt.

Ein Datalog-Programm heißt linear, wenn es nur lineare oder nichtrekursive Regeln hat.

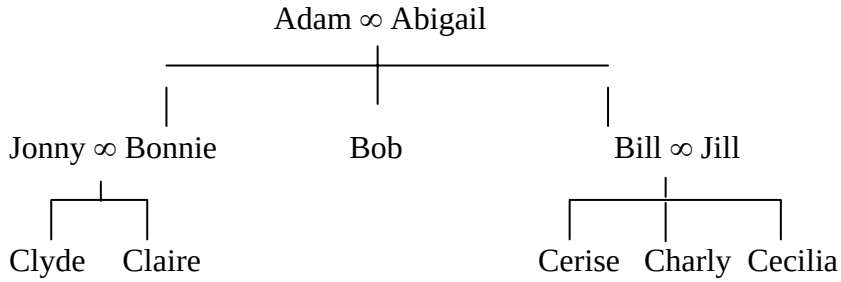
Satz:

Die Menge der Anfragen, die sich mit Datalog ohne Rekursion und ohne Negation ausdrücken lassen, ist eine echte Untermenge der Anfragen, die sich mit dem Domain-Relationenkalkül ausdrücken lassen.

Die Menge der Anfragen, die sich mit Datalog mit Negation, aber ohne Rekursion ausdrücken lassen, ist identisch mit der Menge der Anfragen, die sich mit dem Domain-Relationenkalkül ausdrücken lassen.

Die Menge der Anfragen, die sich mit Datalog mit Rekursion und mit Negation ausdrücken lassen, ist eine echte Obermenge der Menge von Anfragen, die sich mit dem Domain-Relationenkalkül ausdrücken lassen.

Beispiele:



Fakten

Mann (Adam), Mann (Jonny), Mann (Bob), Mann (Bill), Mann (Clyde) ,Mann (Charly)
Frau (Abigail), Frau (Bonnie), Frau (Jill), Frau (Claire), Frau (Cerise), Frau (Cecilia)
Ehepaar (Adam, Abigail), Ehepaar (Jonny, Bonnie), Ehepaar (Bill, Jill)
Elternteil (Adam, Bonnie), Elternteil (Adam, Bob), Elternteil (Adam, Bill),
Elternteil (Abigail, Bonnie), Elternteil (Abigail, Bob), Elternteil (Abigail, Bill),
Elternteil (Jonny, Clyde), Elternteil (Jonny, Claire),
Elternteil (Bonnie, Clyde), Elternteil (Bonnie, Claire),
Elternteil (Bill, Cerise), Elternteil (Bill, Charly), Elternteil (Bill, Cecilia),
Elternteil (Jill, Cerise), Elternteil (Jill, Charly), Elternteil (Jill, Cecilia)
SelbeGeneration (Adam, Abigail),
SelbeGeneration (Adam, Adam), SelbeGeneration (Abigail, Abigail)
SpieltMit (Clyde, Charly)

Regeln

Elternteil (X, Y) $\Rightarrow$ Vorfahren (X, Y)
Elternteil (X, Y) $\wedge$ Vorfahren (Y, Z) $\Rightarrow$ Vorfahren (X, Z)
Elternteil (X, Y) $\wedge$ Mann (X) $\Rightarrow$ Vater (X, Y)
Elternteil (X, Y) $\wedge$ Frau (X) $\Rightarrow$ Mutter (X, Y)
Elternteil (X, Y) $\wedge$ Elternteil (X, Z) $\wedge$ Y$\neq$Z $\Rightarrow$ Geschwister (Y, Z)
Elternteil (X, Y) $\wedge$ Elternteil (U, W) $\wedge$ Geschwister (X, U) $\wedge$ Frau (W) $\Rightarrow$ Cousine (Y, W)
Elternteil (X, Y) $\wedge$ Elternteil (U, W) $\wedge$ SelbeGeneration (X, U) $\Rightarrow$ SelbeGeneration (Y, W)
Geschwister (X, Y) $\Rightarrow$ SpieltMit (X, Y)
SpieltMit (Clyde, Y) $\Rightarrow$ SpieltMit (Claire, Y)
TRUE $\Rightarrow$ SpieltMit (Cecilia, Y) (leerer Rumpf)
Elternteil (X, Y) $\Rightarrow$ Verwandt (X, Y)
Geschwister (X, Y) $\Rightarrow$ Verwandt (X, Y)
Verwandt (X, Y) $\Rightarrow$ Verwandt (Y, X)
Verwandt (X, Y) $\wedge$ Verwandt (Y, Z) $\Rightarrow$ Verwandt (X, Z) (nichtlinear)
Mann (X) $\wedge$ $\neg$Ehepaar (X, Y) $\Rightarrow$ Ledig (X) (mit Negation)
Frau (Y) $\wedge$ $\neg$Ehepaar (X, Y) $\Rightarrow$ Ledig (Y) (mit Negation)

Beispiele für Anfragen

Ledig (X)
Cousine (Clyde, X)
SelbeGeneration (Clyde, X)

Bemerkung:
Im Gegensatz zu Prolog ist Datalog wirklich nichtprozedural (z.B. ist die Reihenfolge der Regeln irrelevant) und soll große, persistente Sammlungen von Fakten und Regeln verarbeiten.

Ergänzende Literatur zu Kapitel 6:

P. Kandzia, H.-J. Klein, Theoretische Grundlagen relationaler Datenbanksysteme, BI-Verlag, Reihe Informatik, Band 79, 1993
S. Abiteboul, R. Hull, V. Vianu, Foundation of Databases, Addison-Wesley, 1995
P.C. Kanellakis, Elements of Relational Database Theory, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1991
J.D. Ullman, Principles of Database and Knowledge-Based Systems Vol. 1 and Vol.2, Computer Science Press, 1989
S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer-Verlag, 1990
A.B. Cremers, U. Griefahn, R. Hinze, Deduktive Datenbanken, Vieweg-Verlag, 1994
Uwe Schöning: Logik für Informatiker, Spektrum Akademischer Verlag, 1995

Kapitel 7: Die Datenbanksprache SQL

SQL (Structured Query Language) ist die Standardsprache für die Datendefinition und Datenmanipulation in relationalen Datenbanksystemen. Sie umfaßt:

Interaktives ("stand-alone") SQL (z.B. Oracle SQL*Plus)
Eingebettetes ("embedded") SQL (für die gängigen Programmiersprachen)
Anweisungen zur Integritätssicherung und Zugriffskontrolle
Anweisungen zur physischen Datenorganisation (Indizes etc.)

Zusätzlich in vielen Produkten:

Integration von SQL in Skriptsprachen für sog. Stored Procedures (z.B. Oracle PL/SQL)

Zur Historie von SQL:

Entwicklung der Sprache Sequel (Structured English Query Language) in den Siebziger Jahren am IBM Almaden Research Lab sowie der Prototypimplementierung System R.
Erste kommerzielle Produkte seit Anfang der Achtziger Jahre (SQL/DS, Oracle, usw.).
In Form des ANSI- und ISO-Standards SQL-92 (SQL2) produktunabhängig definiert (mit den Stufen „Entry Level", „Intermediate Level" und „Full Level").
Von allen marktrelevanten Datenbanksystemen unterstützt (Oracle, IBM DB2, Informix, Sybase, MS SQL Server) sowie in stark eingeschränktem Umfang auch von dem via GNU-Lizenz kostenfrei verfügbaren System MySQL.
Mit dem neueren ANSI-Standard SQL-99 (SQL3) um mächtige objekt-orientierte bzw. objekt-relationale Features erweitert (die allerdings – soweit sie über den sog. „Core" hinausgehen – in Produkten uneinheitlich unterstützt werden).
Ein weiteres Standard-Release SQL-2003, in dem auch XML-Unterstützung vorgesehen ist, ist in Vorbereitung.

Kapitel 7 orientiert sich primär an SQL-92. Kapitel 10 geht auf die objekt-relationalen Erweiterungen von SQL-99 ein.

7.1 Einfache Datendefinition

Exkurs: Syntaxbeschreibung

Die Syntax einer formalen Sprache wird durch eine **kontextfreie Grammatik** beschrieben, ein 4-Tupel $G = (\Sigma, V, P, S)$ bestehend aus:

- einer endlichen Menge Σ von Terminalsymbolen (Literalen), den Zeichen (Symbolen) der definierten Sprache,
- einer endlichen Menge V von Nonterminalsymbolen (Variablen) (mit $V \cap \Sigma = \varnothing$) zur Definition der grammatikalischen Struktur,
- einer endlichen Menge $P \subseteq V \times (V \cup \Sigma)^*$ von Produktionsregeln mit
  - einem Nonterminalsymbol als linker Seite und einem
  - String aus Terminal- und Nichtterminalsymbolen auf der rechten Seite,
- einem ausgezeichneten Nonterminalsymbol $S \in V$ als Startsymbol.

Die von G erzeugte Sprache $L(G) \subseteq \Sigma^*$ ist die Menge aller aus Terminalsymbolen gebildeten Wörter (Sätze) $w \in \Sigma^*$, so daß es eine endliche Folge von Phrasen x0, ..., xn gibt mit

xi $\in$ (V$\cup\Sigma$)* für alle i, x0=S, xn=w, so daß xi aus x(i-1) durch Anwendung einer Produktionsregel r aus P entsteht (wobei in x(i-1) ein Vorkommen der linken Seite von r durch die rechte Seite von r ersetzt werden).

Einfaches Beispiel:
V={Address, Name, Firstname, Lastname, Street, City, Country, String, Letter, Number, Digit},
Σ={a, b, c, ..., 0, 1, …}, S=Address,
P={Address $\rightarrow$ Name Street City, Address $\rightarrow$ Name Street City Country,
Name $\rightarrow$ Firstname Lastname, Name $\rightarrow$ Letter Lastname,
Firstname $\rightarrow$ String, Lastname $\rightarrow$ String,
Street $\rightarrow$ String Number, Country $\rightarrow$ String,
String $\rightarrow$ Letter, String $\rightarrow$ Letter String, Number $\rightarrow$ Digit Number, Number $\rightarrow$ ε,
Letter $\rightarrow$ a, Letter $\rightarrow$ b, …, Digit $\rightarrow$ 0, Digit $\rightarrow$ 1, …}

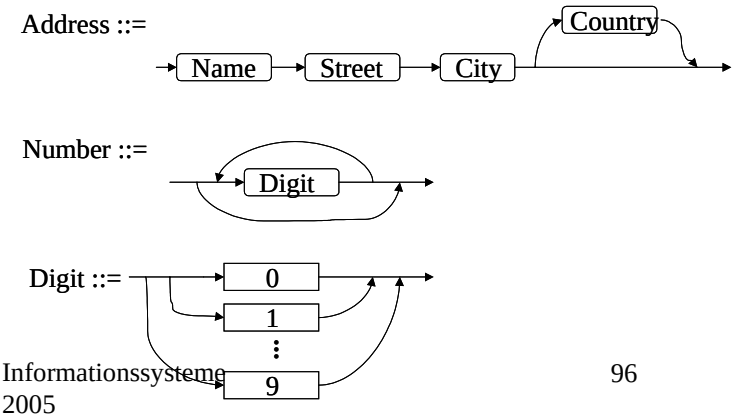
Eine kompaktere "Makroschreibweise" für kontextfreie Grammatiken (insbesondere für rekursive Produktionsregeln) ist die **EBNF (Extended Backus-Naur-Form)** in Verbindung mit notationellen Konventionen:

Nichtterminalsymbole werden klein, Terminalsymbole groß geschrieben (und können ggf. Sonderzeichen beinhalten).
Als Metazeichen für die Regeln selbst dienen
::= zur Trennung von linker und rechter Seite
| zur Trennung von Alternativen für die rechte Seite einer (Gruppe von) Regel(n)
[ ] für optionale Phrasen (die also keinmal oder einmal auf der rechten Seite stehen dürfen)
{ ...} für wiederholbare Phrasen (die keinmal, einmal oder beliebig oft vorkommen dürfen)
{ } zur Klammerung, um die Struktur eindeutig zu machen.
Semantische Optionen, die standardmäßig gesetzte sind (Default-Werte), auch wenn sie syntaktisch gar nicht erzeugt werden, werden unterstrichen.

Beispiel:
Address ::= Name Street City [Country], String ::= Letter {Letter …},
Number ::= {Digit …}, Digit ::= 0|1|2|3|4|5|6|7|8|9, usw.

EBNF-Regeln werden häufig auch in Form von **Syntaxdiagrammen** visualisiert. Dabei werden Nichtterminalsymbole durch Ovale und Terminalsymbole durch Rechtecke dargestellt. Die Nichtterminal- und Terminalsymbole der rechten Seite einer (Gruppe von) Regel(n) sind so durch gerichtete Kanten verbunden, daß jede aus der rechten Seite konstruierbare Ableitung einem Pfad dieses Minigraphen von einer Quelle zu einer Senke entspricht. Rekursive Regeln bzw. wiederholbare Phrasen führen also auf Schleifen in diesem Minigraphen.

Beispiel:



"Grobsyntax" der SQL-Anweisung CREATE TABLE (in EBNF)
(unter Vernachlässigung relativ unwichtiger Features)

```
CREATE TABLE [user .] table (  column_element {, column_element …}
                                        {, table_constraint …} )
```

mit

```
column_element ::=      column data_type [DEFAULT expr] [column_constraint]
column_constraint ::=   [NOT NULL]
                        [PRIMARY KEY | UNIQUE]
                        [REFERENCES [user .] table [ ( column ) ] ]
                        [CHECK ( condition ) ]
table_constraint ::=    [ {PRIMARY KEY | UNIQUE} ( column {, column …} ) ]
                        [ FOREIGN KEY ( column {, column …} )
                           REFERENCES [user .] table [ ( column {, column …} ) ]
                        [CHECK ( condition ) ]
```

Beispiele:

```
CREATE TABLE Kunden (  KNr INTEGER CHECK (KNr > 0) PRIMARY KEY,
   Name VARCHAR(30), Stadt VARCHAR(30),
   Saldo FLOAT,
   Rabatt FLOAT CHECK (Rabatt >= 0.0) )
CREATE TABLE Produkte (PNr INTEGER CHECK (PNr > 0) PRIMARY KEY,
   Bez VARCHAR(30) NOT NULL UNIQUE, Gewicht FLOAT CHECK (Gewicht > 0.0),
   Preis FLOAT CHECK (Preis > 0.0),
   Lagerort VARCHAR(30), Vorrat INTEGER CHECK (Vorrat >= 0) )
 CREATE TABLE Bestellungen (
   BestNr INTEGER CHECK (BestNr > 0) PRIMARY KEY,
   Monat INTEGER NOT NULL CHECK (Monat BETWEEN 1 AND 12),
   Tag INTEGER NOT NULL CHECK (Tag BETWEEN 1 AND 31),
   /*** oder besser: Datum DATE, ***/
   KNr INTEGER CHECK(KNr > 0), PNr INTEGER CHECK (PNr > 0),
   Menge INTEGER CHECK (Menge > 0),
   Summe FLOAT, Status VARCHAR(9) CHECK (Status IN ('neu', 'geliefert', 'bezahlt')),
   FOREIGN KEY (PNr) REFERENCES Produkte (PNr),
   FOREIGN KEY (KNr) REFERENCES Kunden (KNr),
   UNIQUE (Monat, Tag, PNr, KNr) )
```

Semantik von CREATE TABLE:
Es wird ein Relationenschema definiert (ggf. mit zusätzlichen Integritätsbedingungen).

Unterschiede zum "reinen" Relationenmodell:

- Die Festlegung eines Primärschlüssels wird nicht unbedingt erzwungen. Wenn aber ein Primärschlüssel spezifiziert ist, dann wird die Einhaltung der Primärschlüsselbedingung garantiert. Analoges gilt für die Fremdschlüsselbedingung.
- Falls kein Primärschlüssel (und kein Schlüsselkandidat) spezifiziert ist, dürfen Tabellen Duplikate enthalten, d.h. es handelt sich um Multimengen.
- SQL unterstützt eine Vielfalt elementarer Datentypen wie z.B. DATE, Varianten von Strings (VARCHAR etc.) Zahlen (NUMBER etc.) und Binärobjekte (RAW, BLOB, etc.)

Zur Semantik der gemäß column_constraint und table_constraint spezifizierbaren Integritätsbedingungen siehe Kapitel 7.

7.2 Einfache Anfragen (Queries)

"Grobsyntax" von SQL-Anfragen:

select_block { { UNION | INTERSECT | EXCEPT } [ALL] select_block ...}
[ORDER BY result_column [ASC | DESC] {, result_column [ASC | DESC] …}
mit
select_block ::= SELECT [ALL | DISTINCT] {column | {expression [AS result_column]}}
{, {column | {expression [AS result_column]}} …}
FROM table [correlation_var] {, table [correlation_var] …}
[WHERE search_condition]
[GROUP BY column {, column …} [HAVING search_condition]]

Ein erster Vergleich mit der Relationenalgebra und dem Tupel-Relationenkalkül:

SELECT A, B, ... FROM R, S, ..., T, ... WHERE F
(so daß A, B, ... zu R, S, ... gehören, nicht aber zu T, ..., und F über R, S, ..., T, ... definiert ist)
enspricht in der Relationenalgebra
$\pi[A, B, ...] (\sigma [F] (R \times S \times ... \times T \times ...)$
und im Tupel-Relationenkalkül
$\{x.A, y.B, ... \mid x \in R \wedge y \in S \wedge ... \wedge \exists z: z \in T ... \wedge F(x, y, ..., z, ...)\}$

Achtung:

- Alle Joinbedingungen müssen in SQL explizit in der WHERE-Klausel (oder in SQL-99 alternativ in der FROM-Klausel) angegeben werden.
- Standardmäßig sind Anfrageresultate Multirelationen. Es erfolgt also keine Duplikateliminierung; diese kann mit DISTINCT explizit spezifiziert werden.
- Bei der Vereinigung (UNION), dem Durchschnitt (INTERSECT) und der Differenz (EXCEPT) von Teilergebnissen gibt es zwei Varianten: für Multimengen (mit ALL) und für Mengen.

Beispiele:

1) Finden Sie (die Namen) alle(r) Kunden mit negativem Saldo.
   → SELECT KNr, Name, Stadt, Saldo, Rabatt FROM Kunden WHERE Saldo < 0.0
   oder: SELECT * FROM Kunden WHERE Saldo < 0.0
   bzw.: SELECT Name FROM Kunden WHERE Saldo < 0.0

2) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
   → SELECT Name FROM Kunden, Bestellungen
   WHEREMonat < 10 AND Status <> 'bezahlt'
   AND Kunden.KNr = Bestellung.KNr

3) Finden Sie die Namen der Homburger Kunden, die seit Anfang September ein Produkt aus Homburg geliefert bekommen haben, jeweils mit der Bezeichnung des entsprechenden Produkts.

   → SELECT Name FROM Kunden, Bestellungen, Produkte
   WHEREStadt='Homburg'
   AND Monat>9 AND Status<>'neu'
   AND Lagerort='Homburg'
   AND Kunden.KNr=Bestellungen.KNr AND Bestellungen.PNr=Produkt.PNr

4) Finden Sie die Rechnungssume der Bestellung mit BestNr 111 (ohne auf das Attribut Summe der Relation Bestellungen zuzugreifen).
   → SELECT Menge*Preis*(1.0-Rabatt) FROM Bestellungen, Produkte, Kunden
   WHERE BestNr=111
   AND Bestellungen.PNr=Produkte.PNr AND Bestellungen.KNr=Kunden.KNr
   oder:
   SELECT Menge*Preis*(1.0-Rabatt) AS Rechnungssumme FROM ... WHERE ...

Allgemeinere Form der FROM-Klausel: Korrelationsvariable (Tupelvariable)

a) Eindeutige Benennung mehrerer "Inkarnationen" derselben Relation (in θ-Joins)

Beispiel:
Finde alle Paare von Kunden, die in derselben Stadt wohnen.
→ SELECT K1.Name, K2.Name FROM Kunden K1, Kunden K2
WHERE K1.Stadt=K2.Stadt AND K1.KNr < K2.KNr

b) Outer Join

Beispiel:
Gib alle Kunden mit 5 % Rabatt zusammen mit ihren Bestellungen und den bestellten Produkten aus, und zwar auch wenn für einen Kunden gar keine Bestellungen vorliegen.
→ SELECT *
FROM Kunden FULL OUTER JOIN Bestellungen
ON (Kunden.KNr=Bestellungen.KNr),
Produkte
WHERE Rabatt = 0.05
AND Produkte.PNr=Bestellungen.PNr

Analog mit LEFT OUTER JOIN und RIGHT OUTER JOIN.

Allgemeinere Form der WHERE-Klausel: Komplexere Suchprädikate

Achtung: SQL hat (zu) viele Alternativen für dieselbe Anfrage !

Vereinfachte Formulierung von Bereichsanfragen ("range queries")

Beispiel:
Finden Sie die Kunden, deren Rabatt zwischen 10 und 20 Prozent liegt.
→ SELECT * FROM Kunden WHERE Rabatt BETWEEN 0.10 AND 0.20

Pattern-Matching in Zeichenketten

mit % als Platzhalter ("wild card") für eine beliebige Zeichenkette der Länge ≥ 0
und _ als Platzhalter für ein beliebiges Zeichen

Beispiele:

1) Finden Sie alle Kunden, deren Name mit A beginnt.
   → SELECT * FROM Kunden WHERE Name LIKE 'A%'
2) Finden Sie alle Kunden mit Namen Meier, Maier, Meyer, ...
   → SELECT * FROM Kunden WHERE Name LIKE 'M_ _er'

weitergehende Möglichkeiten in einigen Produkten (z.B. Ingres)

Beispiel 2:

→ SELECT * FROM Kunden WHERE Name LIKE 'M[a,e][i,y]er'

Test auf Nullwert

Beispiel:

Finden Sie alle Produkte, die grundsätzlich in keinem Lager geführt werden.
→ SELECT * FROM Produkte WHERE Lagerort IS NULL

IS NULL ist das einzige Vergleichsprädikat, das von einem Nullwert erfüllt wird.

Konsequenz: Die Anfrage
SELECT * FROM Produkte WHERE Vorrat >= 100 OR Vorrat < 100
liefert (nur) alle Tupel der Relation Produkte, deren Wert bzgl. des Attributs Vorrat kein Nullwert ist.

Test auf Mitgliedschaft eines Werts in einer Menge (IN-Prädikat)

Einfaches Beispiel:

Finden Sie alle Kunden aus Homburg, Merzig und Saarlouis.
→ SELECT * FROM Kunden WHERE Stadt IN ('Homburg', 'Merzig', 'Saarlouis')

Allgemeine Form mit *"Subqueries"* (geschachtelten Select-Blöcken):
{column | expression} [NOT] IN ({ value {, value ...} | select_block })

Beispiele:

1) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.
→ SELECT Name FROM Kunden
WHERE KNr IN (SELECT KNr FROM Bestellungen
WHERE Status<>'bezahlt' AND Monat < 10)

2) Finden Sie alle Kunden aus Städten, in denen es mindestens zwei Kunden gibt.
→ SELECT * FROM Kunden K1
WHERE Stadt IN (SELECT Stadt FROM Kunden K2
WHERE K2.KNr <> K1.KNr)
Die Subquery wird in diesem Fall auch als "korrelierte Subquery" bezeichnet.

Achtung:
Es handelt sich hierbei um Tests, ob ein Wert in einer Menge von Werten enthalten ist.
Ein Test, ob ein Tupel (mit mindestens zwei Attributen) in einer Menge von Tupeln enthalten ist, ist auf diese Weise nicht möglich.
Teilmengentests zwischen Mengen sind in SQL ebenfalls nicht möglich.

Beispiel:

Finden Sie die Kundennummern der Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.

Ansatz:

```
SELECT KNr FROM Kunden
WHERE    (  SELECT PNr FROM Produkte )
            IN
            (  SELECT PNr FROM Bestellungen
               WHERE KNr = Kunden.KNr )
```

ist keine korrekte SQL-Anweisung!
(IN wäre hier im Sinne von $\subseteq$ gemeint, nicht - wie zuvor - im Sinne von $\in$.)

"Quantifizierte" Vergleiche zwischen einem Wert und einer Menge

Die Bedingung *Wert* θ*ANY Menge* mit $\theta \in \{\{=, \neq, <, >, \leq, \geq\}$ ist erfüllt,
wenn es in der Menge ein Element gibt, für das *Wert* θ *Element* gilt.
(=ANY ist äquivalent zu IN.)

Die Bedingung *Wert* θ*ALL Menge* mit $\theta \in \{=, \neq, <, >, \leq, \geq\}$ ist erfüllt,
wenn für alle Elemente der Menge gilt: *Wert* θ *Element*.
(<>ALL ist äquivalent zu NOT IN.)

Beispiel: Finden Sie die Kunden mit dem geringsten Rabatt.

```
→ SELECT * FROM Kunden
    WHERE Rabatt <=ALL ( SELECT Rabatt FROM Kunden)
```

Achtung:
Das Prädikat *Wert =ALL Menge* ist nur bei einer einelementigen Menge erfüllbar.
Die Anfrage

```
SELECT KNr FROM Bestellungen
WHERE PNr =ALL (SELECT PNr FROM Produkte)
```

kann nur dann ein nichtleeres Resultat liefern, wenn es nur ein einziges Produkt (oder gar keines) gibt!

Test, ob eine Menge leer oder nichtleer ist (Existenztest)

Beispiele:

1) Finden Sie die Namen aller Kunden, die eine unbezahlte Bestellung haben, die vor Anfang Oktober erfolgte.

```
→ SELECT Name FROM Kunden
  WHERE EXISTS ( SELECT * FROM Bestellungen
                 WHERE Status <> 'bezahlt' AND Monat < 10
                 AND Bestellungen.KNr = Kunden.KNr )
```

2) Finden Sie die Kunden, für die keine Bestellung registriert ist.

```
→ SELECT * FROM Kunden
  WHERE NOT EXISTS ( SELECT * FROM Bestellungen
                     WHERE Bestellungen.KNr = Kunden.KNr )
```

Suchbedingungen der Form "... für alle ..." (Verwendung des Allquantors im Relationenkalkül bzw. der Division in der Relationalgebra) können in SQL mit NOT EXISTS "emuliert" werden.

Beispiel:

Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.

```
→   SELECT * FROM Kunden
    WHERE   NOT EXISTS
            ( SELECT * FROM Produkte
              WHERE   NOT EXISTS
                      ( SELECT * FROM Bestellungen
                        WHERE Bestellungen.PNr = Produkte.PNr
                        AND Bestellungen.KNr = Kunden.KNr ) )
```

7.3 Semantik einfacher SQL-Anfragen

Abbildung von SQL auf den Tupelrelationenkalkül (TRK)

unter Vernachlässigung von Multimengen, Nullwerten u.ä.
und der Annahme der eindeutigen Benennung von Tupelvariablen und eindeutigen Zuordnung von Attributen zu Tupelvariablen

Beispiel: SELECT A FROM REL WHERE EXISTS (SELECT B FROM REL WHERE B>0)
würde ggf. umbenannt in
SELECT R1.A FROM REL R1 WHERE EXISTS (SELECT R2.B FROM REL R2 WHERE R2.B>0)

Definition einer Abbildungsfunktion
sql2trc: sql query → trc query
von syntaktischen Konstruktionen der Form select_block auf sichere TRK-Anfragen
unter Verwendung der Funktion
sql2trc': sql where clause → trc formula
von syntaktischen Konstruktionen der Form search_condition auf TRK-Formeln.

sql2trc [SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F]
(so daß A1, A2, ..., An zu REL1, REL2, …, RELm gehören, nicht aber zu TAB1, …, TABk und F über REL1, ..., RELm, TAB1, …, TABk definiert ist)
= $\{ri_1.A1, ri_2.A2, ... \mid r1 \in REL1 \wedge r2 \in REL2 \wedge \ ... \wedge rm \in RELm \wedge$
$\exists t1 ... \exists tk (t1 \in TAB1 \wedge ... \wedge tk \in TABk \wedge sql2trc'[F]) \}$

sql2trc [select-block1 UNION select-block2]
(mit select-block1: SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F
und select-block2: SELECT B1, B2, ... FROM SET1 S1, SET2 S2, ..., SETm' Sm',
PAR1 P1, ..., PARk' Pk' WHERE G)
= $\{u1, u2, ... \mid (\exists r1 ... \exists rm (u1 = r1.A1 \wedge \ u2 = r2.A2 \wedge \ldots \wedge$
$r1 \in REL1 \wedge r2 \in REL2 \wedge \ ... \wedge rm \in RELm \wedge$
$\exists t1 ... \exists tk \ (t1 \in TAB1 \wedge ... \wedge tk \in TABk \wedge sql2trc'[F] \ ) \ )$
$\vee \quad (\exists s1 ... \exists sm' (u1 = s1.B1 \wedge \ u2 = s2.B2 \wedge \ldots \wedge$
$s1 \in SET1 \wedge s2 \in SET2 \wedge \ ... \wedge sm' \in SETm' \wedge$
$\exists p1 ... \exists pk' (p1 \in PAR1 \wedge ... \wedge pk' \in PARk' \wedge sql2trc'[G] \ ) \ ) \}$

sql2trc [select-block1 INTERSECT select-block2] und
sql2trc [select-block1 EXCEPT select-block2]:
analog

sql2trc' [Ri.Aj θ Tk.Bl] = ri.Aj θ tk.Bl
sql2trc' [Ri.Aj θ c] (mit einer Konstanten c) = ri.Aj θ c
sql2trc' [F AND G] = sql2trc'[F] ∧ sql2trc'[G]
sql2trc' [F OR G] = sql2trc'[F] ∨ sql2trc'[G]
sql2trc' [NOT F] = ¬ sql2trc'[F]

sql2trc' [Ri.Aj IN subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists q1 \ldots \exists qm' (qk.C = ri.Aj \wedge q1 \in QUELL1 \wedge \ldots qm' \in QUELLm' \wedge sql2trc'[H])$

sql2trc' [Ri.Aj θANY subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists qk (qk \in QUELLk \wedge (\exists q1 \ldots \exists q(k-1) \ \exists q(k+1) \ldots \exists qm'$
$(ri.Aj \ \theta \ qk.C \ \wedge q1 \in QUELL1 \wedge \ldots \wedge q(k-1) \in QUELL(k-1) \wedge$
$q(k+1) \in QUELL(k+1) \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])))$

sql2trc' [Ri.Aj θALL subquery]
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \forall qk ((qk \in QUELLk \wedge \ (\exists q1 \ldots \exists q(k-1) \ \exists q(k+1) \ldots \exists qm'$
$(q1 \in QUELL1 \wedge \ldots \wedge q(k-1) \in QUELL(k-1) \wedge$
$q(k+1) \in QUELL(k+1) \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])))$
$\Rightarrow (ri.Aj \ \theta \ qk.C))$

sql2trc' [EXISTS subquery]
(so daß subquery die Form SELECT C1, C2, ...
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
$= \exists q1 \ldots \exists qm' \ (q1 \in QUELL1 \wedge \ldots \wedge qm' \in QUELLm' \wedge \ sql2trc'[H])$

Beispiel:

query = SELECT K.KNr, K.Name FROM Kunden K
WHERE K.Ort='Saarbrücken'
AND NOT EXISTS (SELECT P.PNr FROM Produkte P, Bestellungen B
WHERE P.Preis > 100 AND P.PNr=B.PNr AND B.KNr=K.KNR)

sql2trc [query] = {k.KNr, k.Name | $k \in$ Kunden $\wedge$ sql2trc'[K.Ort=... AND NOT EXISTS …]}
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge \neg$ sql2trc'[NOT EXISTS ...]}
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge$
$\neg (\exists p \exists b (p \in$ Produkte $\wedge b \in$ Bestellungen $\wedge$
sql2trc'[P.Preis>... AND ... AND ...])) }
= {k.KNr, k.Name | $k \in$ Kunden $\wedge$ k.Ort=... $\wedge$
$\neg (\exists p \exists b (p \in$ Produkte $\wedge b \in$ Bestellungen $\wedge$
p.Preis>... $\wedge$ p.PNr=b.PNr $\wedge$ b.KNr=k.KNr)) }

Der Kern der einfachen SQL-Anfragen ist also komplett in den sicheren TRK übersetzbar und hat damit ein präzise definierte Semantik. Umgekehrt sind auch alle möglichen sicheren TRK- (oder RA-) Anfragen in SQL ausdrückbar. Es gilt also:

Satz:

SQL ist relational vollständig.

Abbildung von SQL auf die Relationenalgebra (RA)

unter Vernachlässigung von Multimengen, Nullwerten u.ä.
und der Annahme der eindeutigen Benennung von Tupelvariablen und eindeutigen Zuordnung von Attributen zu Tupelvariablen (wie bei der Abbildung auf den sicheren TRK)

Definition einer Abbildungsfunktion
sql2ra: sql query → ra query
von syntaktischen Konstruktionen der Form select_block auf RA-Anfragen
unter Verwendung der Funktion
sql2ra': sql where clause × ra query → ra query
von syntaktischen Konstruktionen der Form search_condition auf RA-Ausdrücke
sowie der Hilfsfunktion
sql2ra-: sql where clause × ra query → ra query
mit sql2ra- [F](E) = E - π[sch(E)] (sql2ra'[F](E))

sql2ra [SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F]
(so daß A1, A2, ..., An zu REL1, REL2, …, RELm gehören, nicht aber zu TAB1, …, TABk und F über REL1, ..., RELm, TAB1, …, TABk definiert ist)
= R1 := REL1; …; Rm := RELm; T1 := TAB1; ...; Tk := TABk;
π[Ri_1.A1, Ri_2.A2, ...] (sql2ra'[F](R1 × … × Rm × T1 × … × Tk))

sql2ra [select-block1 UNION select-block2]
(mit select-block1: SELECT A1, A2, ... FROM REL1 R1, REL2 R2, ..., RELm Rm,
TAB1 T1, ..., TABk Tk WHERE F
und select-block2: SELECT B1, B2, ... FROM SET1 S1, SET2 S2, ..., SETm' Sm',
PAR1 P1, ..., PARk' Pk' WHERE G)
= sql2ra[select-block1] ∪ sql2ra[select-block2]
(mit ggf. notwendigen Umbenennungen von Attributen)

sql2ra [select-block1 INTERSECT select-block2] und
sql2ra [select-block1 EXCEPT select-block2]:
analog

sql2ra' [Ri.Aj θ Tk.Bl] (E) = σ[Ri.Aj θ Tk.Bl](E)
sql2ra' [Ri.Aj θ c] (E) (mit einer Konstanten c) = σ[Ri.Aj θ c](E)
sql2ra' [F AND G] (E) = sql2ra'[F](E) ∩ sql2tra'[G](E)
sql2ra' [F OR G] (E) = sql2ra'[F](E) ∪ sql2ra'[G](E)
sql2ra' [NOT F] (E) = sql2ra-[F](E) = E - π[sch(E)] (sql2ra'[F](E))

sql2ra' [Ri.Aj IN subquery](E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (σ[Ri.Aj = Qk.C] (E × Q1 × … × Qm')))

sql2ra' [Ri.Aj θANY subquery](E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (σ[Ri.Aj θ Qk.C] (E × Q1 × … × Qm')))

sql2ra' [Ri.Aj θALL subquery] (E)
(so daß subquery die Form SELECT Qk.C
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= sql2ra- [Ri.Aj θ'ANY subquery](E)
mit θ' gleich ≠ für θ gleich =, = für ≠, < für ≥, > für ≤, usw.
= E - π[sch(E)] (sql2ra'[H] (σ[Ri.Aj θ' Qk.C] (E × Q1 × … × Qm')))

sql2ra' [EXISTS subquery] (E)
(so daß subquery die Form SELECT C1, C2, ...
FROM QUELL1 Q1, ..., QUELLm' Qm' WHERE H hat)
= Q1 := QUELL1; … Qm' := QUELLm';
π[sch(E)] (sql2ra'[H] (E × Q1 × … × Qm'))

Beispiel:

query = SELECT K.KNr, K.Name FROM Kunden K
WHERE K.Ort='Saarbrücken'
AND NOT EXISTS (SELECT P.PNr FROM Produkte P, Bestellungen B
WHERE P.Preis > 100 AND P.PNr=B.PNr AND B.KNr=K.KNR)

sql2ra [query] = K := Kunden; B := Bestellungen; P := Produkte;
π[K.KNr, K.Name] (sql2ra'[K.Ort=… AND NOT EXISTS …] (K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩ sql2ra'[NOT EXISTS …](K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩
sql2ra- [EXISTS ...](K))
= … π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](sql2ra' [EXISTS ...](K))))
= ... π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](sql2ra'[P.Preis...AND…AND…](K × P × B)))
= ... π[K.KNr, K.Name] (σ[Ort=…](K) ∩
(K - π[sch(K)](σ[P.Preis>100](K × P × B) ∩
σ[P.PNr=B.PNr](K × P × B) ∩
σ[P.PNr=B.PNr](K × P × B))))

Für den Umgang mit Multimengen könnte man eine Übersetzung in die erweiterte Relationenalgebra in analoger Weise angeben. Beispielsweise würde sql2ra [SELECT DISTINCT A FROM ... WHERE ...] übersetzt werden in χ (sql2ra [SELECT A FROM ... WHERE ...).

7.4 Erweiterte SQL-Anfragen

Aggregationsfunktionen

Zweck: Abbildung einer Menge oder Multimenge von skalaren Werten auf einen Wert.

"Grobsyntax":
{ MAX | MIN | AVG | SUM | COUNT } ({ ALL | DISTINCT } {column | expression | *})

Beispiele:

1) Finden Sie den höchsten Rabatt aller Kunden.
   → SELECT MAX (Rabatt) FROM Kunden
   Finden Sie den durchschnittlichen Rabatt aller Kunden.
   → SELECT AVG (Rabatt) FROM Kunden

2) An wievielen Lagerorten werden Produkte gelagert?
   → SELECT COUNT (DISTINCT Lagerort) FROM Produkte

3) Wieviele Kunden haben einen Rabatt von mehr als 15 Prozent?
   → SELECT COUNT (*) FROM Kunden WHERE Rabatt > 0.15

4) Welche Kunden haben einen überdurchschnittlichen Rabatt?
   → SELECT * FROM Kunden
   WHERE Rabatt > SELECT AVG (Rabatt) FROM Kunden

5) Wie hoch ist der Gesamtumsatz?
   → SELECT SUM (Menge*Preis*(1.0-Rabatt))
   FROM Bestellungen, Produkte, Kunden
   WHERE Bestellungen.PNr=Produkte.PNr AND Bestellungen.KNr=Kunden.KNr

Einfache "Built-in"-Funktionen

Zweck: Transformation von skalaren Werten
produktspezifisch, z.B. in Oracle:

SELECT SUBSTR (Name, INSTR(Name, ' ')+1) FROM Kunden
zur Extraktion der Nachnamen (unter der Annahme, daß Name den Vornamen und Nachnamen durch Leerzeichen getrennt enthält)
SELECT TO_CHAR(SYSDATE, 'DY DD MONTH YYYY, HH24:MI:SS')
zur Konvertierung des aktuellen Tagesdatums (Datentyp DATE) in einen String

Ideal: Erweiterbarkeit des Datenbanksystems (siehe Kapitel 8)

Oracle-spezifische Erweiterung zur Textsuche (Oracle interMedia)

Die CONTAINS-Funktion liefert für einen Attributwert und eine Textsuchbedingung einen numerischen Score-Wert (z.B. die Anzahl der Treffer oder eine Ähnlichkeitsangabe)

Beispiele:

1) SELECT PNr, Bez, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Internet', 1) > 0 bzw. >= 5
   ORDER BY SCORE(1) DESC

   liefert alle Produkte, deren Beschreibung das Wort 'Internet' enthält
   bzw. mindestens fünfmal enthält, in einer nach Relevanz absteigend sortierten Rangliste

2) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, 'Oracle AND (Internet OR Web) MINUS SQL') > 0

   liefert alle Produkte, deren Beschreibung das Wort 'Oracle' und mindestens eines der Wörter 'Internet' oder 'Web', nicht jedoch das Wort 'SQL' enthält

3) SELECT PNr, Bez, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, '$Query', 1) > 0 ORDER BY SCORE(1) DESC

   liefert alle Produkte, deren Beschreibung ein Wort mit dem Wortstamm 'Query' enthält, also z.B. auch ”Querying" oder ”Queries"

4) SELECT PNr, Bez, SCORE(1) FROM Produkte
   WHERE CONTAINS (Beschreibung, '?Oracle', 1) > 0 ORDER BY SCORE(1) DESC

   liefert alle Produkte, deren Beschreibung ein Wort enthält,
   das dem Wort 'Oracle' lexikalisch ähnlich ist (wie z.B. 'Orakel')

5) SELECT PNr, Bez FROM Produkte
   WHERE CONTAINS (Beschreibung, '!Oracle') > 0 : 10

   liefert alle Produkte, deren Beschreibung ein Wort enthält, das in phonetischer Hinsicht dem Wort 'Oracle' ähnlich ist (z.B. 'Awewreckle'), und zwar nur die (maximal) zehn ähnlichsten Treffer

6) SELECT PNr, Bez, SCORE(1)*SCORE(2) AS TOTALSCORE FROM Produkte
   WHERE CONTAINS (Beschreibung, 'SYN(database system)', 1) > 0
   AND CONTAINS (Beschreibung, 'NT(search)', 2) > 0 ORDER BY SCORE(1)*SCORE(2) DESC

   mit Nachschlagen in einem (ggf. benutzerspezifischen) Thesaurus (Wörterbuch, Ontologie) zur Berücksichtigung von Synonymen (SYN) und Unterbegriffen (NT)

Aggregation mit Gruppenbildung

Zweck:
Eine Tupelmenge wird aufgrund der Werte eines Attributs oder einer Attributkombination oder eines Ausdrucks in "Gruppen" (Äquivalenzklassen) partitioniert.

Beispiel:
Bestimmen Sie für alle Produkte deren Gesamtverkaufszahl (ab Anfang September).
→ SELECT PNr, SUM(Menge) FROM Bestellungen WHERE Monat >= 9
GROUP BY PNr

Zwischenresultat nach der Gruppierung:

PNr	**Gruppe**		
	BestNr	**...**	**Menge**
1	9		100
2	3		4
3	4		1
4	5		10
5	6		50
	10		50
	11		50
6	7		2
7	8		5
	12		10

Endresultat:

PNr	**SUM(Menge)**
1	100
2	4
3	1
4	10
5	150
6	2
7	15

Achtung:
In der SELECT-Klausel (d.h. der Projektionsliste) dürfen nur Resultate von Aggregationsfunktionen stehen oder Attribute, die auch Gruppierungsattribute sind (also in der GROUP-Klausel vorkommen).

Beispiel mit Auswahl von Gruppen:

Bestimmen Sie alle Kunden mit mindestens zwei Bestellungen seit Anfang Oktober, deren Gesamtwert mindestens 2000 DM betragen hat.

→ SELECT KNr FROM Bestellungen
WHERE Monat >= 10
GROUP BY KNr
HAVING COUNT(*) >= 2 AND SUM(Summe) >= 2000.00

Zwischenresultat nach der Gruppierung:

KNr	**Gruppe**		
	BestNr	**...**	**Summe**
1	6		900.00
	7		180.00
	8		900.00
2	9		1600.00
	10		800.00
3	7		1800.00

nach der Aggregation:

KNr	**COUNT(*)**	**SUM(Summe)**
1	3	1980.00
2	2	2400.00
3	1	1800.00

Endresultat:

KNr
2

weitere Beispiele:

1) Bestimmen Sie für alle Produkte, die mehr als 1000-mal verkauft worden sind, deren Gesamtverkaufszahl (ab Anfang September).
   → SELECT PNr, SUM(Menge) FROM Bestellungen WHERE Monat >= 9
   GROUP BY PNr HAVING SUM(Menge) > 1000

2) Bestimmen Sie für jede Stadt mit mehr als 10 Kunden den durchschnittlichen Rabatt dieser Kunden. Die Ausgabe soll nach Rabatten absteigend sortiert sein.
   → SELECT Stadt, AVG(Rabatt) AS MittlRabatt FROM Kunden
   GROUP BY Stadt HAVING COUNT(*) > 10
   ORDER BY 2 DESC bzw. ORDER BY MittlRabatt DESC

3) Finden Sie die Kunden, die alle überhaupt lieferbaren Produkte irgendwann bestellt haben.
   → SELECT KNr FROM Bestellungen
   GROUP BY KNr
   HAVING COUNT (DISTINCT PNr) =

(SELECT COUNT(*) FROM Produkte)

Semantik der Gruppierung / Aggregation von SQL: Abbildung auf Relationenalgebra-Programme

Gegeben sei eine Anfrage der Form
SELECT A', f(B) FROM … WHERE … GROUP BY A
mit Aggregationsfunktion f und Attributmengen A, A' mit $A' \subseteq A$
(falls $A' \subseteq A$ nicht gilt, hat die Anfrage keine wohldefinierte Semantik und wird vom SQL-Compiler als fehlerhaft abgewiesen).
Die Semantik dieser Anfrage ist durch Übersetzung in die erweiterte Relationenalgebra wie folgt definiert:
sql2ra [SELECT A', f(B) FROM … WHERE … GROUP BY A]
= R(A, F) := γ_+[A,B,f] (sql2ra[SELECT A,B FROM … WHERE …]); π_+[A', F] (R)
wobei sql2ra auf Multirelationen zu erweitern ist.

Gegeben sei eine Anfrage der Form
SELECT A', f(B)
FROM ...
WHERE ...
GROUP BY A
HAVING cond(A, g(C))
mit Aggregationsfunktionen f, g und einer Suchbedingung cond über A und g(C) (die auch wieder Subqueries beinhalten darf, vom äußeren Block aber nur auf Attribute aus A oder Aggregationsfunktionen g(C) Bezug nehmen darf). Der Einfachheit halber sei nur der Fall ohne DISTINCT betrachtet.

Sei Q(A,B,C) das Ergebnis der WHERE-Klausel-Auswertung.
Dann ergibt sich das Ergebnis der Gesamtanfrage durch:

R (A,F) := $\varnothing$
for each x $\in$ π [A](Q) do
  G_X := π_+[A,B,C] (σ_+[A=x] (Q))
  if sql2ra'[cond(A,g(C))](G_X) $\neq \varnothing$ then
    y := f (π_+[B](G_X))
    R := R $\cup_+$ {(x,y)}
  fi
od;
π_+[A', F] (R)

Rekursive Anfragen in SQL-99 bzw. Oracle:

Der neuere Standard SQL-99 sieht die Unterstützung linearer Rekursion vor, wobei akkumulierende Berechnungen in die Rekursion eingebettet werden können.

Beispiel:

```
WITH RECURSIVE Verbindungen (Start, Ziel, Gesamtdistanz) AS
   ( ( SELECT Abflugort, Zielort, Distanz
       FROM Flüge
       WHERE Abflugort = 'Frankfurt' )
     UNION
     ( SELECT V.Start, F. Ziel, V.Gesamtdistanz + F.Distanz
       FROM Verbindungen V, Flüge F
       WHERE V.Ziel = F.Abflugort )
   )
SELECT Ziel, Gesamtdistanz
FROM Verbindungen
```

In Oracle ist Rekursion ebenfalls in eingeschränkter Form möglich, und zwar im wesentlichen für die Berechnung transitiver Hüllen und mit einer Ad-hoc-Erweiterung der SQL-Syntax.

Beispiel:

```
SELECT F.Zielort
FROM Flüge F
START WITH Abflugort = 'Frankfurt'
CONNECT BY Abflugort = PRIOR Zielort
AND PRIOR Ankunftszeit < Abflugzeit - 0.05
```

7.5 Datenmodifikation

Einfügen von Tupeln

"Grobsyntax":
INSERT INTO table [(column {, column …})]
{ VALUES (expression {, expression …}) | subselect }

Beispiele:
1) INSERT INTO Kunden VALUES (7, 'Kunz', 'Neunkirchen', 0.0, 0.0)
2) INSERT INTO Kunden (KNr, Name, Stadt) VALUES (7, 'Kunz', 'Neunkirchen')
3) CREATE TABLE Mahnungen (BestNr ...) mit demselben Schema wie Bestellungen
   INSERT INTO Mahnungen
      SELECT * FROM Bestellungen WHERE Status='geliefert' AND Monat<10

Attributwerte, die beim Einfügen nicht spezifiziert sind, werden auf den Default-Wert oder den Nullwert gesetzt.

Ändern von Tupeln

"Grobsyntax":
UPDATE table [correlation_var]
SET column = expression {, column = expression …} WHERE search_condition

Beispiele:
1) UPDATE Kunden SET Stadt = 'Saarbrücken' WHERE KNr=1
2) UPDATE Kunden SET Rabatt = Rabatt + 0.05
   WHERE Saldo > -10000.0
   AND KNr IN SELECT KNr FROM Bestellungen
   GROUP BY KNr HAVING SUM(Summe) > 100000.0

Löschen von Tupeln

"Grobsyntax":
DELETE FROM table [correlation_var] [WHERE search_condition]

Beispiel: DELETE FROM Bestellungen WHERE Monat < 7

Schemaänderung

"Grobsyntax":
ALTER TABLE table
ADD column datatype [column_constraint] {, column data_type [column_constraint] …}

Beispiel: ALTER TABLE Bestellungen ADD Frist INTEGER CHECK (Frist > 0)

Existierende Tupel werden bezüglich neuer Attribute implizit auf den ggf. spezifizierten Default-Wert oder den Nullwert gesetzt.

Transaktionen

Es ist häufig notwendig, mehrere SQL-Anweisungen zu einer atomaren Einheit zu klammern, die nur als Ganzes einen konsistenten Datenbankzustand in einen anderen konsistenten Zustand überführen (insbesondere ist dies für logisch gekoppelte Änderungen mehrerer Tabellen unumgänglich). Diese Klammerung erfolgt durch Abschluß einer Folge von SQL-Anweisungen mit der Anweisung COMMIT WORK bzw. ROLLBACK WORK, wobei letzteres alle Änderung der Transaktionen rückgängig macht. Mit dem Abschluß einer Transaktion (und initial beim Verbindungsaufbau mit dem Datenbanksystem) wird implizit eine neue Transaktion geöffnet. In manchen Programmierumgebungen gibt es einen Autocommit-Modus, bei dem automatisch jede einzelne SQL-Anweisung eine Transaktion bildet, indem implizit nach jeder Anweisung ein "Commit Work" durchgeführt wird.

Beispiel einer Transaktion: Bearbeitung einer neuen Lieferung

```
UPDATE Produkte SET Vorrat = Vorrat - 10 WHERE PNr = 4711;
UPDATE Bestellungen SET Status = 'geliefert' WHERE BestNr = 333444555;
COMMIT WORK;
```

Transaktionen werden ausführlich in späteren Kapiteln behandelt.

Ergänzende Literatur zu Kapitel 7:

J. Melton, A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

C.J. Date, H. Darwen, A Guide to the SQL Standard, Addison-Wesley, 1997

P. Gulutzan, T. Pelzer, SQL-99 Complete, Really, R&D Books, 1999

C. Türker, SQL:1999 & SQL:2003, dpunkt-Verlag, 2003

Oracle9i SQL Reference,
http://tahiti.oracle.com/pls/db92/db92.show_toc?partno=a96540

Oracle9i SQL*Plus User's Guide and Reference
http://tahiti.oracle.com/pls/db92/db92.show_toc?partno=a90842

M. Negri, S. Pelagatti, L. Sbattella, Formal Semantics of SQL Queries, ACM Transactions on Database Systems, Vol. 16 No. 3, Sept. 1991

M. Gogolla, An Extended Entity-Relationship Model, Chapter 5: Formal Semantics of SQL, Springer-Verlag, Lecture Notes in Computer Science 767, 1994

Kapitel 8: SQL-Einbettung in Programmiersprachen

Kopplungsvarianten zwischen Programmiersprachen und Datenbanksprachen:

1) Erweiterung der Programmiersprache um Datenbankkonstrukte
   → Persistente Programmiersprachen, Datenbankprogrammiersprachen
   (z.B. Pascal/R, DBPL, Persistentes C++)
2) Erweiterung der Datenbanksprache um Programmierkonstrukte
   → Skriptsprachen für Stored Procedures, "4th Generation Languages" (4GLs)
3) Einbettung der Datenbanksprache in die Programmiersprache (oder Web-Skriptsprache)
   → "Embedded SQL" (ESQL)

Beispiel für eine Datenbankprogrammiersprache (à la DBPL im Pascal- bzw. Modula-Stil):

```
TYPE    Produktrecordtyp        = RECORD
                                    PNR: INTEGER;
                                    ...
                                  END;
        Produkterelationentyp   = RELATION OF Produktrecordtyp;

VAR     Produkte: PERSISTENT Produkterelationentyp;
        Ergebnis: Produkterelationentyp;
        x: Produktrecordtyp;

Ergebnis :=  SELECT *
             FROM Produkte
             WHERE ...;

FOR EACH x IN Ergebnis
DO
   ...
END;
```

8.1 Eingebettetes SQL (Embedded SQL)

Kernproblem bei der SQL-Einbettung in konventionelle Programmiersprachen:

Abbildung von Tupelmengen auf die Datentypen der Wirtsprogrammiersprache

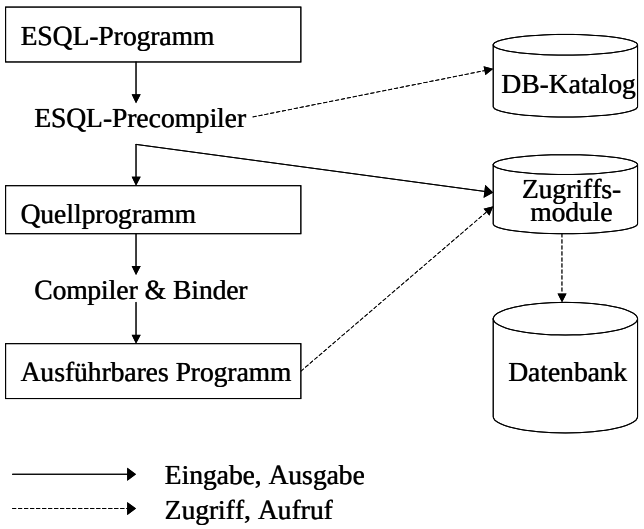
Realisierte Lösung:

Abbildung von Tupeln bzw. Attributen auf die Datentypen der Programmiersprache
→ Wirtsprogrammvariable (engl.: Host Variables)

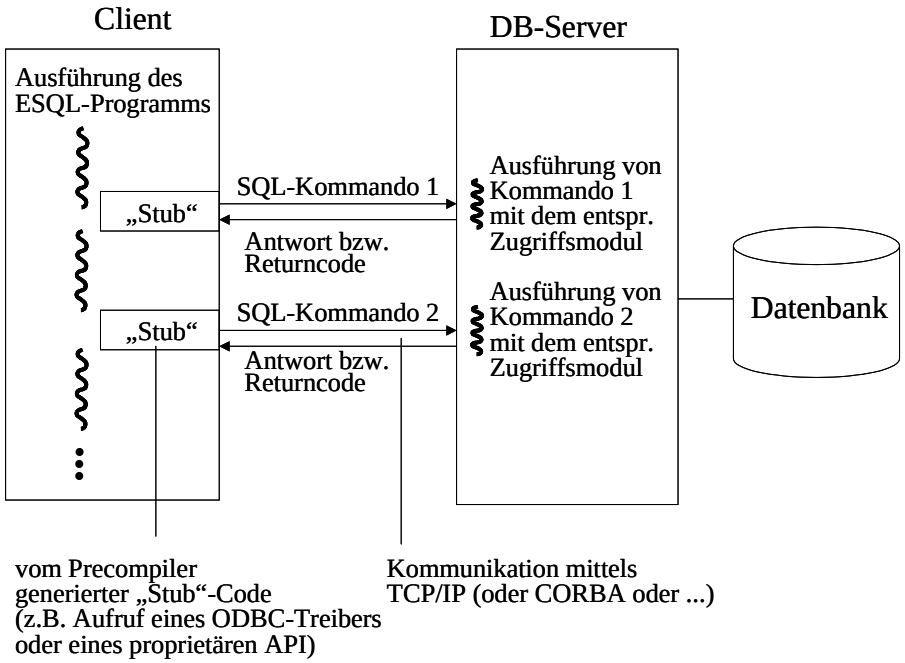
+

Iteratoren (Schleifen) zur Verarbeitung von Tupelmengen
→ Cursor-Konzept

Standardarchitektur:



Laufzeitarchitektur für ESQL-Programme in einem Client-Server-System:



Wirtsprogrammvariable (Host Variables)

Die Attribute eines Resultattupels einer SQL-Anfrage werden an speziell deklarierte Variable des Wirtsprogramms zugewiesen (INTO-Klausel).
Beispiel:

```
EXEC SQL    SELECT PNr, Menge, Status INTO :pnr, :menge, :status
            FROM Bestellungen WHERE BestNr = 555;
```

Analog können SQL-Anweisungen mittels solcher Variable mit Eingabeparametern versorgt werden.
Beispiele:

```
1) EXEC SQL    SELECT PNr, Menge, Status INTO :pnr, :menge, :status
               FROM Bestellungen WHERE BestNr = :bestnr;
2) EXEC SQL    INSERT INTO Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge)
               VALUES (:bestnr, :monat, :tag, :knr, :pnr, :menge);
```

Wirtsprogrammvariable dienen als "Übergabepuffer" zwischen DBS und Programm. Generell werden dabei die SQL-Datentypen auf die Datentypen der jeweiligen Wirtsprogrammiersprache abgebildet.

Indikatorvariable zum Erkennen von Nullwerten

Beispiel:

```
EXEC SQL BEGIN DECLARE SECTION;
   int      pnr;
   int      vorrat;
   short    vorrat_ind;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL    SELECT Vorrat INTO :vorrat:vorrat_ind
            FROM Produkte WHERE PNr = :pnr;
if (vorrat_ind = = 0)
   { /* kein Nullwert */ ... }
else
   { /* Nullwert */ ... };
```

Allgemein können Indikatorvariable folgende Werte haben:

= 0	die entsprechende Wirtsprogrammvariable hat einen regulären Wert
= -1	die entsprechende Wirtsprogrammvariable hat einen Nullwert
> 0	die entsprechende Wirtsprogrammvariable enthält eine abgeschnittene Zeichenkette

Beispiel eines ESQL-Programms (lieferung.pc)

```
/*** Programm zur Erfassung von Lieferungen ***/

#include <stdio.h>
#include <string.h>

EXEC SQL INCLUDE SQLCA; /* Importieren der SQL Communication Area */
/*  Innerhalb der SQL-Anweisungen ist Gross-/Kleinschreibung irrelevant.
    Sie dient hier nur zur Hervorhebung der SQL-Anweisungen          */

/* Der Precompiler erzeugt an dieser Stelle die folgende Datenstruktur:
    struct sqlca {
        char    sqlcaid[8];
        long    sqlabc;
        long    sqlcode;
        struct  {
                unsigned short  sqlerrml;
                char            sqlerrmc[70];
                } sqlerrm;
        char    sqlerrp[8];
        long    sqlerrd[6];
        char    sqlwarn[8];
        char    sqlext[8];
        };
    struct sqlca sqlca;
*/

main ()
{
/* Deklaration der Wirtsprogrammvariablen */
EXEC SQL BEGIN DECLARE SECTION;
    /*  Die hier deklarierten Wirtsprogrammvariablen unterliegen bei ihrer
        Verwendung in SQL-Ausdrücken einer Typprüfung durch den Precompiler. */
    int bestnr;
    int pnr;
    int menge;
    VARCHAR status[10];

    /* Der Precompiler erzeugt aus einer solchen Deklaration die folgende Struktur:
        struct {
                unsigned short len;
                unsigned char arr[10];
                } status;
    */

    VARCHAR user[20];
    VARCHAR passwd[10];
EXEC SQL END DECLARE SECTION;

/* Globale Fehlerbehandlung */
EXEC SQL WHENEVER SQLERROR STOP;
```

```
/* Verbindungsaufbau mit dem DBS */
printf ("Benutzername?"); scanf ("%s", &(user.arr));
user.len = strlen(user.arr); /* Konvertierung von C-String in VARCHAR */
printf ("Kennwort?"); scanf ("%s", &(passwd.arr));
passwd.len = strlen(passwd.arr); /* Konvertierung von C-String in VARCHAR */
EXEC SQL CONNECT :user IDENTIFIED BY :passwd;
/* gleichzeitig Beginn einer Transaktion */

/* Dialogschleife */
printf ("Bitte geben Sie eine Bestellnummer ein. (0 = Programmende)\n");
scanf ("%d", &bestnr);

while (bestnr != 0)
   {
   EXEC SQL    SELECT PNr, Menge, Status
               INTO :pnr, :menge, :status
               FROM Bestellungen
               WHERE BestNr = :bestnr;
   if (sqlca.sqlcode = = 0) /* Testen des SQL-Returncodes */
      {
      status.arr[status.len] = '\0'; /* Konvertierung von VARCHAR in C-String */
      if (strcmp (status.arr, "neu") = = 0)
         {
         EXEC SQL    UPDATE Produkte
                     SET Vorrat = Vorrat - :menge
                     WHERE PNr = :pnr AND Vorrat >= :menge;
         if (sqlca.sqlcode = = 0) /* Testen des SQL-Returncodes */
            {
            strcpy (status.arr, "geliefert");
            status.len = strlen (status.arr);
            EXEC SQL    UPDATE Bestellungen
                        SET Status = :status
                        WHERE BestNr = :bestnr;
            }
         else
            printf ("\n *** Ungenuegender Lagervorrat ***\n");
         }
      else
         printf ("\n *** Lieferung bereits erfolgt ***\n");
      }
   else
      printf ("\n *** Bestellung nicht gefunden ***\n");
   EXEC SQL COMMIT WORK;  /* Ende Transaktion und Beginn neue Transaktion */
   printf ("Bitte geben Sie eine Bestellnummer ein. (0 = Programmende)\n");
   scanf ("%d", &bestnr);
   }; /*while*/

EXEC SQL DISCONNECT;    /* Verbindung mit dem DBS abbauen */
exit (0);
} /*main*/
```

Fehlerbehandlung in ESQL

1) Explizites Testen von sqlca.sqlcode im Wirtsprogramm nach einer SQL-Anweisung:

= 0 → Anweisung korrekt ausgeführt, keine besonderen Vorkommnisse
< 0 → Fehler bei der Ausführung (siehe Fehlercodes im Manual)
> 0 → Anweisung ausgeführt, Auftreten eines Sonderfalls,
z.B. signalisiert der Wert 1403, daß keine (weiteren) Treffertupel existieren

Zusatzinformation in den restlichen Komponenten von sqlca, z.B.:

sqlca.sqlerrd[2]	Anzahl der Tupel, die von einer Insert-, Update- oder Delete-Anweisung betroffen waren
sqlca.sqlerrm.sqlerrmc	Fehlermeldung als Ascii-Text (max. 70 Zeichen)
sqlca.sqlerrm.sqlerrml	Länge der Fehlermeldung

2) Deklaration von "Exception-Handling"-Strategien:

```
EXEC SQL WHENEVER    ( SQLERROR | NOT FOUND | SQLWARNING )
                        ( STOP | GOTO label | CONTINUE )
```

wobei
SQLERROR einem SQLCODE < 0 entspricht,
NOT FOUND dem SQLCODE 1403 und
SQLWARNING einem SQLCODE > 0 (aber ungleich 1403).

Der ESQL-Precompiler erzeugt automatisch nach jeder SQL-Anweisung einen entsprechenden Vergleich mit sqlca.sqlcode, und zwar jeweils aufgrund der *textuell* letzten WHENEVER-Anweisung vor der SQL-Anweisung. Extrem schwerwiegende Fehler führen immer zum sofortigen Abbruch.

Achtung: Die WHENEVER-Klausel hat potentielle Fallen.

1) Im folgenden Programm ist in f die WHENEVER-Spezifikation von g nicht wirksam.

```
void f (...) ...
   { ...  EXEC SQL UPDATE ...; ... };
void g (..) ...
   { ...  EXEC SQL WHENEVER SQLERROR GOTO handle_error; ...
          f (...); ...
   };
```

2) Das folgende Programm führt u.U. zu einer Endlosschleife.

```
...
EXEC SQL WHENEVER SQLERROR GOTO handle_error;
EXEC SQL CREATE TABLE Mytable ( ... );
...
handle_error:
   EXEC SQL DROP TABLE Mytable;
   EXEC SQL DISCONNECT;
   exit (1);
```

Korrektur:
Im Teil hinter der Marke "handle_error" als erstes spezifizieren:
EXEC SQL WHENEVER SQLERROR CONTINUE;

Verarbeitung von Tupelmengen mittels Cursor-Konzept

Zweck:
Verarbeitung von Tupelmengen in einer nichtmengenorientierten Wirtsprogrammiersprache.
Notwendig für alle Anfragen mit mehr als einem Resultattupel.

Beispiel: Ausgabe aller Bestellungen eines Kunden

```
#define TRUE 1
#define FALSE 0
...
printf ("Bitte geben Sie eine Kundennummer ein. (0 = Programmende)\n");
scanf ("%d", &knr);
EXEC SQL DECLARE Kundeniterator CURSOR FOR
    SELECT Monat, Tag, Bez, Menge
    FROM Bestellungen WHERE KNr = :knr
    ORDER BY Monat DESC, Tag DESC;
/* Cursor-Deklaration immer ohne INTO-Klausel */
...
EXEC SQL OPEN Kundeniterator;
/*  An dieser Stelle werden die Eingabeparameter der SQL-Anweisung (:knr) ausgewertet,
    und gedanklich wird hier die Resultattupelmenge ermittelt.                    */
found = TRUE;
while (found) /* solange es noch Resultattupel gibt */
    {
    EXEC SQL FETCH Kundeniterator INTO :monat, :tag, :bez, :menge;
    /* Hier werden die Wirtsprogrammvariable für die Resultattupel festgelegt. */
    bez.arr[bez.len] = '\0';
    if ((sqlca.sqlcode >= 0) && (sqlca.sqlcode != 1403))
        printf ("%d.%d.: %d %s", tag, monat, menge, bez);
    else
        found = FALSE;
    }; /*while*/
EXEC SQL CLOSE Kundeniterator;
/* Freigabe des Cursors und der damit verbundenen DBS-Ressourcen */
```

8.2 Dynamisches ESQL

Zweck:

Einbettung von SQL-Anweisungen, deren Struktur erst zur Laufzeit des Wirtsprogramms bekannt ist.

Bei Einbettung von SQL in die (auf Bytecodebene interpretierte) Programmiersprache Java mit JDBC als Datenbankzugriffsprotokoll (einer Erweiterung von ODBC) ist dynamisches SQL der Regelfall. Die Vorübersetzung von SQL-Anweisungen im Java-Programm ist im Standard SQLJ vorgesehen, bislang aber wenig verbreitet.

Einfacher Fall: Vorgehensweise bei statisch festgelegtem Resultatschema und statisch festgelegten Eingabeparametern:

Beispiel ("dynamischer Anteil" ist unterstrichen):

```
SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge
FROM Bestellungen, Produkte WHERE Bestellungen.PNr = Produkte.PNR
AND ( Bez LIKE 'Druck%' OR Bez LIKE 'Papier%' OR Bez LIKE 'Platte%' )
ORDER BY Monat DESC, Tag DESC
```

Vorgehensweise:

1) Deklaration der Wirtsprogrammvariablen
2) Aufbau der SQL-Anweisung als Zeichenkette
3) Dynamische Precompilation: PREPARE DynQuery FROM :sqltext
4) DECLARE Iterator CURSOR FOR DynQuery
5) Auswertung der Eingabeparameter und Vorbereitung der "Cursor"-Schleife:
   OPEN Iterator
6) FETCH Iterator INTO Wirtsprogrammvariablen
7) Resultatsattribute in den Wirtsprogrammvariablen verarbeiten
8) Wiederholung der Schritte 6 und 7
   für jedes Resultatstupel
9) CLOSE Iterator

Sonderfall: Anfrageresult hat höchstens ein Tupel
(z.B. bei Anfragen über den Primärschlüssel oder bei Änderungsoperationen)
statt 3) bis 9) einfacher:
EXECUTE IMMEDIATE :sqltext

Komplexer Fall: Vorgehensweise bei dynamisch festgelegtem Resultatschema oder dynamisch festgelegten Eingabeparametern (mit dynamischer Allokation der Datenübergabebereiche und dynamischer Übersetzung für wiederholte Ausführung):

Abfrage der Typbeschreibung mit DESCRIBE queryname INTO ptr-to-sqldescriptorarea,
dynamische Allokation der Übergabebereiche im Programm und
Eintragen entsprechender Pointer in die sqldescriptorarea (SQLDA) vor der Queryausführung.
(siehe Handbücher; für Vorlesung nicht relevant)

Beispiel für den einfachen Fall:

```
/***  Programm zum Ausdrucken der Bestellungen fuer eine Liste von Produkten ***/
/***  Eingabe: Liste von Produktbezeichnungen                                ***/
/***  Ausgabe: Bestellungen dieser Produkte, absteigend sortiert nach Datum  ***/

#include <stdio.h>
#include <string.h>
#define TRUE 1
#define FALSE 0
#define MAXPRODUKTE 10
/* Die Bestellungen umfassen maximal 10 Produkte */
EXEC SQL INCLUDE SQLCA; /* Importieren der SQLCA */

main () {
EXEC SQL BEGIN DECLARE SECTION;
    int monat, tag, pnr, menge;
    VARCHAR bez[31];
    VARCHAR sqltext[512];
    VARCHAR user[20];
    VARCHAR passwd[10];
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR GOTO handle_error;

/* Typdeklarationen */
typedef char prod_bez_t[31];
typedef int bool;

/* Variablendeklarationen */
prod_bez_t     prod_bez[10];
int            i;
int            num_of_prod;
bool           found;

printf ("Benutzername?"); scanf ("%s", &(user.arr)); user.len = strlen(user.arr);
printf ("Kennwort?"); scanf ("%s", &(passwd.arr)); passwd.len = strlen(passwd.arr);
EXEC SQL CONNECT :user IDENTIFIED BY :passwd;

/* Eingabe */
i = 0;
printf ("%s%s", "Bitte geben Sie eine Produktbezeichnung oder \"ende\" ein.\n");
scanf ("%s", &(prod_bez[i]));
while ((i < MAXPRODUKTE) && (strcmp (prod_bez[i], "ende") != 0))
    {
    i++;
    printf ("%s%s", "Bitte geben Sie eine Produktbezeichnung oder \"ende\" ein.\n");
    scanf ("%s", &(prod_bez[i]));
    }; /*while*/
num_of_prod = i;
if (num_of_prod = = 0) exit(-1);
```

```
/* Aufbau der SQL-Anfrage */
strcpy (sqltext.arr, "SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge ");
strcat (sqltext.arr, "FROM Bestellungen, Produkte ");
strcat (sqltext.arr, "WHERE Bestellungen.PNr = Produkte.PNr ");
strcat (sqltext.arr, "AND ( ");
strcat (sqltext.arr, "Bez LIKE \'");
strcat (sqltext.arr, prod_bez[0]);
strcat (sqltext.arr, "%\' ");
for (i=1; i < num_of_prod; i++) {
    strcat (sqltext.arr, "OR Bez LIKE \'");
    strcat (sqltext.arr, prod_bez[i]);
    strcat (sqltext.arr, "%\' ");
    }; /*for*/
strcat (sqltext.arr, ") ");
strcat (sqltext.arr, "ORDER BY Monat DESC, Tag DESC");
sqltext.len = strlen(sqltext.arr);

/*  Nur zu Testzwecken */
printf ("\nDie generierte SQL-Anfrage lautet:\n"); printf ("%s\n\n", sqltext.arr);
/*  zum Beispiel:
    SELECT Monat, Tag, Bestellungen.PNr, Bez, Menge
    FROM Bestellungen, Produkte WHERE Bestellungen.PNr = Produkte.PNR
    AND ( Bez LIKE 'Druck%' OR Bez LIKE 'Papier%' OR Bez LIKE 'Platte%' )
    ORDER BY Monat DESC, Tag DESC                                          */

EXEC SQL PREPARE DynQuery FROM :sqltext;
EXEC SQL DECLARE BestIterator CURSOR FOR DynQuery;
EXEC SQL OPEN BestIterator;
found = TRUE;
while (found)
    {
    EXEC SQL FETCH BestIterator INTO :monat, :tag, :pnr, :bez, :menge;
    if (sqlca.sqlcode = = 0) {
        bez.arr[bez.len] = '\0';
        /* Ausgabe */
        printf ("%d.%d.: %d Stueck %s (PNr %d)\n", tag, monat, menge, bez.arr, pnr);
        }
    else
        found = FALSE;
    }; /*while*/
EXEC SQL CLOSE BestIterator;
EXEC SQL COMMIT WORK RELEASE;
exit (0);

/* Fehlerbehandlung */
handle_error:
    printf ("\n*** Error %d in program. ***\n", sqlca.sqlcode);
    printf ("%.70s\n", sqlca.sqlerrm.sqlerrmc);
    exit (-1);
} /*main*/
```

Beispiele für Anwendungen, die dynamisches ESQL erfordern:

- SQL*Plus und ähnliche universelle, interaktive DBS-Schnittstellen sind Programme, die dynamisches ESQL verwenden.

- Anwendungen, die sowohl Daten als auch Metadaten (Relationen des DB-Katalogs) verarbeiten, benötigen in vielen Fällen dynamisches ESQL.
  Solche Anwendungen sind mit dynamischem ESQL realisierbar, weil in praktisch allen relationalen DBS der DB-Katalog in Form von Relationen zugreifbar ist.

  Beispiel (Achtung: in SQL so nicht möglich !):
  SELECT * FROM * WHERE * LIKE '%Lafontaine%'

Wichtige Katalogrelationen in Oracle:

SYSCATALOG bzw. SYS.ALL_TABLES UNION SYS.ALL_VIEWS

OWNER	TABLE_NAME	TABLE_TYPE
DBS	BESTELLUNGEN	TABLE
DBS	KUNDEN	TABLE
DBS	PRODUKTE	TABLE
SYS	ALL_TABLES	VIEW
.		
.		
.		

SYS.ALL_TAB_COLUMNS

OWNER	TABLE_NAME	COLUMN_ NAME	DATA_ TYPE	DATA_ LENGTH	...	COLUMN_ ID	...
DBS	KUNDEN	KNR	NUMBER	22		1	
DBS	KUNDEN	NAME	CHAR	30		2	
DBS	KUNDEN	STADT	CHAR	30		3	
DBS	KUNDEN	SALDO	NUMBER	22		4	
DBS	KUNDEN	RABATT	NUMBER	22		5	
DBS	PRODUKTE	PNR	NUMBER	22		1	
.							
.							
.							
SYS	ALL_TABLES	OWNER	CHAR	30		1	
SYS	ALL_TABLES	TABLE_NAME	CHAR	30		2	
.							
.							
.							

Stored Procedures: Einbettung von SQL in eine 4GL
(Beispiel: PL/SQL von Oracle)

Die Prinzipien sind wie bei ESQL. Syntaktisch ergeben sich einige Vereinfachungen. Andererseits sind 4GLs häufig gegenüber Standardprogrammiersprachen wie (z.B. C) in ihren Datentypen und sonstigen Ausdrucksmitteln beschränkt.

Grobsyntax einer Prozedurdeklaration:

```
proc-decl =    CREATE PROCEDURE proc-name
               [ ”(" param-name type { ”," param-name type } ”)" ] AS proc-body
proc-body =    decl-part block
block =  BEGIN statement-list [ exception-part ] END
```

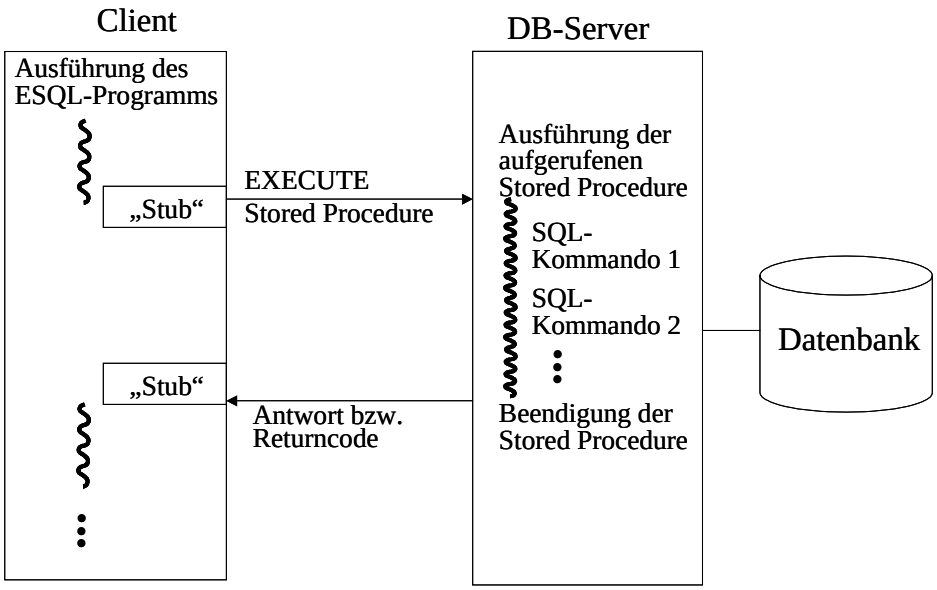
Beispiel:

```
CREATE PROCEDURE Lager_Auffuellen AS
DECLARE ...;
BEGIN
   DECLARE CURSOR Knappe_Produkte IS
      SELECT PNr, Vorrat
      FROM Produkte
      WHERE Vorrat < 100;
   KP Knappe_Produkte%ROWTYPE;
   BEGIN
      OPEN Knappe_Produkte;
      LOOP
         FETCH Knappe_Produkte INTO KP;
         EXIT WHEN Knappe_Produkte%NOTFOUND;
         ...
         IF KP.Vorrat > 10
         THEN
            EXECUTE Nachbestellen (KP.PNr, 1000 - KP.Vorrat)
         ELSE
            EXECUTE Express_Bestellen (KP.PNr, 10);
            EXECUTE Nachbestellen (KP.PNr, 1000 - KP.Vorrat - 10);
         END IF;
      END LOOP;
   END;
END;
```

Aufruf der PL/SQL Stored Procedure direkt von SQL*PLus, SQL*Forms oder ESQL-Programmen möglich,
z.B.: EXECUTE Lager_Auffuellen

Mehrere Prozeduren und Funktionen können zu einem Modul - in Oracle-Terminologie einem *Package* - zusammengefaßt und unter einem global bekannten Namen den Anwendungsentwicklern zur Verfügung gestellt werden.

Laufzeitarchitektur für Stored Procedures:



8.3 JDBC

Von Java-Programmen aus warden SQL-Datenbanken mit dem JDBC-Protokoll (Java Database Connectivity) angesprochen. Dies ist eine Java-orientierte konkrete Syntax für dynamisches ESQL (siehe 8.2). Details zu JDBC können in Handbüchern bzw. Dokumentations-Webseiten nachgelesen werden. Das folgende Beispielprogramm verdeutlicht die Programmierkonventionen von JDBC; es bezieht sich auf eine Datenbank mit den folgenden Relationen:

```
-- Kunden-Tabelle:
create table customer
   (name varchar(20) primary key, ipnumber varchar(20), city varchar(20));
-- Konto-Tabelle:
create table account (name varchar(20) primary key, balance integer);
-- Kunden-Tabelle:
create table history (name varchar(20), bookingdate date, amount integer);
```

Beispielprogramm

```
import java.io.*; import java.util.*; import java.sql.*;
public class myjdbc {
public static void main (String args[]) {

Connection con = null; Statement stmt = null;
String sqlstring;

String driver = "oracle.jdbc.driver.OracleDriver";
try { Class.forName(driver);}
  catch(Exception e) {
  System.out.println ("error when loading jdbc driver");};

try {
String jdbcURL = "jdbc:oracle:thin:";
String db = "(DESCRIPTION =(ADDRESS_LIST = (ADDRESS = " +
  "(PROTOCOL = TCP)(HOST = TOKYO)(PORT = 1521)))" +
  "(CONNECT_DATA =(SERVICE_NAME = TOKYO.WORLD)))";
String user = "lehre40"; String password = "leere4zig";
```

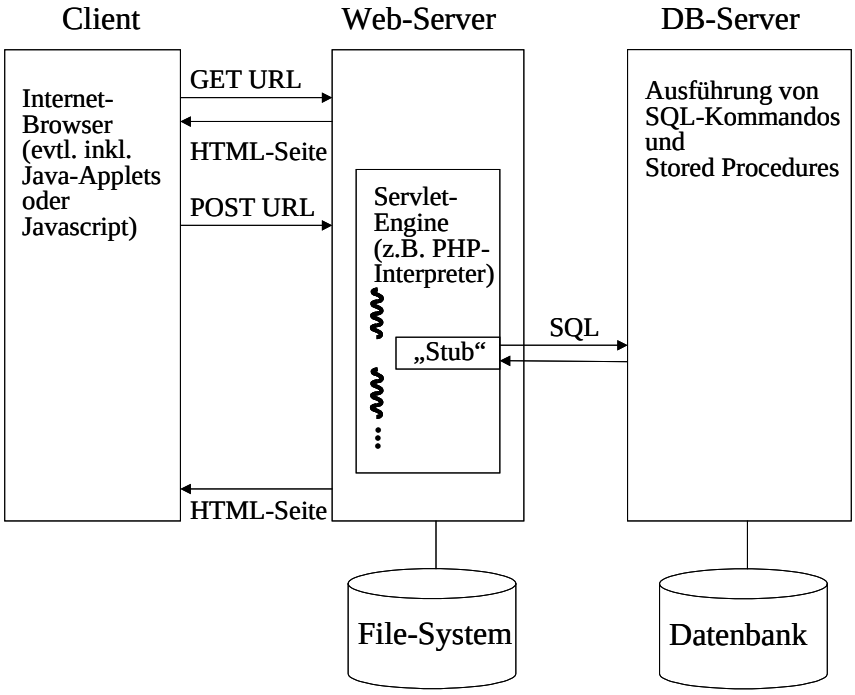
```
con = DriverManager.getConnection
     (jdbcURL + "@" + db, user, password);
con.setAutoCommit (false);
stmt = con.createStatement(); }
  catch (Exception e) {
  System.out.println ("error with jdbc connection"); };

String inputline = null;
try{
System.out.println ("\n Please type name. \n");
DataInputStream myinput = new DataInputStream (System.in);
inputline = myinput.readLine(); }
catch (IOException e) {System.out.println ("IO error");};

try{
sqlstring = "SELECT * FROM CUSTOMER " +
          "WHERE NAME LIKE '" + inputline + "'";
ResultSet result = stmt.executeQuery (sqlstring);
System.out.println ("CUSTOMER:");
System.out.println ("=========");
while (result.next()) {
  String namevar = result.getString("NAME");
  String ipnumbervar = result.getString("IPNUMBER");
  String cityvar = result.getString("CITY");
  System.out.println ("NAME: " + namevar + " IPNUMBER: "
                + ipnumbervar + " CITY: " + cityvar);
}; //while
con.commit(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};
System.out.println ("\n");
try {
sqlstring = "SELECT * FROM ACCOUNT " +
          "WHERE NAME LIKE '" + inputline +"'";
ResultSet result = stmt.executeQuery (sqlstring);

System.out.println ("ACCOUNT:");
System.out.println ("========");
while (result.next()) {
  String namevar = result.getString("NAME");
  int balancevar = result.getInt("BALANCE");
  System.out.println ("NAME: " + namevar +
                " BALANCE: " + balancevar);
}; //while
con.commit(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};

try{ con.close(); }
  catch (SQLException sqlex)
  {System.out.println (sqlex.getMessage());};

}; //main
} //myjdbc
```

8.4 Web-Anwendungen mit Java Servlets

Web-fähige Datenbankanwendungen basieren in der Regel auf Servlets, relativ kleinen Programmen bzw. Skripts, die unter der Kontrolle eines Web-Servers bzw. einer Servlet-Engine laufen (z.B. Apache/Tomcat). Servlets werden durch eine HTTP-Nachricht (GET oder POST) vom Web-Server aktiviert und generieren - typischerweise mit Hilfe von Datenbankzugriffen - dynamische HTML-Seiten, die der Web-Server an den Internet-Browser des Clients sendet.

Die syntaktisch an C und Perl angelehnte Skriptsprache PHP (PHP Hypertext Preprocessor) ist eine - vermutlich die populärste - Sprache zur Erstellung solcher Servlets. PHP wird direkt in eine HTML-Datei eingebettet, die die Datei-Extension .php haben muß. Wenn auf diese Datei via HTTP GET zugegriffen wird, interpretiert der Web-Server die Datei zunächst als HTML-Seite; er erkennt eingebetteten PHP-Code durch ein Tag <?php ... ?> und schaltet die eigentliche PHP-Engine zur Interpretation des PHP-Codes ein. Die - mittels echo oder printf erzeugte - Ausgabe des PHP-Codes wird einfach an die entsprechenden Stellen der HTML-Seite hineinkopiert, und die so erzeugte dynamische HTML-Seite wird an den Internet-Browser geschickt.

Laufzeitarchitektur für Web-fähige Servlets



Web-Anwendungen, die mit Servlets arbeiten, haben eine 3-Schichten-Architektur (3-Tier Architecture) mit:

- einem Frontend-Client, typischerweise ein Internet-Browser, in dem ggf. Javascript-Code oder ganze Java-Applets laufen können,
- dem Applikations-Server in der Mitte, der Web-Server und Servlet-Engine umfasst, und
- einem Backend-Datenbank-Server, der vom Applikations-Server mit ODBC oder JDBC angesprochen wird.

Gegenüber einer 2-Schichten-Architektur, bei der ein mächtigerer Client direkt mit dem Datenbank-Server verbunden ware, hat die 3-Schichten-Architektur gewichtige Vorteile:

- Sie ist von der Code-Installation viel leichter zu verwalten und zu pflegen.
- Sie ist sicherer, da kein kritischer Code in den anfälligeren Clients läuft.
- Sie ist weitaus besser skalierbar (da sie z.B. weniger Datenbank-Sessions verwendet).

Servlets in Java arbeiten nach demselben Grundprinzip, nur dass der Java-Code nicht direkt in HTML-Seiten eingebettet ist; dazu müsste man sogenannte JSPs (Java Servlet Pages) verwen-

den. Zum einfacheren Umgang mit dem Input des zum Servlet-Start führenden HTTP-Aufrufs und der HTTP-Antwort verwenden Java-Servlets typischerweise die JSDK-Klasse HttpServlet, die sie selbst geeignet erweitern können.

Das folgende Beispiel für ein Java-Servlet arbeitet auf einer Datenbank mit den folgenden Relationen:

```
-- Kunden-Tabelle:
create table customer
   (name varchar(20) primary key, ipnumber varchar(20), city varchar(20));
-- Konto-Tabelle:
create table account (name varchar(20) primary key, balance integer);
-- Kunden-Tabelle:
create table history (name varchar(20), bookingdate date, amount integer);
```

Das Servlet arbeitet nach der folgenden Ablauflogik:

Empfange HTTP-Auftrag (Aufruf der Webseite);
Versuche Kunden aufgrund seiner IP-Nummer in der Datenbank zu finden;
Prüfe, ob der Kunde Parameter im Eingabeformular der Webseite angegeben hat;
if Kunde ist in Datenbank oder Kundendaten im Eingabeformular übergeben
then
Speichere Kundendaten in der Datenbank;
Erzeuge Text in HTTP-Antwort mit perönlicher Begrüßung;
else Sende leeres Eingabeformular;
fi;
Erzeuge vollständige Webseite als HTTP-Antwort mit Links auf Servlets „transfer“ und „audit“;
Sende HTTP-Antwort;

Beispiel-Servlet:

```
import java.lang.*; import java.io.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;
import java.sql.*; import java.net.*;
public class welcome extends HttpServlet {

public void doGet
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String namevar; String cityvar;
BufferedReader readervar; String linestring;
String ipnumbervar = null; String browservar = null;
boolean newcustomer; String nexturl;

PrintWriter out = response.getWriter();
out.println("<html><body>");
namevar = request.getParameter ("inputname");
cityvar = request.getParameter ("inputcity");
ipnumbervar = request.getRemoteAddr();
// create JDBC connection
// ...



// actual application code
newcustomer = true;
try{
```

```
sqlstring = "SELECT * FROM CUSTOMER " +
          "WHERE IPNUMBER='" + ipnumbervar + "'";
ResultSet result = stmt.executeQuery (sqlstring);
if (result.next()) {
   newcustomer = false;
   namevar = result.getString("NAME");
   cityvar = result.getString("CITY");
}; //if
}
catch (SQLException sqlex)
   {System.out.println (sqlex.getMessage());};

if ((namevar != "") & (namevar != null)) {
  // insert customer info into db
  if (newcustomer){
    try {
    sqlstring = "INSERT INTO CUSTOMER " +
        "(NAME,IPNUMBER,CITY) VALUES (' " + namevar +
        "','" + ipnumbervar + "','" + cityvar + "')";
    stmt.execute (sqlstring);
    sqlstring = "INSERT INTO ACCOUNT (NAME,BALANCE) " +
        " VALUES (' " + namevar + "',0)";
    stmt.execute (sqlstring); con.commit(); }
    catch (SQLException sqlex)
      {System.out.println (sqlex.getMessage());};
   } } //then
else {
  // post form for inquiring customer info
  out.println("<form method=post action=welcome>");
  out.println("<br> Please type your name and city. <br>");
  out.println("Name: <input type=text name=inputname><br>");
  out.println("City: <input type=text "  +
            "name=inputcity value=Saarbruecken><br>");
  out.println("<input type=submit name=inputenter " +
            "value=\"Here you go!\"><br>");
  out.println("</form>");
}; //if

if ((namevar != "") && (namevar != null)) {
  out.println ("Welcome, " + namevar + "!<br>");
  out.println ("Today's date is ");
  out.println (new java.util.Date());
  out.println ("<br>");
  out.println ("Your IP number is " + ipnumbervar + "<br>");
  out.println ("How is the weather in " +
             cityvar + "?<br><br>");
  out.println ("How can we help you today?<br>");
  nexturl = "transfer?customername=" +
          URLEncoder.encode(namevar);
  out.println ("<a href=" + nexturl +
             "> Deposit or withdraw money</a><br>");
  nexturl = "audit?customername=" +
          URLEncoder.encode(namevar);
  out.println ("<a href=" + nexturl +
             "> Audit trail of your account</a><br>");
}; //if
out.println("</body></html>");
} //doGet
```

```
public void doPost
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

   doGet(request, response);

}  //  doPost

} // class welcome
```

In vielen Web-Anwendungen müssen ganze Sitzungen mit mehreren Dialogschritten zwischen Client und Applikations-Server verwaltet werden. Die Klasse HttpServlet bietet dazu Session-Objekte an, über die ein Client und der dazugehörige Session-Kontext vom Servlet eindeutig identifiziert werden können, obwohl das Servlet selbst in jedem Dialogschritt „quasi-zustandslos“ aufgerufen wird. Die Implementierung der Session-Objekte verwendet sog. Cookies, die daher im Browser aktiviert sein müssen. Cookies sind kurze Identifikatoren, die vom Web-Server erzeugt werden und zwischen Client und Server als Teil der HTTP-Header ständig hin- und hergeschickt werden. Für den Programmierer des Java-Servlets sind diese Details verborgen; er/sie muss lediglich im Servlet-Code für die Abfrage des Session-Objekts sorgen. Das folgende Beispielprogramm verdeutlicht diese – typische – Vorgehensweise.

Java-Servlet mit Sessions:

```
import java.lang.*; import java.io.*; import java.util.*;
import javax.servlet.*; import javax.servlet.http.*;
public class sessionexample extends HttpServlet {

public void doGet
(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

Integer mycounter;
PrintWriter out = response.getWriter();
out.println("<html><body>");
out.println("<h1>Welcome!</h1><br><br>");


// create session if not yet existing,
// otherwise fetch session data
HttpSession session = request.getSession(true);
if (session.isNew()) {
   mycounter = new Integer(1);
   session.putValue("mycounter", mycounter); }
else {
   mycounter = (Integer) session.getValue("mycounter");
   mycounter = new Integer(mycounter.intValue() + 1);
   out.println("You are visiting us the ");
   out.println(mycounter.intValue());
   out.println("th time");
   session.putValue("mycounter", mycounter);
}; //if
out.println("</body></html>");
} //doGet

} // class sessionexample
```

Ergänzende Literatur zu Kapitel 8:

J. Melton, A. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

Oracle9i Pro*C/C++ Precompiler Programmer's Guide

J.W. Schmidt, F. Matthes, The DBPL Project: Advances in Modular Database Programming, Information Systems Vol. 19 No. 2, 1994

G. Saake, K.-U. Sattler, Datenbanken und Java, dpunkt-Verlag, 2000

J. Hunter, Java Servlet Programming, O'Reilly, 2001

B.W. Perry, Java Servlet & JSP Cookbook, O'Reilly, 2003

A.Harbourne-Thomas et al., Professional Java Servlets 2.3, Wrox Press, 2002

Sun Microsystems, JDBC Technology, http://java.sun.com/products/jdbc/

Oracle9*i* JDBC Developer's Guide and Reference

Oracle9i JDBC Reference

Kapitel 9: Integritätssicherung und Zugriffskontrolle

9.1 Integritätsbedingungen

Ziel:
Die Datenbank soll zu jedem Zeitpunkt die Zusammenhänge und Regeln der realen (Geschäfts-) Welt so akkurat wie möglich widerspiegeln. Daher finden in Informationssystemen vielfältige Plausibilitätsprüfungen statt; die Realisierung dieser Prüfungen kann einen signifikanten Anteil der Anwendungsentwicklung ausmachen. Besser ist es, die Gewährleistung der Datenintegrität aus den Anwendungsprogrammen "herauszufaktorisieren" und in deklarativer Form an das Datenbanksystem selbst zu delegieren. Dieses Vorgehen hat zwei große Vorteile:

- eine effektivere Kontrolle der Integrität
- eine signifikante Vereinfachung der Anwendungsprogrammierung

9.1.1 Typen von Integritätsbedingungen

- Statische Integritätsbedingungen (Invarianten bzgl. des Datenbankzustands, in Form von prädikatenlogischen Formeln)
  - datenmodellinhärente Integritätsbedingungen
    - Primärschlüsselbedingung
    - Fremdschlüsselbedingung
  - anwendungsspezifische Integritätsbedingungen
    - für ein Attribut eines Tupels
    - für ein Tupel
    - für mehrere Tupel einer Relation
    - für mehrere Relationen
- Dynamische Integritätsbedingungen (Invarianten bzgl. erlaubter Änderungen des Datenbankzustands)

Achtung:
Die logische Widerspruchsfreiheit der spezifizierten Integritätsbedingungen muß (vom Datenbankdesigner) sichergestellt werden. Widerspruchsfreiheit bedeutet hier, daß die Konjunktion aller Integritätsbedingungen erfüllbar (im Sinne der Logik) sein muß.

Zeitpunkt der Prüfung von Integritätsbedingungen

- am Ende einer Datenbankoperation (SQL-Anweisung)
- am Ende einer Transaktion (beim COMMIT WORK)

Reaktion auf Integritätsverletzungen

- Nichtausführung bzw. Rückgängigmachen der Datenbankoperation
- Abbruch der Transaktion (implizites ROLLBACK WORK)
- Ausführung von Folgeänderungen zur Wiederherstellung der Integrität

Beispiele:

Statische Integritätsbedingungen

1) Der Rabatt eines Kunden darf nicht über 50 Prozent liegen.
2) Der Rabatt eines ausländischen Kunden darf nicht über 30 Prozent liegen. (Annahme: Es gibt ein zusätzliches Kundenattribut Land.)
3) Der durchschnittliche Rabatt aller Kunden darf 30 Prozent nicht überschreiten.
4) Der Gesamtwert aller Produkte im selben Lager darf 1 Mio. DM nicht überschreiten.
5) Es muß mindestens ein Produkt geben.
6) Die Rechnungssumme einer Bestellung ergibt sich aus dem Produkt von Preis und bestellter Menge des bestellten Produkts abzüglich des Kundenrabatts.
7) Der Saldo eines Kunden ist die (negative) Summe der Rechnungssummen aller noch nicht bezahlten Bestellungen des Kunden.

Dynamische Integritätsbedingungen

8) Der Rabatt eines Kunden darf nie reduziert werden.
9) Der Rabatt eines Kunden darf innerhalb eines Jahres um maximal 10 Prozent angehoben werden.
10) Von Kunden, deren Saldo unter - 100000 DM liegt, werden keine Bestellungen mehr angenommen.
11) Der Status einer Bestellung darf sich nur in "geliefert" ändern, der Status einer gelieferten Bestellung nur in "bezahlt". Der Status einer bezahlten Bestellung darf sich nie mehr ändern.

9.1.2 Ausdrucksmittel zur Spezifikation von Integritätsbedingungen in relationalen DBS

Achtung: Die Möglichkeiten, die der SQL-Standard vorsieht, sind nicht notwendigerweise in allen Produkten (identisch) implementiert.

1) beim CREATE TABLE und mit CREATE ASSERTION

"Grobsyntax":

```
CREATE TABLE tablename
( colname datatype [DEFAULT {defaultconst | NULL}] [colconstraint {, colconstraint …}],
{, colname datatype [DEFAULT {defaultconst | NULL}] [colconstraint {, colconstraint …}] …}
)
[tabconstraint {, tabconstraint …}]

colconstraint ::= NOT NULL |
                  [CONSTRAINT constrname]
                    { UNIQUE | PRIMARY KEY | CHECK (searchcond) |
                      REFERENCES  tablename [(colname)] […]  }

tabconstraint ::= [CONSTRAINT constrname]
                    { UNIQUE colname {, colname …} |
                      PRIMARY KEY colname {, colname …} |
                      CHECK (searchcond) |
                      FOREIGN KEY colname {, colname …}
                        REFERENCES  tablename [(colname)] […]  }

CREATE ASSERTION assertname CHECK (searchcond)
                     [INITIALLY {DEFERRED | IMMEDIATE}]
```

Constraints und Assertions sind benannt, um auf sie bei ALTER TABLE ... DROP CONSTRAINT ... oder DROP ASSERTION ... Bezug nehmen zu können.

Semantik von Constraints und Assertions:

Die in der CHECK-Klausel erlaubten Search Conditions sind genau die, die in der WHERE-Klausel einer Anfrage erlaubt sind, können also, wenn man sich auf den Kern von SQL beschränkt, mittels sql2trc' auf Formeln des sicheren TRK abgebildet werden (siehe Kapitel 5). Diese sind von der Form F(t1, ..., tn) mit den freien Tupelvariablen t1, ..., tn, die jeweils einer der Relationen R1, ..., Rn zugeordnet werden können. (Im einfachsten Fall gibt es genau eine freie Variable für die Relation, zu deren CREATE-TABLE-Anweisung die Constraint gehört.) Die Semantik der CHECK-Klausel ist dann die folgende prädikatenlogische Formel ohne freie Variablen:

$$\forall t1 \ldots \forall tn \, (\, (\, t1 \in R1 \wedge \ldots \wedge tn \in Rn \,) \Rightarrow F(t1, \ldots, tn) \,)$$

Da eine Datenbank als (Modell der) konjunktive Verknüpfung einer Menge H elementarer prädikatenlogischer Formeln (mit jeweils einer solchen Formel pro Tupel) interpretiert werden kann (siehe Kapitel 4), können Constraints und Assertions einfach als weitere konjunktiv mit H verknüpfte (nichtelementare) Formeln interpretiert werden. Für Integritätsbedingungen I1, ..., Im ist die Datenbank damit eine Formel $H \wedge I1 \wedge \ldots \wedge Im$ bzw. ein Modell dieser Formel.

Dabei gelten Constraints, die beim CREATE TABLE spezifiziert werden,

für leere Relationen immer als erfüllt.

Mögliche Implementierung von Integritätsprüfungen:

Für jede SQL-Änderungsanweisung, die eine in einer CHECK-Klausel definierte Integritätsbedingung potentiell verletzen könnte, wird eine SQL-Anfrage wie folgt generiert:

Die searchcond der CHECK-Klausel entspreche der prädikatenlogischen Formel F(t1, ..., tn) mit freien Variablen t1, ..., tn, die Relationen R1, ..., Rn zugeordnet sind. Dann wird die Anfrage SELECT t1, ..., tn FROM R1, ..., Rn WHERE NOT searchcond generiert. Wenn das Anfrageergebnis nichtleer ist, wird die Integritätsbedingung verletzt, ansonsten ist sie eingehalten.

Zeitpunkt der Integritätsprüfung:

am Ende jeder SQL-Änderungsanweisung (bei ASSERTION ... IMMEDIATE)
oder am Ende der Transaktion (bei ASSERTION ... DEFERRED).

Reaktion bei Integritätsverletzung:

Die SQL-Änderungsanweisung wird nicht ausgeführt bzw. rückgängig gemacht;
bei verzögerter Prüfung wird die gesamte Transaktion zurückgesetzt.
Eine flexiblere Reaktion ist nur für Verletzungen der referentiellen Integrität vorgesehen.

Beispiele:

Bedingungen 1, 2, 3:

```
CREATE TABLE Kunden ( ...
        Rabatt FLOAT
                CONSTRAINT Rabattbedingung CHECK (Rabatt BETWEEN 0.0 AND 0.5)
                CONSTRAINT Auslandsrabattbedingung CHECK (Land = 'D' OR Rabatt <= 0.3),
        ...,
        CONSTRAINT Durchschnittsrabattbedingung CHECK
                (0.3 >= (SELECT AVG(Rabatt) FROM Kunden))  )
```

Bedingung 4:

```
CREATE TABLE Produkte ( ...,
        CONSTRAINT Lagerwertbedingung CHECK
        (1000000.0 >=ALL (SELECT SUM(Vorrat*Preis) FROM Produkte GROUP BY Lager))  )
```

Bedingung 5:

```
CREATE ASSERTION Produktexistenzbedingung CHECK
        (EXISTS (SELECT * FROM Produkte) )
```

Bedingung 6:

```
CREATE ASSERTION Rechnungssummenbedingung CHECK
        ( Bestellungen.Summe =
        (SELECT B.Menge * Produkte.Preis * (1.0 - Kunden.Rabatt)
        FROM Bestellungen B, Produkte, Kunden
        WHERE B.PNr=Produkte.PNr AND B.KNr=Kunden.KNr
        AND B.BestNr=Bestellungen.BestNr ) )
```

Bedingung 7:

```
CREATE ASSERTION Saldobedingung CHECK
        ( Kunden.Saldo =     (SELECT SUM(Summe) FROM Bestellungen WHERE
                                Bestellungen.KNr=Kunden.KNr AND Status <> 'bezahlt') )
```

Die Semantik von Bedingung 6 beispielsweise ist dann die logische Invariante:

$\forall$ best (best $\in$ Bestellungen $\Rightarrow$
($\exists$ b $\exists$ p $\exists$ k (b $\in$ Bestellungen $\wedge$ p $\in$ Produkte $\wedge$ k $\in$ Kunden $\wedge$

b.PNr=P.PNr ∧ b.KNr=k.KNr ∧ b.BestNr=best.BestNr ∧
best.Summe = b.Menge * p.Preis * (1 - k.Rabatt))))

Flexible Reaktionsmöglichkeit bei Verletzung der referentiellen Integrität

"Grobsyntax" der FOREIGN-KEY-Klausel:

... FOREIGN KEY (column {, column ...})
REFERENCES [user .] table [(column {, column ...})]
[ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
[ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT }]
[INITIALLY { IMMEDIATE | DEFERRED }] ...

Semantik von CREATE TABLE R ...
FOREIGN KEY A1, …, Am REFERENCES R1 (B1), …, Rm (Bm):

$$\forall t \, (t \in R \Rightarrow ((t.A1=\omega \wedge \ldots \wedge t.Am=\omega) \vee (\exists t1 \ldots \exists tm \, (t1 \in R1 \wedge \ldots \wedge tm \in Rm \wedge t1.B1=t.A1 \wedge \ldots \wedge tm.Bm=t.Am))))$$

Zeitpunkt der Integritätsprüfung:
am Ende jeder SQL-Anweisung oder am Ende der Transaktion

Reaktion bei Integritätsverletzung:
Zurückweisung der Löschung/Änderung (bei NO ACTION)
Löschen/Ändern aller "abhängigen" Tupel (bei CASCADE)
Fremdschlüssel in allen "abhängigen" Tupeln auf Nullwert/Default-Wert setzen

Beispiel:

Kunden

KNr	**...**
1	
2	
...	

Produkte

PNr	**...**
1	
2	
...	

Bestellungen
(Fremdschlüssel KNr)

BestNr	**Monat**	**Tag**	**KNr**	**Summe**	**Status**
1001			1		
1002			2		
1003			1		
...					

Bestellposten
(Fremdschlüssel BestNr, PNr)

BestNr	**PNr**	**Menge**
1001	1	
1002	2	
1003	1	
1003	2	
...		

```
CREATE TABLE Bestellungen ( ... ,
        FOREIGN KEY KNr REFERENCES Kunden (KNr)
        ON DELETE SET NULL )
CREATE TABLE Bestellposten ( ... ,
        FOREIGN KEY PNr REFERENCES Produkte (PNr)
        ON DELETE CASCADE,
        FOREIGN KEY BestNr REFERENCES Bestellungen (BestNr)
        ON DELETE CASCADE )
```

Löschen von Kunde 1 $\Rightarrow$ Bestellungen 1001 und 1003 erhalten den Nullwert als KNr
Löschen von Produkt 1 $\Rightarrow$ Die Bestellposten für Produkt 1 werden gelöscht.

2) Trigger

Kernidee:
Vor oder nach einer bestimmten Art von Änderungsoperationen wird bei Erfüllung einer spezifizierten Bedingung eine Folge von SQL-Anweisungen automatisch ausgeführt. (Sprechweise: Der Trigger "feuert".)

Vorteile gegenüber rein deklarativer Spezifikation von Integritätsbedingungen:

- flexible Reaktion auf Integritätsverletzungen
- spezifischere Wahl der Überprüfungszeitpunkte
  (und damit u.U. eine effizientere, wenngleich eher prozedurale, Realisierung der Integritätssicherung)

"Grobsyntax":

CREATE TRIGGER trigger-name
{BEFORE | AFTER } { DELETE | INSERT | UPDATE [OF column {, column …}] }
ON table [REFERENCING OLD AS correlation-var NEW AS correlation-var]
[FOR EACH ROW | FOR EACH STATEMENT]
[WHEN (condition)]
(statement-sequence)

Dabei kann die REFERENCING-Klausel nur in Kombination mit FOR EACH ROW stehen.

Semantik:

Vor oder nach der spezifizierten Änderungsoperation wird die Condition in der WHEN-Klausel ausgewertet. Wenn Condition wahr ist, wird die spezifizierte Statement-Sequence ausgeführt, und zwar entweder wiederholt für jedes von der Änderungsoperation geänderte Tupel (Option FOR EACH ROW) oder nur einmal am Ende der gesamten Änderungsoperation (Option FOR EACH STATEMENT).

Die Condition der WHEN-Klausel ist dabei von der syntaktischen Form einer WHERE-Klausel (bzw. Constraint), im Kern also einer prädikatenlogischen Formel F entsprechend.

Bei der Option FOR EACH STATEMENT darf F keine freien Variablen haben. Nur wenn die Auswertung von F den Wert wahr ergibt, wird die vorgesehene Statement-Sequence ausgeführt.

Bei der Option FOR EACH ROW kann die der WHEN-Condition entsprechende prädikatenlogische Formel F ein oder zwei freie Variablen haben, die sich beide auf die von der Änderungsoperation betroffene Tabelle beziehen. Diese beiden Tupelvariablen - told und tnew - sind die in der REFERENCING-Klausel bei OLD bzw. NEW spezifizierten Variablen; fehlt die REFERENCING-Klausel, so gibt es nur eine implizite Tupelvariable t (den Tabellennamen), und zwar in der NEW-Rolle, falls AFTER spezifiziert wurde, und in der OLD-Rolle, falls BEFORE spezifiziert wurde. Die Tupelvariablen tnew, told bzw. t sind syntaktisch freie Variablen in t, sie werden aber wie Konstanten behandelt, bei denen die Attributwerte durch den alten bzw. neuen Zustand des aktuell geänderten Tupels gesetzt. Nur wenn mit dieser Substitution die Auswertung von F den Wert wahr ergibt, wird die vorgesehen Statement-Sequence ausgeführt.

Bei INSERT oder DELETE als Trigger-Ereignis ist - je nach Angabe von BEFORE oder AFTER - nur jeweils eine der beiden Tupelvariablen told und tnew relevant und die Angabe einer REFERENCING-Klausel sinnlos.

Beispiele:

Bedingung 7:
```
CREATE TRIGGER Saldoeintrag
AFTER INSERT ON Bestellungen FOR EACH ROW WHEN ( Status = 'neu' )
( UPDATE Kunden SET Saldo = Saldo - Summe WHERE Kunden.KNr=Bestellungen.KNr )
```
```
CREATE TRIGGER Saldoausgleich
AFTER UPDATE OF Status ON Bestellungen
REFERENCING OLD AS BOld  NEW AS BNew
FOR EACH ROW
WHEN ( BNew.Status = 'bezahlt' AND BOld.Status <> 'bezahlt' )
( UPDATE Kunden SET Saldo = Saldo + Summe WHERE Kunden.KNr=Bestellungen.KNr )
```

Bedingung 8:
```
CREATE TRIGGER Rabattmonotonie
AFTER UPDATE OF Rabatt ON Kunden
REFERENCING OLD AS KOld  NEW AS KNew
FOR EACH ROW
WHEN ( KNew.Rabatt < KOld.Rabatt )
      ( ROLLBACK WORK )
```

Bedingung 10:
```
CREATE TRIGGER Kundensperrung
BEFORE INSERT ON Bestellungen FOR EACH ROW
WHEN (    (SELECT Saldo FROM Kunden
           WHERE Kunden.KNr=Bestellungen.KNr) < -100000.0 )
      ( <Fehlermeldung ausgeben>; ROLLBACK WORK )
```

Achtung: Die Reihenfolge, in der die Trigger "feuern", ist u.U. essentiell.
Die durch einen Trigger ausgelöste Anweisungsfolge kann selbst wieder andere oder denselben Trigger "feuern".
(Da die Termination der so ausgelösten Anweisungsketten i.a. unentscheidbar ist und somit nicht prüfbar ist, beschränken kommerzielle Datenbanksystem zur Laufzeit einfach die maximale Schachtelung von „feuernden“ Trigger-Ausführungen.)

→ Trigger sind ein sehr mächtiges Konzept zur Integritätssicherung; Triggerspezifikationen sind aber auch potentiell sehr fehleranfällig.

Trigger sind nicht nur zur unmittelbaren Integritätssicherung nützlich, sondern können u.U. auch zur aktiven Steuerung der „Business-Logik“ verwendet werden..

Beispiel:

```
CREATE TRIGGER Kundenbeförderung
AFTER INSERT ON Bestellungen
REFERENCING NEW AS BNew
FOR EACH ROW
WHEN (    (   (SELECT SUM(Summe) FROM Bestellungen
                 WHERE Bestellungen.KNr=BNew.KNr) > 100000.0 )
              AND
              (   (SELECT SUM(Summe) - BNew.Summe FROM Bestellungen
                 WHERE Bestellungen.KNr=BNew.KNr) <= 100000.0 ) )
(   UPDATE Kunden SET Rabatt = Rabatt + 0.05 WHERE Kunden.KNr=BNew.KNr;
    INSERT INTO Stammkunden SELECT * FROM Kunden WHERE Kunden.KNr=BNew.KNr )
```

In der durch den Trigger ausgelösten SQL-Anweisungsfolge können auch Stored Procedures aufgerufen werden (die z.B. bei Oracle in PL/SQL oder in Java geschrieben sein können).

Weiterentwicklung des Trigger-Konzepts für "Aktive Datenbanken"

In sog. aktiven Datenbanken hat man das versucht, das Trigger-Konzept auf allgemeinere **ECA-Regeln** der Form

on event **if** condition **do** action

zu erweitern. Im Gegensatz zu einem Trigger können sich ECA-Regeln auf zusammengesetzte Ereignisse (also nicht nur auf einzelne SQL-Änderungsoperationen) beziehen (Beispiele s.u.). Diese Ansätze hatten Einflüsse auf Workflow-Management-Systeme (siehe Kapitel 14), sind in kommerziellen Datenbanksystemen selbst aber nur in geringem Maße und nur für interne Zwecke realisiert.

Beispiele:

1) Auffüllen des Lagers für knappe Produkte automatisch initiieren:
   on after update Produkte.Vorrat
   if Produkte.Vorrat < 100
   do execute LagerAuffüllen(...)
2) Werbebriefe drucken für Kunden, die seit einem Jahr nichts mehr bestellt haben:
   on 0:00 **daily**
   do execute JunkMail(...)
3) Verschicken eines Mahnbriefs an Kunden, die dreimal hintereinander die Zahlungsfrist überschreiten.
4) Ausgabe einer Meldung, wenn der Kurs einer Aktie innerhalb der letzten 5 Minuten um mehr als 50% variiert.

9.2 Views (Sichten, virtuelle Relationen, intensionale Relationen)

Idee (eine von mehreren Motivationen für das View-Konzept):

Integritätssicherung wird einfacher, wenn weniger abgeleitete Daten gespeichert werden. Solche abgeleiteten Daten (z.B. Saldo) sollen vielmehr nur bei Bedarf berechnet werden. Um die Formulierung der entsprechenden Anfragen so einfach wie möglich zu machen, können abgeleitete Daten als "Views" zur Verfügung gestellt werden. Views erscheinen gegenüber dem SQL-Programmierer praktisch wie gespeicherte Relationen, ohne daß die Tupel der View wirklich gespeichert sind.

"Grobsyntax" zur Definition von Views:
CREATE VIEW view-name [(column {, column …})]
AS select-block [WITH CHECK OPTION]

Beispiel:
gespeicherte Relationen:
Kunden (KNr, Name, Stadt, Rabatt)
Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge, Summe, Status)
zusätzliche View:

```
CREATE VIEW KundenInfo ( KNr, Name, Stadt, Rabatt, Saldo ) AS
    SELECT Kunden.KNr, Name, Stadt, Rabatt, SUM(Summe)
    FROM Kunden, Bestellungen
    WHERE Kunden.KNr = Bestellungen.KNr
    AND Status <> 'bezahlt'
    GROUP BY KNr, Name, Stadt, Rabatt
```

Abfrage des Saldos:

```
SELECT Saldo FROM KundenInfo WHERE KNr=1
```

oder beispielsweise auch:

```
SELECT Saldo FROM KundenInfo, Bestellungen
WHERE KundenInfo.KNr=Bestellungen.KNr AND BestNr=1001
```

Views können generell zur Vereinfachung von Abfragen definiert werden
(analog zu Zuweisungen in der Relationenalgebra).
Auf Views können wiederum weitere Views definiert werden.

Beispiele:

1) CREATE VIEW BestellungsInfo
   (BestNr, Monat, Tag, KNr, Kundenname, Rabatt, PNr, Produktbez, Menge, Summe, Status)
   AS
   SELECT BestNr, Monat, Tag, Kunden.KNr, Name, Rabatt,
   Produkte.PNr, Bez, Menge, Summe, Status
   FROM Bestellungen, Kunden, Produkte
   WHERE Bestellungen.KNr=Kunden.KNr AND Bestellungen.PNr=Produkte.PNr
   Anfrage z.B.:
   SELECT Kundenname FROM BestellungsInfo WHERE Produktbez='Platte'
2) CREATE VIEW SuperBestellungsInfo AS
   SELECT * FROM BestellungsInfo WHERE Summe > 10000.0

Ausführung von Operationen auf Views

Anfragen auf Views werden DBS-intern durch Substitution in Anfragen auf
gespeicherten Relationen transformiert.

Für CREATE VIEW VIRT AS viewquery ist
sql2ra [SELECT A1, …, Am FROM TAB1, …, TABk, VIRT WHERE F] =
π[A1, ..., Am] (sql2ra' [F] (TAB1 × … × TABk × sql2ra[viewquery]))

Einfaches Beispiel:
σ[Rabatt>0.3] (SuperBestellungsInfo)
= σ[Rabatt>0.3] (σ[Summe>10000.0] (BestellungsInfo))
= σ[Rabatt>0.3] (σ[Summe>10000.0] (π[BestNr, ...] (Kunden |x| Bestellungen |x| Produkte)))

Mit Hilfe von Views sind u.U. sogar Anfragen möglich, die mit der SELECT-Anweisung des ursprünglichen SQL-Standards nicht oder nur schwierig ausdrückbar wären.

Beispiel:

Gegeben sei die Relation Prüfungen (Fach, Student, Note).
Welches ist das Fach mit der besten Durchschnittsnote?

Die Lösung

```
SELECT Fach FROM Prüfungen GROUP BY Fach
HAVING AVG(Note) <= ALL  (  SELECT AVG(Note) FROM Prüfungen
                            GROUP BY Fach )
```

war lange Zeit in vielen Systemen unzulässig (wird aber vom SQL-Standard erlaubt und z.B. von Oracle unterstützt).

Die einfachere Lösung mittels View:

```
CREATE VIEW Fachstatistik AS
   SELECT Fach, AVG(Note) AS Durchschnitt FROM Prüfungen
   GROUP BY Fach;
SELECT Fach FROM Fachstatistik
WHERE Durchschnitt = (  SELECT MIN(Durchschnitt) FROM Fachstatistik )
```

entspricht einer geschachtelten Subquery in der FROM-Klausel,
was in SQL92 in direkter Form (also ohne View) unzulässig wäre
und auch nur von einem Teil der kommerziell wichtigen Produkte unterstützt wird.

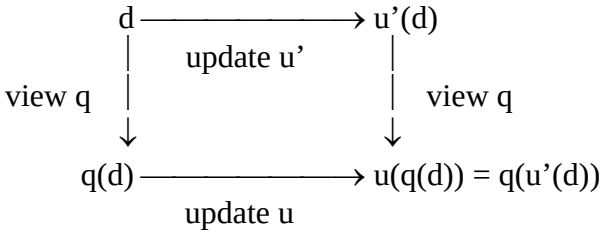
Ausführung von Update-Operationen auf Views

Änderungen eines Tupels einer View sind nur möglich, wenn sie eindeutig auf ein Tupel einer gespeicherten Relation abgebildet werden können (analog für Einfügen und Löschen).

Definitionen:

Sei D die Menge aller Datenbanken. Eine View ist eine partielle Funktion q: D → D; ein Update ist eine partielle Funktion u: D → D. Eine View q heißt *änderbar* genau dann, wenn es für jede Datenbank d, auf der q definiert ist, und jeden auf q(d) definierten Update u einen eindeutigen Update u' gibt, der auf d definiert ist und für den u(q(d)) = q(u'(d)) gilt.
(Zusätzlich wird häufig informell gewünscht, daß u' "intuitiv" definiert ist.)

Das folgende Abbildungsdiagramm muß also „kommutieren“:



Beispiele:

1) UPDATE KundenInfo SET Stadt='Homburg' WHERE KNr=1
   ist erlaubt
2) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE Monat=11
   ist nicht erlaubt !
3) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE PNr=1
   ist theoretisch zulässig, aber in den meisten DBS nicht erlaubt !
4) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE BestNr=1
   ist nicht erlaubt !
5) CREATE VIEW BestellungsKurzInfo (BestNr, Kundenname, Produktbez, Menge) AS
   SELECT BestNr, Name, Bez, Menge FROM Kunden, Bestellungen, Produkte
   WHERE Bestellungen.KNr=Kunden.KNr AND Bestellungen.PNr=Produkte.PNr
   INSERT INTO BestellungsKurzInfo VALUES (1111, 'Hempel', 'Maus', 5)
   ist nicht erlaubt (der zu generierende Update u' ist nicht eindeutig bestimmt) !
6) UPDATE KundenInfo SET Saldo = Saldo + 1000.0 WHERE KNr=1
   ist nicht erlaubt (der zu generierende Update u' ist nicht eindeutig bestimmt) !

Die Entscheidung, ob eine View änderbar ist oder nicht, ist ein schwieriges Problem. Daher erlauben die meisten Datenbanksysteme View-Updates nur, wenn die View durch Selektion und ggf. Projektion aus einer einzigen Tabelle erzeugt ist und den Primärschlüssel der Tabelle enthält. (Dies ist übertrieben konservativ.)

Änderungen eines View-Tupels, die dazu führen, daß das Tupel aus der View "verschwindet", können durch Spezifikation der CHECK OPTION verboten werden.

Beispiel:

CREATE VIEW SuperKunden AS SELECT * FROM Kunden WHERE Rabatt>0.3
WITH CHECK OPTION
INSERT INTO SuperKunden (KNr, Name, Stadt, Rabatt) VALUES (100, 'Meier', 'Homburg', 0.1)
wird zurückgewiesen
UPDATE SuperKunden SET Rabatt = Rabatt - 0.05 WHERE KNr=10
wird u.U. zurückgewiesen

Views als Mittel zur Datenunabhängigkeit bei Schemaänderungen

bisherige Anfrage:
SELECT * FROM Kunden WHERE KNr=1
Schemaänderung (plus Datenbank aktualisieren oder neu laden):
ALTER TABLE Kunden ADD Vorname CHAR(20), Nachname CHAR(20)
UPDATE Kunden SET Vorname = SUBSTR (Name, ...), Nachname = SUBSTR (Name, ...)
ALTER TABLE Kunden DROP Name
Vorgehen zur "Bewahrung" der bisherigen Anfrage:

- Kunden in KundenDaten umbenennen
- CREATE VIEW Kunden (KNr, Name, Stadt, Rabatt, Saldo) AS
  SELECT KNr, Vorname | | ' ' | | Nachname, Stadt, Rabatt, Saldo FROM KundenDaten

aber: Kunden.Name ist nicht änderbar!

9.3 Datenschutz und Zugriffskontrolle

Begriffe

Datenschutz (engl.: data privacy):
Einschränkungen bei der Speicherung und Verarbeitung "kritischer" Daten, insbesondere personenbezogener Daten (Schutz der Privatsphäre von Personen)
→ Datenschutzgesetz („schützt Personen vor Daten“)

Zugriffskontrolle / Autorisierung (engl.: data security, authorization):
Verhinderung von unbefugten Zugriffen auf gespeicherte Daten („schützt Daten vor Personen“)

Maßnahmen der Zugriffskontrolle

1) Organisatorische Maßnahmen (kontrollierter Zugang zu den Rechnerräumen, zum u.U. drahtlosen Netz, usw.; z.B. Zugriff nur von bestimmten Geräten oder IP-Nummern)
2) Technische Maßnahmen (Datenverschlüsselung, kryptographische Protokolle etc.)
3) Maßnahmen des Betriebssystems
(Die der Datenbank zugrundeliegenden Dateien bzw. Platten sind nur für das DBS zugreifbar, also z.B. nur vom Account "Oracle" aus.)
4) Authentikation des DB-Benutzers
(typischerweise durch Angabe eines Kennworts beim CONNECT; besser durch digitalen Schlüssel oder andere kryptographische Protokolle)
5) Prüfung der Zugriffsrechte des DB-Benutzers beim Zugriff auf Daten

Prüfung von Zugriffsrechten ("Discretionary Access Control")

Prinzip:
Subjekte haben *Rechte* (zur Ausführung von Operationen) auf *Objekten*.

Vergabe von Rechten durch die GRANT-Anweisung in SQL.
"Grobsyntax":

```
GRANT { ALL | privilege {, privilege ...} } ON { table | view }
TO { PUBLIC | user {, user …} } [ WITH GRANT OPTION ]
```

Mögliche Rechte zum Zugriff auf relationale Datenbanken sind:

SELECT	lesender Zugriff auf eine Relation
INSERT	Einfügen in eine Relation
UPDATE	Ändern von Tupeln einer Relation (ggf. nur bestimmte Attribute)
DELETE	Löschen von Tupeln einer Relation
CONNECT	Verbindung zum DBS aufnehmen ("Login"-Recht)
RESOURCE	Anlegen neuer Relationen (ggf. mit Limit für den Plattenplatz)
DBA	Datenbankadministration (z.B. Aufruf von Dienstprogrammen)
EXECUTE	Ausführung eines Anwendungsprogramms
IO_LIMIT	Beschränkung des Ressourcenverbrauchs für SQL-Anweisungen

Beispiele:
1) Benutzer Meier hat das Recht zur Ausführung von SELECT-Anweisungen
   auf der Relation Bestellungen.
   GRANT SELECT ON Bestellungen TO Meier
2) Benutzer Meier hat das Recht zur Ausführung des Programms Lieferung.
   GRANT EXECUTE Lieferung TO Meier
3) Das Programm Lieferung hat das Recht zur Änderung der Relation Bestellungen.
   GRANT UPDATE Status ON Bestellungen TO Lieferung

Prädikatorientierte Verfeinerung von Rechten durch Views

Beispiel:
Benutzer Meier hat das Recht zum Lesen der Kundendaten der Stadt Homburg.
CREATE VIEW KundenHOM AS SELECT * FROM Kunden
WHERE Stadt='Homburg'
GRANT SELECT ON KundenHOM TO Meier

Weitergabe und Rücknahme von Rechten

Für jedes Objekt gibt es genau ein Subjekt, genannt *Eigentümer* (engl.: owner),
das alle Rechte für das Objekt besitzt.
Rechte können mit GRANT an andere Subjekte weitergegeben werden.

Bei Angabe der GRANT OPTION darf der Empfänger eines Rechts dieses selbst
wiederum an andere Subjekte weitergeben !
Weitergegebene Rechte können mit der Anweisung
REVOKE privilege FROM user
wieder zurückgenommen werden.

Problemszenario:

Meier sei der Eigentümer der Relation MeierTabelle

Meier:	GRANT SELECT ON MeierTabelle TO Schmid WITH GRANT OPTION
Schmid:	GRANT SELECT ON MeierTabelle TO SchmidFreund WITH GRANT OPTION
Meier:	REVOKE SELECT ON MeierTabelle FROM Schmid
SchmidFreund:	GRANT SELECT ON MeierTabelle TO Schmid

Lösung in relationalen DBS:
REVOKE wirkt transitiv, nimmt also auch die vom Empfänger eines Rechts
an Dritte weitergegebenen Rechte wieder zurück.

Ergänzende Literatur zu Kapitel 9:

J. Melton, A.R. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

A. Behrend, R. Manthey, B. Pieper, An Amateur's Introduction to Integrity Constraints and Integrity Checking in SQL, 9. GI-Fachtagung über Datenbanksysteme in Büro, Technik und Wissenschaft, Oldenburg, 2001, Springer-Verlag

J. Widom, S. Ceri (Editors), Active Database Systems, Morgan Kaufmann, 1996

S. Ceri, R.J. Cochrane, J. Widom, Practical Applications of Triggers and Constraints: Successes and Lingering Issues, International Conference on Very Large Data Bases (VLDB), Cairo, 2000

S. Castano, M. Fugini, G. Martella, P. Samarati, Database Security, Addison-Wesley, 1995

Kapitel 10: Objektorientierte Datenbanksprachen und Erweiterungen von SQL

10.1 Schwachpunkte relationaler Datenbanksysteme

Relationale Datenbanksysteme haben sich in vielen Anwendungen bewährt und repräsentieren in vielerlei Hinsicht den Stand der Kunst in der Datenbanktechnologie. Es gibt jedoch auch Anwendungsklassen, die von relationalen Datenbanksystemen nur schlecht unterstützt werden. Eine bessere Unterstützung solcher Anwendungsklassen ist eine Hauptmotivation für objektorientierte Datenbanksysteme (bzw. allgemeiner jede Art von “postrelationalen" Datenbanksystemen). Zu diesen besonders anspruchsvollen Anwendungsklassen zählen:

- *CAD (Computer-Aided Design):*
  Im mechanischen CAD ist z.B. die Geometrie von Werkstücken zu modellieren und in einer Datenbank abzulegen. Ein Standardvorgehen dafür ist die Beschreibung eines 3-dimensionalen Körpers durch Anwenden von Zusammensetzungsoperationen auf eine Reihe von (ggf. transformierten) Standardvolumina wie Quader, Kegel und Zylinder. Im elektronischen CAD (VLSI-Entwurf) ist der Aufbau von Schaltungen zu modellieren. Dies sind z.T. sehr tiefe Hierarchien, die aus einer enormen Anzahl von Bausteinen bestehen können. Darüber hinaus ist u.a. auch das Layout eines Chips zu beschreiben, was wiederum auf geometrische Strukturen führt.
- *Geowissenschaften:*
  Kartographische Daten sind in praktisch allen Geowissenschaften zu verwalten (Geographie, Geologie, Seismologie, Ozeanographie, ...). Diese Anwendungen benötigen beispielsweise Daten über Ländergrenzen, Stadtgrenzen, Grundstückskataster, Straßenverläufe, Leitungspläne, Gewässer, usw. Häufig kommen zu den eigentlichen geographischen Daten noch weitergehende, durch Meßstationen oder auf sonstige Weise erhobene ortsbezogene Daten wie z.B. Emissionswerte, Gewässerverschmutzung, aktuelle Autoverkehrsdichte, Flächennutzungspläne, usw., die die Grundlage für ökologische Analysen bilden. Mit diesen Daten lassen sich auch Dienste für den Alltagsgebrauch realisieren, beispielsweise ein Autofahrernavigationssystem.
- *Desktop Publishing:*
  Auf dem Rechner erstellte Textdokumente haben eine inhaltliche Struktur (Kapitel, Abschnitte, Absätze, Sätze, Referenzen) und eine Layout-Struktur (Seitenumbruch usw.). Beide Aspekte sind zu modellieren und in einer Datenbank zu hinterlegen. Darüber hinaus sind “Hypertext"-Strukturen (z.B. HTML) populär, bei denen man vom “linearen" Lesen eines Dokuments abweicht und stattdessen Referenzen (sog. “Hyperlinks") verfolgt. Beispielsweise könnte man beim Nachschlagen in einem Lexikon oder auch beim Durcharbeiten eines Lehrbuches über solche Referenzen je nach Vorkenntnissen oder spezifischen Informationsbedürfnissen geeignet geführt werden. Ferner sind Textdokumente durch Hinzunahme von Bildern und Grafiken häufig multimediale Dokumente bis hin zur Integration von Video- und Audioaufzeichnungen. Nachrichtenarchive - beispielsweise in Zeitungs- und Fernsehredaktionen, aber auch für öffentliche Bildungs- und Informationsangebote – basieren auf multimedialem Desktop Publishing.

Forderungen an objektorientierte bzw. “objektrelationale" Datenbanksysteme

1) Unterstützung komplexer Objektstrukturen:

Jedes Objekt der Anwendung soll als ein - komplex strukturiertes - Objekt der Datenbank repräsentierbar sein.

Beispiel:
Repräsentation einer Straße mit ihrem Verlauf als Polylinie, so daß einerseits einfach auf das Objekt “Straße" als Ganzes zugegriffen werden kann, andererseits aber auch einzelne Stützpunkte des Straßenverlaufs einfach verarbeitet werden können.

2) Erweiterbarkeit des DBS:

Das DBS soll erweiterbar sein, indem anwendungsspezifische Funktionen in der Datenbank registriert werden, die dann in beliebigen Anfragen und für integritätserhaltende Änderungen verwendbar sind. Die konkrete Implementierung einer solchen Funktion soll austauschbar bleiben, sofern ihre Schnittstelle und ihre Semantik erhalten bleiben.

Beispiel:
Definition einer Längenberechnungsfunktion für die Polylinie eines Straßenverlaufs, so daß die Funktion innerhalb von Queries frei verwendet werden kann.

3) Wiederverwendbarkeit:

Datenbankteilschemata und Anwendungsfunktionen sollten - sofern sie generischen Charakter haben - direkt wiederverwendbar sein.

Beispiel:
Wiederverwendung des Schemas von Polylinien und der entsprechenden Funktionen für Straßen, Bahnlinien, Stromleitungen, usw.

10.2 Grundkonzepte objektorientierter Datenmodelle

10.2.1 Komplexe Objekte (Strukturelle Objektorientierung)

Jedes *Objekt* hat eine Menge von Attributen (engl.: Attribute, Property). Für jedes Attribut besitzt das Objekt einen Wert; die Gesamtheit der Attributwerte eines Objekts bilden den Zustand des Objekts.

Jedes *Attribut* hat einen Namen und einen Wertebereich (engl.: Domain).
Zulässige Wertebereiche sind
- elementare Wertebereiche (Integer, Real, String, Boolean)
- *Referenzen* (*Relationships*) auf andere Objekte
- Verbunde (Structures, Records), Mengen (Sets), Multimengen (Bags), Listen (Lists), Felder (Arrays) mit zulässigen Wertebereichen für die jeweiligen Komponenten bzw. Elemente, insbesondere auch mit Objektreferenzen als Komponenten bzw. Elemente (Aggregation von Objekten)

Der *Typ* eines Objekts ist die Menge der für das Objekt definierten Attribute.

Die Menge der Objekte gleichen Typs werden zu einer *Klasse* zusammengefaßt (Klassifikation von Objekten).
Klassen sind selbst wieder Objekte (vom Typ Set<Elementobjekttyp>) und können zusätzliche (Meta-) Attribute haben wie z.B. die Kardinalität der Klasse, Schlüsseleigenschaft von Attributen.

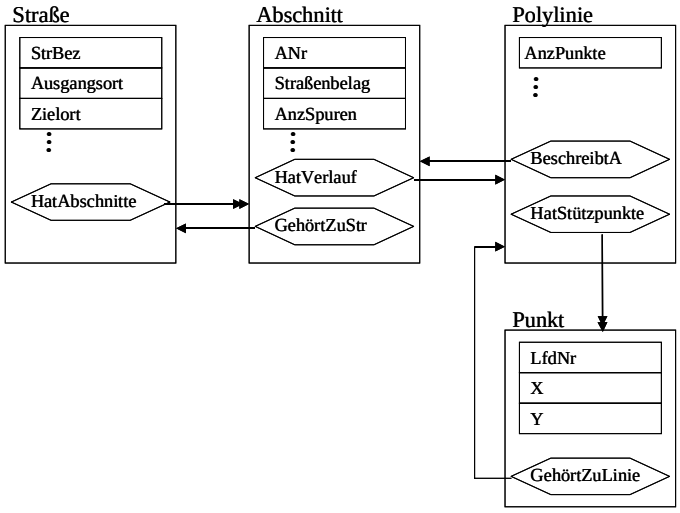
Eine objektorientierte Datenbank ist eine Menge von Objektklassen.

Bemerkung:
In objektorientierten Programmiersprachen (z.B. C++, Java) ist eine Klasse ein Programmodul, d.h. eine Menge von Operationen und ggf. Datenstrukturen, und es gibt Konstruktoren zur Erzeugung von Objekten, auf die dieser Programmodul anwendbar ist. Bei Programmiersprachen stehen typischerweise die Operationen (d.h. der Programmcode) im Vordergrund, bei DBS dagegen die Kollektion der Datenobjekte (d.h. der Instantiierungen der zugrundeliegenden Datenstruktur). Im Gegensatz zu DBS ist man in Programmiersprachen traditionell nicht an deklarativen Anfragen über Mengen von Objekten interessiert.

Beispiel:

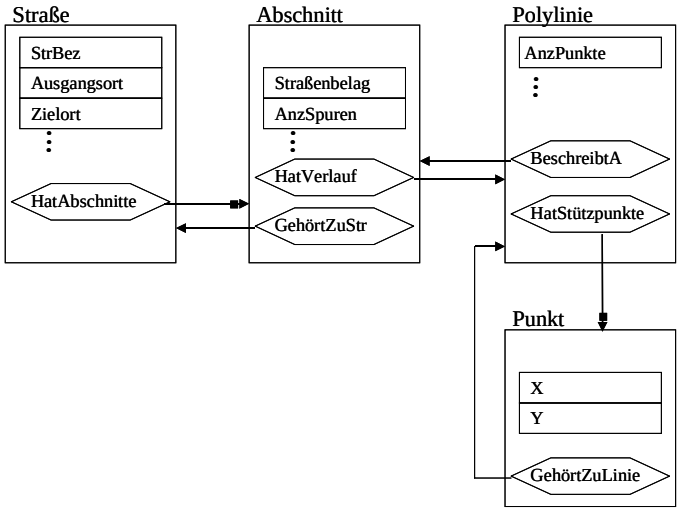
In einer Datenbank sollen Informationen über Straßen verwaltet werden. Eine Straße besteht aus mehreren Abschnitten, die sich im Straßenbelag, Spurenausbau usw. unterscheiden können. Der Verlauf eines Straßenabschnitts soll durch eine Polylinie modelliert werden; zwischen je zwei Stützpunkten wird mittels eines Geradenstücks interpoliert.

Beispiel - Alternative 1:




```
class Straße {
        extent Straßen;
        key StrBez;

        attribute String StrBez;
        attribute String Ausgangsort;
        attribute String Zielort;
        ...
        relationship Set<Abschnitt> HatAbschnitte;
        }
class Abschnitt {
        extent Abschnitte;
        key (GehörtZuStr.StrBez, ANr);

        attribute Integer ANr;
        attribute String Straßenbelag;
        attribute Integer AnzSpuren;
        attribute Integer Tempolimit;
        ..
        relationship Polylinie HatVerlauf;
        relationship Straße GehörtZuStr;
        }
...
```

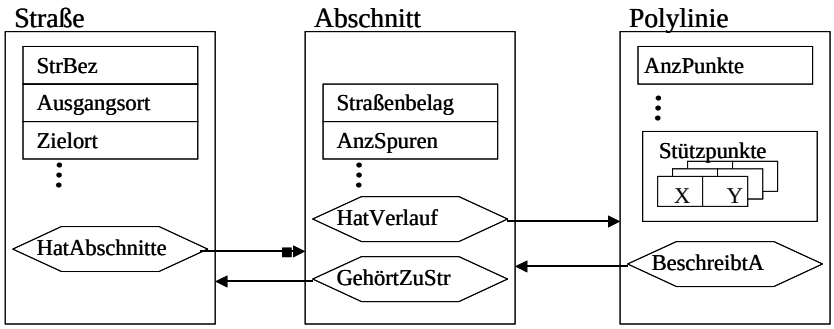
Beispiel - Alternative 2:




```
class Straße {
        ...
        relationship List<Abschnitt> HatAbschnitte;
        }
...
class Polylinie {
        ...
        relationship List<Punkt> HatStützpunkte;
        }
...
```

Im Gegensatz zu Alternative 1 sind hier die Referenzen auf Abschnitte und auf Punkte als Listen modelliert, so daß künstliche Attribute wie LfdNr entfallen können.

Beispiel - Alternative 3:



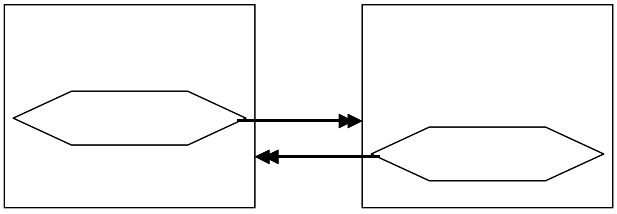
...

```
class Polylinie {
        ...
        attribute Integer AnzPunkte;
        attribute List<Struct<X: Real; Y: Real>> Stützpunkte;
        }
```

Im Gegensatz zu den Alternativen 1 und 2 sind hier Punkte als mengenwertige Attribute von Polylinien modelliert und nicht als eigenständige Objekte. Die Konsequenz ist, daß sich Polylinien nicht mehr Punkte teilen können, indem sie Referenzen auf dieselben Punkte enthalten. Stattdessen existieren hier in einem solchen Fall Kopien der Punktdaten in verschiedenen Polylinien. Die Konsequenz ist, daß bei einer Verschiebung eines Punktes ggf. Kopien von der Anwendung mit geändert werden müssen.

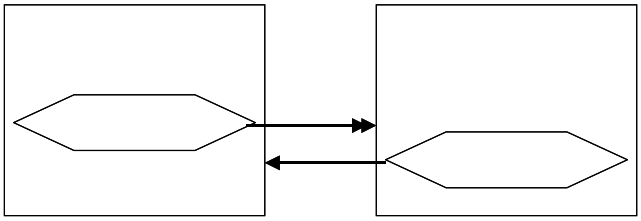
Verschiedene Arten von Relationships

M:N-Relationship (Viele-zu-viele-Beziehung, komplexe Beziehung)



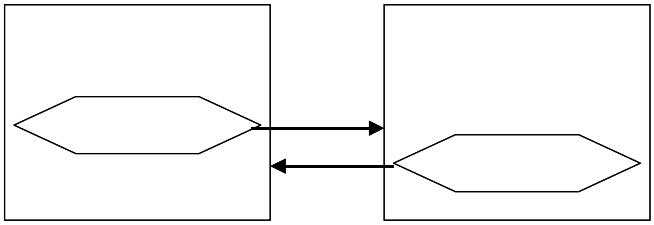
Beispiel: Lager und Produkte (sofern ein Produkt in mehreren Lagern gehalten werden kann)

1:N-Relationship (Eins-zu-viele-Beziehung, hierarchische Beziehung) bzw.
N:1-Relationship (Viele-zu-eins-Beziehung, hierarchische Beziehung)



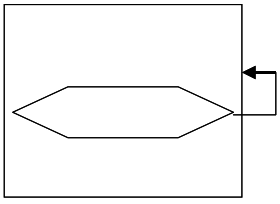
Beispiel: Bestellungen und Bestellposten

1:1-Relationship (Eins-zu-eins-Beziehung, injektive Beziehung)



Beispiel: Männer und Frauen in der Beziehung Ehe

Relationship zwischen Objekten derselben Klasse



Beispiel: Relationship „Chef“ der Klasse „Angestellte“ oder
Relationship “Ehepartner" der Klasse “Person"

Integritätsbedingungen für Relationships

In vielen Fällen soll eine Referenz in einem referenzierten Objekt wieder auf das referenzierende Objekt verweisen, also eine Art Rückwärtszeiger darstellen. Wenn man Referenzen bzw. Relationships als Abbildungen (bzw. Relationen) zwischen zwei Klassen versteht, dann bedeutet dies, daß in vielen Fällen eine Abbildung von Klasse A nach Klasse B und eine Abbildung von B nach A invers zueinander sein sollen.

Sei $R \subseteq A \times B$ eine binäre Relation zwischen Klassen A und B.
R entspricht zwei Relationships R_A und R_B, d.h. Abbildungen
$R_A: A \rightarrow 2^B$ und $R_B: B \rightarrow 2^A$.
Daß R_A und R_B invers zueinander sind, entspricht dann formal folgenden Invarianten:

$$\forall x \in A: \; x \in \bigcup_{y \in R_A(x)} R_B(y) \quad \text{und} \quad \forall y \in B: \; y \in \bigcup_{x \in R_B(y)} R_A(x)$$

oder dazu äquivalent:
$R_A(x) = \{y \in B \mid (x,y) \in R\}$ und $R_B(y) = \{x \in A \mid (x,y) \in R\}$.

Beispiele:

1) Gegeben seien zwei Klassen
   Männer mit “relationship Frauen Ehefrau" und
   Frauen mit “relationship Männer Ehemann".
   Dann soll immer gelten: $\forall$ m $\in$ Männer : m.Ehefrau.Ehemann = m und
   $\forall$ f $\in$ Frauen: f.Ehemann.Ehefrau = f.
2) $\forall$ s $\in$ Straße $\forall$ a $\in$ s.HatAbschnitte : a.GehörtZuStr = s
   und
   $\forall$ a $\in$ Abschnitte $\forall$ s $\in$ a.GehörtZuStr : a $\in$ s.HatAbschnitte

Diese Art von Integritätsbedingungen kann wie folgt spezifiziert werden:

1)
```
class Männer {
    ...
    relationship Frauen Ehefrau inverse Frauen::Ehemann;
    }
class Frauen {
    ...
    relationship Männer Ehemann inverse Männer::Ehefrau;
    }
```
2)
```
class Straße {
    ...
    relationship Set<Abschnitt> HatAbschnitte inverse Abschnitt::GehörtZuStr;
    }
class Abschnitt {
    ...
    relationship Straße GehörtZuStr inverse Straße::HatAbschnitte;
    ...
    }
```

Die Spezifikation von inversen Relationships erlaubt die Gewährleistung der referentiellen Integrität durch das DBS. Wenn z.B. eine Straße gelöscht wird, müssen die Relationships “GehörtZuStr“ ihrer Abschnitte in Nullzeiger geändert werden (oder diese Abschnitte müssen sogar ebenfalls gelöscht werden).

Objekt-Sharing

Ein Objekt kann von mehreren Objekten referenziert werden. Ein solches Objekt wird als “shared object" bezeichnet.

Beispiel:
Nehmen wir an, verschiedene Straßen können gemeinsame Abschnitte haben (z.B. teilen sich die Mainzer Straße in Saarbrücken und die Bundesstraße B51 einige Abschnitte). Um dies zu erlauben ist lediglich die Definition der Klasse “Abschnitt" wie folgt zu ändern:

```
class Abschnitt {
    ...
    relationship Set<Straße> GehörtZuStr inverse Straße::HatAbschnitte;
    ... }
```

Änderungen, die auf einem “shared object" über einen bestimmten Referenzpfad durchgeführt werden, sind für alle referenzierenden Objekte gleichzeitig sichtbar.
Beispiel:
Nehmen wir stark vereinfachend an, daß der erste Abschnitt der Mainzer Str. gleichzeitig der letzte Abschnitt der Bundesstraße B51 sei. Dieser Abschnitt soll von 2 auf 4 Spuren verbreitert werden.
Die Wirkung der Änderungsoperation

```
Straße [StrBez="Mainzer Str."].HatAbschnitte.first().AnzSpuren = 4
```

wird auch für die B51 unmittelbar wirksam, z.B. in der darauffolgenden Query

```
Straße [StrBez="B51"].HatAbschnitte.last().AnzSpuren,
```

die den Wert 4 zurückliefert.

Objekt-Identität

Es kann verschiedene Objekte geben, die in allen Attributwerten übereinstimmen. Dennoch sind diese Objekte über ihre *Objekt-ID* (OID) eindeutig unterscheidbar. Zwei Referenzen auf je eines der beiden Objekte sind auf Gleichheit testbar; der Test liefert in diesem Fall “false" als Resultat.

Beispiel:
Es könnte zwei Objekte in der Klasse “Punkt" geben, die in ihren Koordinaten übereinstimmen. Wenn sie aber zu verschiedenen Polylinien gehören und eine der beiden Linien wird verschoben, wird dabei nur das eine Punkt-Objekt geändert.

10.2.2 Objektmethoden und Kapselung (Verhaltensmäßige Objektorientierung)

Für jede Klasse kann eine Menge von *Methoden* definiert werden. Dies sind Funktionen und Änderungsoperationen, die auf jedes Objekt der Klasse anwendbar sind. Zusätzlich zu dem Objekt, auf das eine Methode angewandt wird, kann eine Methode weitere Parameter (sowohl Werte als auch Objekte) haben und auch ein Funktionsresultat zurückliefern.

Die Signatur einer Methode m ist eine Liste von Typen $<T_1, ..., T_n, T_{n+1}>$ derart, daß die Methode n Parameter der Typen $T_1, .., T_n$ und ein Resultat des Typs T_{n+1} hat. In Anlehnung an algebraische Strukturen wird auch wie folgt geschrieben: m: $T_1 \ ... \ T_n \rightarrow T_{n+1}$.

Der Aufruf einer Methode für ein Objekt x wird wie der Zugriff auf ein Attribut von x geschrieben, nämlich in der Form x.m oder x->m (bzw. x.m(...) oder x->m(...)). Damit kann auf gespeicherte Daten und (mittels Methoden) berechnete Daten einheitlich zugegriffen werden.

Zur Schnittstellendefinition einer Klasse gehört nur die Angabe der Namen und Signaturen von Methoden. Die Implementierung einer Methode ist nach außen hin verborgen. Eine Klasse bildet somit einen *Abstrakten Datentyp (ADT)*, also ein Modul, das Basisoperationen auf einer Menge von Objekten (gleichen Typs) anbietet und das die Implementierung der Operationen sowie z.T. auch die zugrundeliegenden Datenstrukturen für die Objekte verbirgt. Durch dieses “Information-Hiding"-Prinzip wird eine hohe Modularität errreicht; große Systeme werden leichter wartbar. Man spricht in diesem Kontext auch von “*gekapselten Objekten*": die Implementierung der Methoden ist austauschbar, ohne die Schnittstelle zu ändern.

Zu den für eine Klasse angebotenen Methoden gehört insbesondere immer eine Methode zur Erzeugung und evtl. Initialisierung neuer Objekte der Klasse (sog. “Konstruktoren"). Diese Methode trägt üblicherweise denselben Namen wie die Klasse selbst und wird häufig gar nicht explizit angegeben.

Zur Implementierung der Methoden können die Objekte einer Klasse weitere Attribute haben, die nach außen nicht sichtbar sind. Diese Attribute heißen “private" (engl.: private, hidden) Attribute, die an der Schnittstellen sichtbaren Attribute heißen “öffentliche" (engl.: public) Attribute.

Beispiele:

1) ADT “Elektronisches Telefonbuch":
   Ein elektronisches Telefonbuch bietet Methoden wie das Nachschlagen der Telefonnummer zu einem gegebenen Namen und einer Adresse an. Die zugrundeliegende Datenstruktur könnte ein Array mit sortierten Einträgen, eine Hashtabelle oder ein Suchbaum sein. Da die Datenstruktur und die Implementierung der Methoden auf der jeweiligen Datenstruktur gekapselt - also nach außen nicht sichtbar - sind, läßt sich die Implementierung jederzeit austauschen, ohne daß sonstiger Programmcode, der die Methoden des ADTs Telefonbuch benutzt, geändert werden muß.

2) Straßen und Straßenabschnitte:

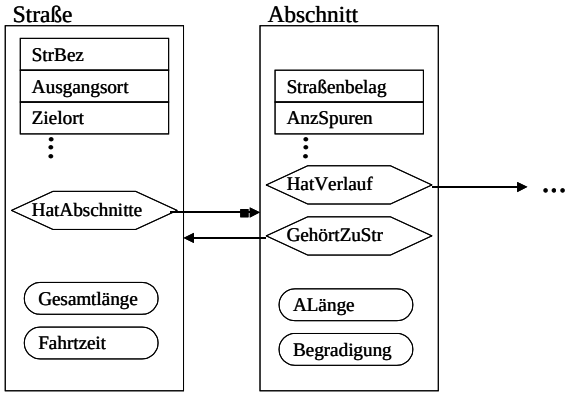
```
class Straße {
        ...
        private attribute Integer Durchschnittstempo;
        Real Gesamtlänge ( );
        Integer Fahrtzeit ( );
        }
class Abschnitt {
        ...
        Real ALänge ( );
        Boolean Begradigung (in Punkt, in Punkt);
        }
```

mit der folgenden Implementierung für Gesamtlänge und ALänge:

```
Straße:: Gesamtlänge () {
        float l = 0.0;
        Set<Ref<Abschnitt>> MeineAbschnitte = this->HatAbschnitte;
        Iterator<Abschnitt> it = MeineAbschnitte->create_iterator();
        Ref<Abschnitt> a;
        while (a=it.next())
                { l += a->ALänge(); }
        return l };
```

```
Abschnitt:: ALänge () {
        float l = 0.0;
        List<Ref<Punkt>> MeinePunkte = this->HatVerlauf->HatStützpunkte;
        Iterator<Punkt> it = MeinePunkte->create_iterator();
        Ref<Punkt> p, q;
        p = it->next();
        while (q=it->next()) {
                l += sqrt(    (p->X - q->X) * (p->X - q->X) +
                              (p->Y - q->Y) * (p->Y - q->Y));
                p = q; };
        return l };
```

Graphische Illustration:



Vorteil der Kapselung:

Nehmen wir an, die Implementierung der Längenberechnung soll wie folgt geändert werden. Zwischen je zwei aufeinanderfolgenden Stützpunkten soll der Verlauf durch einen Kreisbogen interpoliert werden. Dazu soll zusätzlich zu den Stützpunkten die Steigung einer durch den ersten Stützpunkt gehenden Tangente an das erste Kreisbogenstück gespeichert werden. Damit sind alle Kreisbogenstücke eindeutig bestimmt.

Um diese Änderung zu realisieren, muß die Klasse Polylinie um ein zusätzliches Attribut “Tangentensteigung" erweitert werden, und die Implementierung der Methode “ALänge" ist zu ändern. Die Schnittstellen der Klassen “Abschnitt" und “Straße" aber bleiben unverändert.

Analoge Überlegungen gelten beispielsweise auch für eine Umstellung der Punktkoordinaten von kartesischen Koordinaten auf Polarkoordinaten usw.

10.2.3 Vererbung

Eine Klasse B heißt *Subklasse* einer Klasse A und A heißt *Superklasse* von B, wenn

- die Attribute und Methoden von B eine (evtl. unechte) Obermenge der Attribute und Methoden von A sind und
- die Objektmenge von B eine (evtl. unechte) Teilmenge der Objektmenge von A ist.

Formal gilt also die Integritätsbedingung:
$\forall b \in B\ \exists a \in A$: b.sch(A)=a.sch(A)

B wird als *Spezialisierung* von A und A als *Generalisierung* von B bezeichnet. Man sagt auch, daß B die Attribute und Methoden von A “erbt". Dadurch werden Schemadefinition und Methoden der Klasse A in der Klasse B wiederverwendet.
Zusätzlich können für B weitere Attribute und Methoden definiert werden.

Die Generalisierungs-/Spezialisierungsbeziehung zwischen Klassen bildet eine partielle Ordnung, und man spricht auch von einer Generalisierungshierarchie oder einer Vererbungshierarchie.

Ein Objekt einer Subklasse kann überall dort verwendet werden, wo ein Objekt der entsprechenden Superklasse verwendet werden darf (Substitutionsprinzip).

Beispiele:

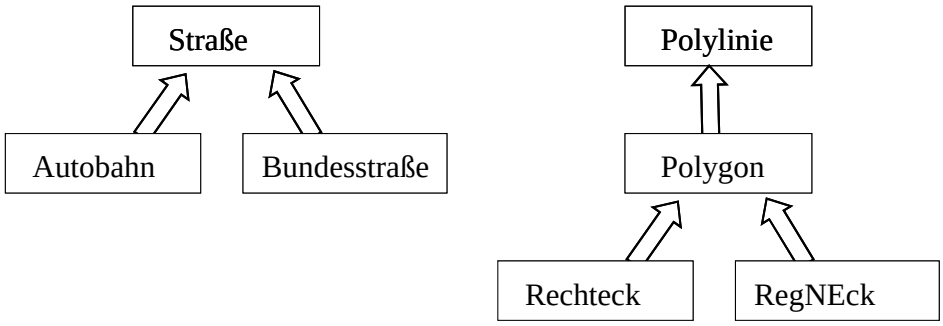
1) Spezialisierung von Straßen in Bundesstraßen und Autobahnen:

```
class Bundesstraße: Straße { extent Bundesstraßen;
        attribute Integer Verkehrsdichte;
        }

class Autobahn: Straße { extent Autobahnen;
        attribute Integer Mindesttempo;
        attribute Array<Integer> VerkehrsdichteProTag;
        relationship Set<Punkt> Auffahrten;
        }
```

2) Spezialisierung von Polylinien in Polygone und weitergehende Spezialisierungen (Rechtecke, regelmäßige N-Ecke):

```
class Polygon: Polylinie { extent Polygone;
        relationship Punkt Zentrum;
        relationship Stadt BeschreibtS inverse Stadt::Stadtgrenze;
        relationship Land BeschreibtL inverse Land::Landesgrenze;
        Real Fläche ( );
        }

class Rechteck: Polygon { extent Rechtecke;
        attribute Real Diagonallänge;
        }

class RegNEck: Polygon { extent RegNEcke;
        attribute Integer AnzEcken;
        Real Inkreisradius ();
        Real Umkreisradius ();
        }
```

Graphische Darstellung:



Die Subklassen einer Superklasse müssen weder disjunkt sein noch müssen sie zusammen eine Überdeckung der Superklasse bilden.
Beispiele:

1) Es gibt Straßen, die weder Bundesstraßen noch Autobahnen sind.
2) Quadrate sind sowohl Rechtecke als auch regelmäßige N-Ecke.

Überschreiben von Methoden (Overriding)

Die von einer Superklasse geerbten Methoden können in einer Subklasse anders implementiert werden. Dies kann aufgrund der besonderen “Semantik" der Subklassenobjekte oder auch aus Effizienzgründen sinnvoll sein.

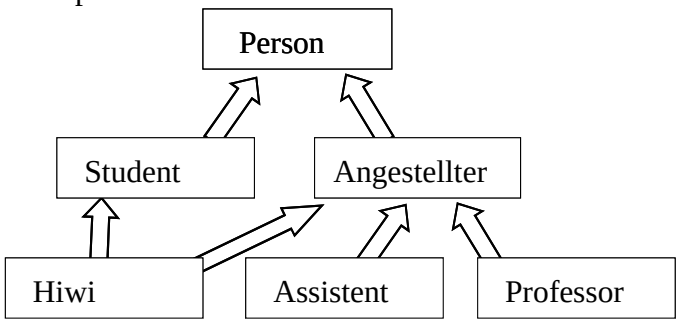
Beispiele:

1) Spezialisierung von Polygonen in Polygone mit Löchern (z.B. Seen mit Inseln):
   Die Methode “Fläche" sollte überschrieben werden, indem z.B. die Fläche der Löcher abgezogen wird.
2) Spezialisierung von Polygonen in Rechtecke und RegNEcke:
   Die Methode “Fläche" könnte überschrieben werden, um die Flächenberechnung effizienter durchzuführen.
3) Spezialisierung von Polylinien in Polygone:
   Die Methode “SchneidetBox (in Rechteck)", die testet, ob eine Polylinie ein gegebenes Rechteck schneidet, sollte überschrieben werden. Ein Polygon schneidet ein Rechteck auch, wenn kein einziger Punkt seines Randes im Rechteck liegt, nämlich wenn das Rechteck vollständig im Polygon liegt.

Mehrfachvererbung

Eine Klasse kann mehrere Superklassen haben. Bei der Vererbung müssen dann ggf. Namenskonflikte bzgl. der geerbten Attribute oder Methoden durch Umbenennung gelöst werden.

Beispiel:



wobei sowohl die Klasse “Student" als auch die Klasse “Angestellter" jeweils ein Attribut “Wochenpensum" haben.

Eines der beiden von Hiwi-Objekten geerbten Attribute “Student.Wochenpensum" und “Angestellter.Wochenpensum" muß umbenannt werden, z.B. das von Student geerbte Attribut in “Hiwi.Vorlesungsstunden".

Verarbeitung polymorpher Objektmengen

Die Objekte einer Klasse mit einer oder mehreren Subklassen haben eine heterogene Struktur. Dennoch können sie bzgl. der gemeinsamen Attribute und Methoden einheitlich verarbeitet werden. Wenn jedoch Methoden in den Subklassen überschrieben worden sind, sind mit einem Methodennamen eigentlich verschiedene Implementierung der Methodenausführung verbunden.
Die Methode heißt dann auch “überladen" (engl.: overloaded).

Beispiel:

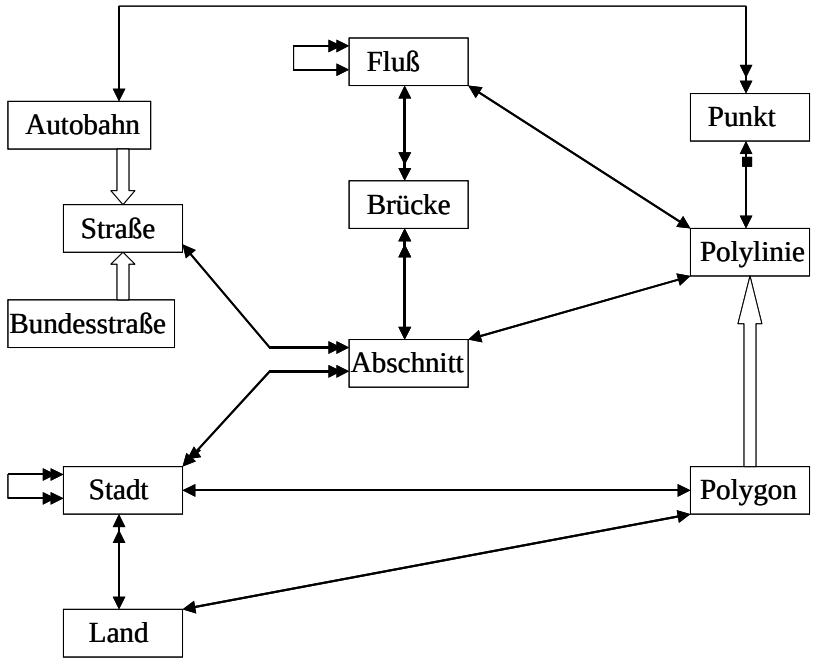
Für eine Menge von Objekten der Klasse “Polylinie" soll mittels der Methode “SchneidetBox" getestet werden, welche Polylinien ein gegebenes Rechteck schneiden.
Da die Klasse “Polylinie" auch Polygone enthält, für die die Methode “SchneidetBox" überschrieben wurde, müssen zwei verschiedene Methodenimplementierungen aufgerufen werden, je nachdem, ob es sich um ein Polygon oder eine offene Polylinie handelt. Damit bei der Verarbeitung solcher polymorpher Objektmengen jeweils die für jede Subklasse passende Methodenimplementierung verwendet wird, darf der Code für überschriebene Methoden nicht statisch gebunden werden. Vielmehr entscheidet es sich erst zur Laufzeit von Objekt zu Objekt, welche Implementierung je nach Subklassenzugehörigkeit verwendet werden muß. Diese dynamische Festlegung der passenden Methodenimplementierung heißt “*dynamisches Binden*" oder “*spätes Binden*" (engl.: late binding).

10.3 Schemabeschreibung nach dem ODMG-Standard

Die Object Database Management Group (ODMG), ein Herstellerkonsortium, hat einen Standard zur objektorientierten Schemabeschreibung entwickelt, der eine Schemabeschreibungssprache ODL (Object Definition Language) beinhaltet, sowie einen Standard für eine objektorientierte Anfragesprache OQL (Object Query Language) und einen Vorschlag für deren Einbettung in C++ und Smalltalk. Dieser Standard wird u.a. von dem objektorientierten Datenbanksystem O2 weitestgehend unterstützt.

ODL bietet im wesentlichen die in Abschnitt 8.2 vorgestellten Grundkonzepte in syntaktisch ausgefeilter Form an. ODL ist eine Erweiterung der Interface Definition Language (IDL) der Object Management Group (OMG).

Beispiel:



Bemerkungen:
Dieses Beispiel bringt auch einige Schwachstellen des ODMG-Modells zu Tage:

- Es fehlen Möglichkeiten, explizite Integritätsbedingungen zu spezifizieren.
  Beispielsweise müßte man eigentlich fordern, daß alle Partnerstädte einer Stadt in einem anderen Land liegen, und zwar in paarweise verschiedenen Ländern.
  Ein weiteres Beispiel wäre, daß man nicht wirklich erzwingen kann, daß bei Polygonen der erste und der letzte Stützpunkt übereinstimmen.
- Ternäre (oder noch höherstellige) Beziehungen sind nicht ausdrückbar.
  Beispielsweise kann man eine Beziehung zwischen zwei Städten und einer Straße, die die jeweils schnellste Verbindung zwischen den Städten angibt, nicht sauber modellieren.

Ausführliche Modellierung des Beispiels in ODL:

```
class Straße { extent Straßen;
        key StrBez;
        attribute String StrBez;
        attribute String Ausgangsort;
        attribute String Zielort;
        relationship Set<Abschnitt> HatAbschnitte inverse Abschnitt::GehörtZuStr;
        Real Gesamtlänge ( );
        Integer Fahrtzeit ( );
        }
class Bundesstraße: Straße { extent Bundesstraßen;
        attribute Integer Verkehrsdichte;
        }
class Autobahn: Straße { extent Autobahnen;
        attribute Integer Mindesttempo;
        attribute Array<Integer> VerkehrsdichteProTag;
        relationship Set<Punkt> Auffahrten;
        }
class Abschnitt { extent Abschnitte;
        key (GehörtZuStr.StrBez, ANr);
        attribute Integer ANr;
        attribute String Straßenbelag;
        attribute Integer AnzSpuren;
        attribute Integer Tempolimit;
        relationship Polylinie HatVerlauf inverse Polylinie::BeschreibtA;
        relationship Straße GehörtZuStr inverse Straße::HatAbschnitte;
        relationship Set<Brücke> HatBrücken inverse Brücke::GehörtZuA;
        relationship Set<Stadt> FührtDurch inverse Stadt::LiegtAn;
        Real ALänge ( );
        }
class Polylinie { extent Polylinien;
        attribute Integer AnzPunkte;
        relationship List<Punkt> HatStützpunkte inverse Punkt::GehörtZuLinie;
        relationship BeschreibtA inverse Abschnitt::HatVerlauf;
        relationship BeschreibtF inverse Fluß::HatVerlauf;
        Real Länge ( );
        void Begradigung (in Punkt, in Punkt) Raises (Zu_große_Anschlußkrümmung);
        Boolean Schneidet (in Polylinie);
        Boolean SchneidetBox (in Rechteck);
        Boolean LiegtInBox (in Rechteck);
        }
class Punkt { extent Punkte;
        attribute Real X;
        attribute Real Y;
        relationship Polylinie GehörtZuLinie inverse Polylinie::HatStützpunkte
        Boolean LiegtInBox (in Rechteck);
        Boolean LiegtInRegion (in Polygon);
        }
```

```
class Polygon: Polylinie { extent Polygone;
        relationship Punkt Zentrum;
        relationship Stadt BeschreibtS inverse Stadt::Stadtgrenze;
        relationship Land BeschreibtL inverse Land::Landesgrenze;
        Real Fläche ( );
        }
class Fluß { extent Flüsse;
        key Flußbez;
        attribute String Flußbez;
        attribute Array<Real> Verschmutzungsgrad;
        relationship Punkt Quelle;
        relationship Punkt Mündung;
        relationship Fluß MündetIn inverse Fluß::HatZuflüsse;
        relationship Set<Fluß> HatZuflüsse inverse Fluß::MündetIn;
        relationship Polylinie HatVerlauf inverse Polylinie::BeschreibtF;
        relationship Set<Brücke> FließtUnter inverse Brücke::ÜberquertF;
        }
class Brücke { extent Brücken;
        attribute Real Höhe;
        attribute Real Spannweite;
        relationship Punkt Anfang;
        relationship Punkt Ende;
        relationship Fluß ÜberquertF inverse Fluß::FließtUnter;
        relationship Abschnitt GehörtZuA inverse Abschnitt::HatBrücken;
        }
class Stadt { extent Städte;
        attribute String Stadtname;
        attribute Integer Einwohnerzahl;
        relationship Person Bürgermeister;
        relationship Set<Stadt> Partnerstädte inverse Stadt::Partnerstädte;
        relationship Land GehörtZuL inverse Land::HatStädte;
        relationship Polygon Stadtgrenze inverse Polygon::BeschreibtS;
        relationship Set<Abschnitt> LiegtAn inverse Abschnitt::FührtDurch;
        Real Einwohnerdichte ( );
        void Eingemeindung (Inout Stadt);
        }
class Land { extent Länder;
        key Landesbez;
        attribute String Landesbez;
        relationship Person Staatsoberhaupt;
        relationship Set<Stadt> HatStädte inverse Stadt::GehörtZuL;
        relationship Polygon Landesgrenze inverse Polygon::BeschreibtL;
        Integer Bevölkerung ( );
        }
```

Das ODMG-Modell sieht darüber hinaus eine Reihe von vordefinierten, parameterisierten Klassen (Typgeneratoren, sog. “Templates") vor, von denen neu definierte Klassen geeignet erben können. Dies sind vor allem die Klasse “Collection<T>" mit einem Klassenparameter T sowie deren Subklassen Set<T>, Bag<T>, List<T> und Array<T>. Durch Einsetzen einer konkreten Klasse für den Parameter T entsteht eine Klasse, die alle Methoden der entsprechenden Kollektionsklasse erbt.

Die wichtigsten Attribute und Methoden der parametrisierten Kollektionsklassen sind:

für Collection<T>:

```
attribute Integer cardinality;
attribute Boolean empty?;
Collection<T> create ();
void delete ();
void insert_element (in T);
void remove_element (in T);
Collection<T> select (in String);  // mit einem Prädikat als Parameter vom Typ String
Boolean exists? (in String);
Boolean contains_element? (in T);
Iterator create_iterator ();
...
```

für Set<T>:

```
Set<T> union (in Set<T>);
Set<T> intersection (in Set<T>);
Set<T> difference (in Set<T>);
Boolean is_subset? (in Set<T>);
Boolean is_proper_subset? (in Set<T>);
Boolean is_superset? (in Set<T>);
Boolean is_proper_superset? (in Set<T>);
...
```

für List<T>:

```
T retrieve_first_element ();
T retrieve_last_element ();
T retrieve_element_at (in Integer);  // mit einer Listenposition als Parameter
void insert_first_element (in T);
void insert_last_element (in T);
void insert_element_after (in T, in Integer);
void insert_element_before (in T, in Integer);
...
```

Für den Datenteil von ODL (also ohne Methoden) ließe sich eine formale Semantik wie folgt entwickeln:

- Jede Klasse entspricht einer Relation, Multirelation oder geordneten Relation.
  Anders als im Relationenmodell können die zugrundeliegenden Domains selbst wieder Mengen, Multimengen oder Listen sein (und deren Basisdomains auch wieder usw.).
  Die Typkonstruktoren Record, Set, Bag und List können also beliebig geschachtelt werden.
- Jede Relationship zwischen Klassen A und B ist eine weitere Relation, ggf. mit der Einschränkung, daß zu jedem $x \in A$ höchstens ein $y \in B$ gehört.
- Für jedes Paar inverser Relationships wird eine Integritätsbedingung im Stil von Kapitel 8.2.1 definiert.
- Für jede Subklassenbeziehung wird eine Integritätsbedingung im Stil von Kapitel 8.2.3 definiert.

10.4 Objektorientierte Anfragesprachen

Die Sprache OQL (Object Query Language)

OQL ist die Anfragesprache des ODMG-Standards. Sie wird in ihren wesentlichen Elementen u.a. vom objektorientierten Datenbanksystem O2 unterstützt.

OQL folgt der syntaktischen Grundidee von SQL, indem es einen SELECT-FROM-WHERE-Block unterstützt. Darüber hinaus aber trägt OQL den Besonderheiten des ODMG-Modells Rechnung, indem es vor allem die Traversierung von Relationships (Objektreferenzen) auf einfache Weise ermöglicht und Methodenaufrufe in Queries erlaubt.

Außerdem ist es möglich, SELECT-FROM-WHERE-Blöcke auch in der SELECT-Klausel oder der FROM-Klausel zu schachteln (wohingegen in SQL diese Schachtelung nur in der WHERE-Klausel erlaubt ist, und dies auch nur in eingeschränkter Form).

Die Schachtelung in der WHERE-Klausel entspricht der von SQL.

Die Schachtelung in der FROM-Klausel entspricht einer Substitution im Sinne von Anfragen auf Views.

Die Schachtelung in der SELECT-Klausel konstruiert geschachtelte Ergebnisse, z.B. Tupelmengen mit Komponenten, die selbst Listen, Mengen oder Multimengen von Tupeln sein können (oder als Basistyp sogar ebenfalls einen Collection-Typ haben). Wenn geschachtelte Collections den Record-Konstruktor nur auf der tiefsten Ebene verwenden (und ansonsten eben nur die List-, Set- oder Bag-Konstruktoren), können sie mit der Funktion FLATTEN in eine einfache (d.h. nicht geschachtelte) Liste, Menge oder Multimenge von Tupeln konvertiert werden.

Traversierung von Relationships

Ein Ausdruck der Form C.R mit einem Klassennamen C (bzw. einer entsprechenden Objektvariablen, dem Pendant zu einer Tupelvariablen) und einer Relationship R von der Klasse C zur Klasse Z kann überall verwendet werden, wo der Ausdruck Z verwendet werden darf. Die Bedeutung des Ausdrucks C.R ist die Menge der Objekte der Klasse Z, die von der - ggf. durch ein Prädikat eingeschränkten - Klasse C aus über die Relationship R erreichbar sind.

Einschränkungen einer Klasse C, also die Selektion von Teilmengen, erfolgen über die WHERE-Klausel oder in der Form C[Prädikat].

Damit können ganze *Pfadausdrücke* der Form $C_0.R_1.R_2. \ldots .R_n.A$ gebildet werden, wobei

- R_1 eine Relationship von der Klasse C_0 zur Klasse C_1 ist,
- R_i eine Relationship von der Klasse C_{i-1} zur Klasse C_i ist und
- A ein Attribut der Klasse C_n ist.

Pfadausdrücke erlauben es, “implizite Joins" auf einfache Weise auszudrücken.
Ein Pfadausdruck $C_0.R_1.R_2. \ldots .R_n.A$ in der SELECT-Klausel steht für

$\{tn.A, \ldots \mid \ldots \wedge \exists\, t0\, \exists\, t1 \ldots \exists\, t(n-1)\ (t0.R1 = t1.OID \wedge t1.R2 = t2.OID \wedge \ldots \wedge t(n-1).Rn = tn.OID)\}$,
und ein Ausdruck $C_0.R_1.R_2. \ldots .R_n.A\ \theta$ wert in der WHERE-Klausel steht für

$\exists\, t0\, \exists\, t1 \ldots \exists\, t(n-1)\ (t0.R1 = t1.OID \wedge t1.R2 = t2.OID \wedge \ldots \wedge t(n-1).Rn = tn.OID \wedge tn.A\ \theta\ wert)$.

Beispiele:

1) Welche Flüsse überquert die B51?

```
SELECT FLATTEN (S.HatAbschnitte.HatBrücken.ÜberquertF)
FROM Straßen S
WHERE S.StrBez="B51"
```

oder:

```
SELECT B.ÜberquertF
FROM (    SELECT FLATTEN(A.HatBrücken)
          FROM (    SELECT FLATTEN (S.HatAbschnitte)
                    FROM S IN Straßen
                    WHERE S.StrBez="B51"
                ) AS A
      ) AS B
```

2) Welche Straßen überqueren die Mosel?

```
SELECT S
FROM Straßen S
WHERE S.HatAbschnitte.HatBrücken.ÜberquertF.Flußbez="Mosel"
```

3) Welche Partnerstädte haben die Städte der Elfenbeinküste?

```
SELECT FLATTEN(L.HatStädte.Partnerstädte)
FROM Länder L
WHERE L.Landesbez="Elfenbeinküste"
```

4) In welchen Ländern haben Städte der Elfenbeinküste Partnerstädte?

```
SELECT DISTINCT FLATTEN ( L.HatStädte.Partnerstädte.GehörtZuL )
FROM Länder L
WHERE L.Landesbez="Elfenbeinküste"
```

5) Ausgabe des Verlaufs der B51:

```
SELECT S.HatAbschnitte.HatVerlauf.HatStützpunkte
FROM Straßen S
WHERE S.StrBez="B51"
```

oder:

```
SELECT A.HatVerlauf.HatStützpunkte
FROM  (   SELECT S.HatAbschnitte
          FROM Straßen S
          WHERE S.StrBez="B51"
          SORT A IN S.HatAbschnitte BY A.ANr ) AS A
```

Einbezug von Methodenaufrufen

Ein Ausdruck der Form C.M(...) mit einem Klassennamen C und einer Methode M der Klasse C kann überall verwendet werden, wo - unter Berücksichtigung des Resultattyps von M - Attribute von C verwendet werden dürfen.

Beispiele:

1) Wieviele Kilometer hat das gesamte Straßennetz?

```
SELECT SUM ( S.Gesamtlänge() )
FROM Straßen S
```

2) Welche Städte haben eine Fläche von mehr als 30 Quadratkilometern?

```
SELECT S.Stadtname
FROM Städte S
WHERE S.Stadtgrenze.Fläche() > 30
```

3) In welchen Städten mit einer Fläche von mehr als 30 Quadratkilometern haben alle durch die Stadt führenden Straßenabschnitte ein Tempolimit von nicht mehr als 80 km/h?

```
SELECT S.Stadtname
FROM Städte S
WHERE S.Stadtgrenze.Fläche() > 30
AND ( FOR ALL A IN S.LiegtAn : A.Tempolimit <= 80 )
```

4) In welchen Ländern mit mehr als 100 Millionen Einwohnern haben Städte der Elfenbeinküste Partnerstädte?

```
SELECT DISTINCT PL
FROM ( SELECT FLATTEN (L.HatStädte.Partnerstädte.GehörtZuL)
            FROM L IN Länder
            WHERE L.Landesbez="Elfenbeinküste" ) AS PL
WHERE PL.Bevölkerung() > 100000000
```

5) Bestimmung aller Straßen mit einem Stützpunkt innerhalb des Rechtecks mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8)

```
SELECT S.StrBez
FROM Straßen S
WHERE   S.HatAbschnitte.Verlauf.HatStützpunkte.LiegtInBox (
            STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

6) Bestimmung aller Straßen, die das Rechteck mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8) schneiden

```
SELECT S.StrBez
FROM Straßen S
WHERE   S.HatAbschnitte.Verlauf.SchneidetBox (
            STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

Berücksichtigung der Generalisierungshierarchie

Eine Query über einer Klasse C wird über C und allen Subklassen von C ausgewertet, sofern darin nur Attribute und Methoden von C verwendet werden.

Durch explizite “Klassenindikatoren" kann eine Anfrageauswertung auch auf bestimmte Subklassen beschränkt werden.

Beispiele:

1) Welche Polylinien, unter Einbeziehung von Polygonen, schneiden das Rechteck mit der linken unteren Ecke (1,2) und der rechten oberen Ecke (5,8)?

```
SELECT P
FROM Polylinien P
WHERE P.SchneidetBox (
        STRUCT(lu:STRUCT(x:1,y:2), ro:STRUCT(x:5,y:8)) )
```

2) Welches ist das niedrigste Mindesttempo auf einer Straße von mehr als 1000 Kilometern Gesamtlänge?
(Annahme: Nur Autobahnen haben eine solche Länge.)

```
SELECT MIN ( ((Autobahn) S).Mindesttempo )
FROM Straßen S
WHERE S.Gesamtlänge() > 1000
```

3) Auf welchen Autobahnen, die die Mosel überqueren, gibt es einen Abschnitt mit einem Tempolimit von 60 km/h ?

```
SELECT A.GehörtZuStr.StrBez
FROM (     SELECT F.FließtUnter.GehörtZuA
           FROM Flüsse F
           WHERE F.Flußbez = “Mosel" ) AS A
WHERE ( EXISTS B IN Autobahnen: B.StrBez = A.GehörtZuStr.StrBez )
AND A.Tempolimit = 60
```

oder:

```
SELECT ((Autobahn) A.GehörtZuStr).StrBez
FROM (     SELECT F.FließtUnter.GehörtZuA
           FROM Flüsse F
           WHERE F.Flußbez = “Mosel" ) AS A
WHERE A.Tempolimit = 60
```

Gruppierung und Aggregation

Gruppierungen führen auf geschachtelte Objekte (mit mengenwertigen Attributen), die eleganter als in SQL weiterverarbeitet werden können.

Das Resultat von
... GROUP BY G1: Gruppierungsattribut1, ...
hat den Typ
set< struct(G1: type_of(Gruppierungsattribut1), ...),
partition: bag<type_of(Nichtgruppierungsattribute)>>.

Aggregationsfunktionen können auch benutzerdefinierte Methoden sein.

Beispiele:

1) Ausgabe der Gesamteinwohnerzahl aller Städte, die an der E12 liegen, gruppiert nach Ländern.

```
SELECT   Land,
         Gesamteinwohner: SUM (SELECT R.Einwohnerzahl FROM partition),
         Stadteinwohner: (SELECT R.Stadtname, R. Einwohnerzahl FROM partition)
FROM (   SELECT S
         FROM Städte S
         WHERE S.LiegtAn.GehörtZu.StrBez = “E12" )
      AS R
GROUP BY Land: R.GehörtZuL
```

Resultat:

Land	*Gesamteinwohner*	*Stadteinwohner*	
Deutschland	450 000	*Stadtname*	*Einwohnerzahl*
		Kaiserslautern	200 000
		Saarbrücken	250 000
Frankreich	5 350 000	*Stadtname*	*Einwohnerzahl*
		Metz	200 000
		Reims	150 000
		Paris	5 000 000

2) Sei Bestellungen eine Objektklasse mit Attributen BestNr, KNr, PNr, Summe, ... Sei ferner Zahlen eine folgendermaßen definierte Klasse (ohne Extent) mit Funktionen für die Berechnung des Mittelwerts, des Medians und der Standardabweichung einer Multimenge von reellen Zahlen:

```
class Zahlen {
        attribute bag<Zahl: real> Zahlenmultimenge;
        Real mean();
        Real median();
        Real stddev();
        }
```

a) Ausgabe von Mittelwert und Standardabweichung der Bestellungssummen.

```
SELECT Z.mean(), Z.stddev()
FROM (    SELECT B.Summe FROM Bestellungen B ) AS Z
```

b) Ausgabe von Mittelwert und Standardabweichung der Bestellungssummen für jedes einzelne Produkt.

```
SELECT    G.Prod,
          G.Best.mean(), G.Best.stddev()
FROM (    SELECT Prod, (SELECT B.Summe FROM partition) AS Best
          FROM Bestellungen B
          GROUP BY Prod: B.PNr ) AS G
```

10.5 Objektorientierte Erweiterungen von SQL und “Objektrelationale" Datenbanksysteme

Wesentliche Konzepte objektorientierter Datenmodelle und Anfragesprachen werden zunehmend auch in relationale DBS integriert und führen zu sogenannten objektrelationalen Datenbanksystemen. Insbesondere wird die Sprache SQL in diese Richtung weiterentwickelt; der Standard SQL-99 reflektiert dies bereits.

10.5.1 Komplexe Datentypen (mit Typhierarchie)

... in SQL-99

Die Typkonstruktoren ROW, SET, LIST und MULTISET sind orthogonal kombinierbar zur Konstruktion komplexer Datentypen. Mit CREATE TABLE wird eine Ausprägung eines Typs “MULTISET OF ..." angelegt.

Mit der UNDER-Klausel können Typspezialisierungen vorgenommen werden. Bei Anlegung entsprechender Tabellen bezieht die allgemeinere Tabelle die speziellere mit ein, bildet also eine Obermenge (bzw. eigentlich Ober-Multiset).

Beispiele:

```
CREATE ROW TYPE Punkt (X FLOAT, Y FLOAT);
CREATE ROW TYPE Polylinie
   (AnzPunkte INTEGER,
   HatStützpunkte LIST OF Punkt);
CREATE ROW TYPE Straße
   (StrBez VARCHAR(10),
   Ausgangsort VARCHAR(30), Zielort VARCHAR(30),
   HatVerlauf Polylinie);
CREATE TABLE Straßen OF TYPE Straße;

CREATE TYPE Autobahn UNDER Straße
  (Mindesttempo FLOAT);
CREATE TABLE Autobahnen OF Autobahn;

SELECT * FROM Autobahnen
   liefert alle Autobahnen
SELECT * FROM Straßen
   liefert alle Straßen inklusive der Autobahnen
```

... in Oracle

In Oracle (ab Version 8) werden ebenfalls Typ- und Datendefinition getrennt. Die Komponenten eines neu definierten Typs können von beliebigem Typ sein, so daß auf diese Wiese geschachtelte Relationen oder andere komplexe Datentypen spezifizierbar sind. Bei Angabe von „AS OBJECT“ in der Typdefinition sind die Instanzen des Typs eigenständige Objekte, für die das DBS automatisch eine OID erzeugt; solche Objekte können mit REF von verschiedenen Stellen als „shared (sub-) objects“ referenziert werden. In erster Näherung entspricht der OBJECT-Konstruktor dem ROW-Konstruktor von SQL-99.

Gegenüber den Typkonstruktoren von SQL-99 sind speziellere Konstruktoren zum Aufbau komplexer Datentypen vorgesehen, nämlich Arrays und Nested Tables. Oracle unterstützt z.Zt. keine Typhierarchien mit Vererbung.

„Grobsyntax“:

```
typedef ::=
  CREATE [OR REPLACE] TYPE typename
     { AS OBJECT (attrname datatype {, attrname datatype …})
           [ MEMBER FUNCTION methodname [(paramtype {, paramtype …}] RETURN datatype
             { MEMBER FUNCTION methodname [(paramtype {, paramtype …}] RETURN datatype, …} ] |
       AS TABLE OF datatype  |
       AS VARRAY (maxelements) OF datatype }

tabledef ::=
  CREATE TABLE [ OF typename ] ( {columndef | tableconstraint}
                                          {, {columndef | tableconstraint} …} )
                        [ NESTED TABLE columnname STORE AS tablename
                         {, NESTED TABLE columnname STORE AS tablename …} ]
columndef ::=
  columnname [datatype] [DEFAULT | {defaultconst | NULL}] [colconstraint {, colconstraint …}]
```

Beispiele:

1)
```
CREATE TYPE Kenntnissetyp AS VARRAY(5) OF VARCHAR(30);
CREATE TABLE Angestellte
   (AngNr NUMBER,
   Name VARCHAR(50),
   Kenntnisse Kenntnissetyp);
```

Konstanten eines komplexen Typs werden durch Angabe des Typnamens als Konstruktor spezifiziert, wie z.B. in:

```
INSERT INTO Angestellte VALUES (0815, 'Heinz Becker',
   Kenntnissetyp ('Oracle', 'Java', 'Urpils') );
```

2)
```
CREATE TYPE Pruefungstyp AS OBJECT
   (Fach VARCHAR(30), Datum DATE, Note NUMBER);
CREATE TYPE Pruefungstabellentyp AS TABLE OF Pruefungstyp;
CREATE TABLE Studenten
   (MatrNr NUMBER, Name VARCHAR(30),
   AbgelegtePruefungen Pruefungstabellentyp)
   NESTED TABLE AbgelegtePruefungen STORE AS Pruefungstabelle;
INSERT INTO Studenten VALUES (4711, 'Jacques Bistro',
    Pruefungstabellentyp (Pruefungstyp ('Praktische Informatik', 11.11.1998, 1.3),
                          Pruefungstyp ('Nebenfach', 24.12.1998, 1.7) ) );
```

Mit NESTED TABLES können Relationen nur einfach geschachtelt werden; für mehrfache Schachtelung muß man Objekttypdefinition verwenden, z.B. wie folgt:

```
CREATE TYPE Frageantworttyp AS OBJECT
   (LfdNr NUMBER, Frage VARCHAR(200), Antwort VARCHAR(2000));
CREATE TYPE Protokolltyp AS TABLE OF Frageantworttyp;
CREATE TYPE Pruefungstyp AS OBJECT
   (Fach VARCHAR(30), Datum DATE, Note NUMBER, Protokoll Protokolltyp);
CREATE TYPE Pruefungstabellentyp AS TABLE OF Pruefungstyp;
CREATE TABLE Studenten
   (MatrNr NUMBER, Name VARCHAR(30),
   AbgelegtePruefungen Pruefungstabellentyp)
   NESTED TABLE AbgelegtePruefungen STORE AS Pruefungstabelle;
INSERT INTO Studenten VALUES (4711, 'Jacques Bistro',
    Pruefungstabellentyp
       (Pruefungstyp ('Praktische Informatik', 11.11.1998, 1.3,
                    Protokolltyp (
                       Frageantworttyp (1, 'Wie baut man ... ?', 'Das macht man ... '),
                       Frageantworttyp (2, 'Wieso ist ...? ', 'Der Grund ist ... '))),
        Pruefungstyp ('Nebenfach', 24.12.1998, 1.7,
                    Protokolltyp (
                       Frageantworttyp (1, 'Was ist ... ? ', 'Das ist ... '),
                       Frageantworttyp (2, 'Wie funktioniert ...? ', 'Das ... '))) ));
```

In SQL-Anfragen nimmt man auf Subobjekte einer Tabelle im Prinzip mit Pfadausdrücken ähnlich wie bei OQL Bezug. Allerdings kann man bei Subrelationen (also mengenwertigen Subobjekten) nicht einfach die komfortable Punktschreibweise verwenden, sondern muß solche Subrelationen mit der Funktion TABLE zunächst explizit auf das „Niveau“ von Top-Level-Relationen „anheben“ (eine Art Type-Casting) und kann diese dann wie gewöhnliche Relationen verarbeiten. Oracle bezeichnet dies als „Collection Unnesting“; ohne Anwendung der Funktion TABLE wird eine Subrelation wie ein skalares Attribut behandelt. Im Umgang mit geschachtelten Relationen kann man ausnutzen, daß Oracle ab Version 8 sowohl in der SELECT-Klausel als auch in der FROM-Klausel Subqueries erlaubt.

„Grobsyntax“:

```
subquery ::=
SELECT [ALL | DISTINCT]  {col | expr  | subquery} {, {col | expr | subquery} …}
FROM {table | (subquery) | TABLE (collectionexpr) } [corrvar]
        {, {table | (subquery) | TABLE (collectionexpr) } [corrvar]  …}
WHERE condition [GROUP BY col {, col …} [HAVING condition]]
{{UNION [ALL] | INTERSECT | MINUS} subquery …}
ORDER BY col {, col …} [ASC | DESC]
```

Beispiele:

```
CREATE TYPE Namenstyp AS OBJECT
   (Vorname VARCHAR(30), Nachname VARCHAR(30), Spitzname VARCHAR(30));
CREATE TYPE Pruefungstyp AS OBJECT
   (Fach VARCHAR(30), Datum DATE, Note NUMBER);
CREATE TYPE Pruefungstabellentyp AS TABLE OF Pruefungstyp;
CREATE TABLE Studenten
   (MatrNr NUMBER, Name Namenstyp,
   AbgelegtePruefungen Pruefungstabellentyp)
   NESTED TABLE AbgelegtePruefungen STORE AS Pruefungstabelle;
```

1) Gib die vollständigen Namen von Studenten mit Spitznamen „Schlumpf“ aus.

```
SELECT S.Name.Vorname, S.Name.Nachname FROM Studenten S
WHERE S.Name.Spitzname = 'Schlumpf'
```

2) Gib alle Prüfungen des Studenten mit Matrikelnummer 123456 aus.

```
SELECT S.AbgelegtePruefungen FROM Studenten S WHERE S.MatrNr = 123456
```

Die Ausgabe besteht aber aus einem einzigen skalaren Wert und nicht aus einer Tabelle von Prüfungstupeln. Man muß ansonsten die Anfrage wie folgt umformulieren:

```
SELECT *
FROM TABLE ( SELECT S.AbgelegtePruefungen FROM Studenten S
                    WHERE S.MatrNr = 123456 )
```

3) Gib die Note des Studenten mit Matrikelnummer 123456 im Fach Informationssysteme aus.

```
SELECT P.Note
FROM TABLE ( SELECT S.AbgelegtePruefungen FROM Studenten S
                    WHERE S.MatrNr = 123456 ) P
WHERE P.Fach = 'Informationssysteme';
```

Um ein Anfrageergebnis zu konstruieren, bei dem sowohl Attribute von Top-Level-Relationen als auch von Subrelationen ausgegeben werden, muß man gedanklich ein Kreuzprodukt der äußeren Relation mit der mittels TABLE auf Top-Level-Niveau abgehobenen Subrelation bilden. Durch die Verwendung einer Tupelvariablen für die äußere Relation beim Ansprechen der Subrelation wird implizit und automatisch eine „Joinbedingung“ erzeugt.

Beispiele:

4) Gib die Noten aller Studenten in allen Fächern aus.

```
SELECT S.MatrNr, P.Fach, P.Note
FROM Studenten S, TABLE (S.AbgelegtePruefungen) P
```

5) Gib die Noten aller Studenten im Fach Informationssysteme aus.

```
SELECT S.MatrNr, P.Note
FROM Studenten S,  TABLE (S.AbgelegtePruefungen) P
WHERE P.Fach = 'Informationssysteme';
```

6) Gib für jeden Studenten das letzte Prüfungsdatum aus
bzw. einen Nullwert für Studenten ohne abgelegte Prüfungen

```
SELECT S.MatrNr, Max(P.Datum)
FROM Studenten S, TABLE (S.AbgelegtePruefungen) (+) P
```

oder:

```
SELECT S.MatrNr, (SELECT Max(P.Datum) FROM TABLE(S.AbgelegtePruefungen))
FROM Studenten S
```

... in SQL-99

Die Definition von Funktionen ist ähnlich der von Stored Procedures, wobei der SQL-99-Standard 4GL-artige Erweiterungen um Kontrollflußkonstrukte u.ä. vorsieht.

Bei Abstrakten Datentypen werden private und öffentliche Attribute und Funktionen unterschieden, also eine echte Kapselung unterstützt.

Beispiele:

```
CREATE TYPE Rechteck (lu Punkt, ro Punkt);
CREATE FUNCTION  SchneidetBox (p Polylinie, r Rechteck) RETURNS BOOLEAN
BEGIN ... END;
CREATE FUNCTION Gesamtlänge (p Polylinie) RETURNS FLOAT
BEGIN ... END;
```

```
SELECT S.StrBez, Gesamtlänge(S.HatVerlauf)
FROM Straßen S
WHERE S.Ausgangsort='Saarbrücken'
AND SchneidetBox (S.HatVerlauf, <lu: <X:5.71, Y:6.24>, ro: <X:17.5, Y:22>>)
```

```
CREATE TYPE Straße
  (StrBez VARCHAR(10),
  Ausgangsort VARCHAR(30), Zielort VARCHAR(30),
  PRIVATE
    HatVerlauf Polylinie,
  PUBLIC
    FUNCTION Gesamtlänge (s Straße) RETURNS FLOAT
      ...
    END FUNCTION );
CREATE TABLE Straßen OF TYPE Straße;
```

... in Oracle

In Oracle sind ab Version 8 Abstrakte Datentypen unterstützt, wobei es allerdings offensichtlich keine privaten Attribute gibt. Die Methoden können in PL/SQL oder in Java implementiert werden.

Beispiel:

```
CREATE TYPE Kontotyp AS OBJECT
   (Kontonr NUMBER, Inhaber VARCHAR(30), Kontostand MONEY,
   MEMBER FUNCTION Tageszinsen (Zinssatz IN NUMBER) RETURN MONEY,
   MEMBER PROCEDURE Einzahlung (Betrag IN MONEY),
   MEMBER PROCEDURE Auszahlung (Betrag IN MONEY) RETURN Fehlertyp );
CREATE OR REPLACE TYPE BODY Kontotyp AS
   MEMBER FUNCTION Tageszinsen (Zinssatz IN NUMBER) RETURN MONEY
   IS BEGIN ... RETURN Kontostand*Zinssatz/100; END;
   MEMBER PROCEDURE Einzahlung (...) IS BEGIN ... END; ...
   END;
```

```
CREATE TABLE Konto OF Kontotyp;
```

Retrieval-Methoden auf Objekten vom Typ T können genauso wie Attribute von T verwendet werden.

Beispiele:

SELECT KontoNr, Tageszinsen (0.05) FROM Konto WHERE Kontostand > 100000;

SELECT KontoNr FROM Konto WHERE Tageszinsen (0.05) > 1000;

Update-Methoden werden wie Ausdrücke in SQL-Änderungsanweisungen verwendet. Sie beziehen sich jedoch nicht auf Attribute, sondern auf ein gesamtes Objekt, das durch eine Tupelvariable bezeichnet wird.

Beispiel:

UPDATE Konto K SET K = K.Einzahlung (100) WHERE KontoNr = 1234;

Referenzen auf Objekte

werden in verschiedenen Nuancen unterstützt (z.B. durch Unterscheidung VALUE TYPE vs. OBJECT TYPE) und z.T. in Produkten angeboten (u.a. auch in Oracle Version 8 mit dem Typkonstruktor REF *type*, der für Zeiger auf ein Objekt vom Typ *type* steht) .

Ergänzende Literatur zu Kapitel 10

R.G.G. Cattell, Object Data Management, Addison-Wesley, 2nd Edition, 1994

R.G.G. Cattell (Editor), The Object Database Standard: ODMG-93 Release 1.2, Morgan Kaufmann, 1996

G. Lausen, G. Vossen, Objekt-orientierte Datenbanken: Modelle und Sprachen, Oldenbourg-Verlag, 1996

C.J. Date, H. Darwen, Foundation for Object/Relational Databases: The Third Manifesto, Addison-Wesley, 1998

A. Meier, T. Wüst, Objektorientierte un dobjekt-relationale Datenbanken, dpunkt-Verlag, 2000

A.K. Huge, Formalisierung objektorientierter Datenbanken auf der Grundlage von ODMG, Shaker-Verlag, 1999

P. Gulutzan, T. Pelzer, SQL-99 Complete, Really, R&D Books, 1999

Oracle9i Concepts, Part IV: The Object-Relational DBMS

Oracle9i Application Developer's Guide - Fundamentals, Part IV: The Object-Relational DBMS

Oracle9i SQL Reference

Kapitel 11: Relationale Entwurfstheorie

Ziele:

- Charakterisierung "guter" relationaler Schemata
  - jede Relation entspricht genau einer Objektmenge - eventuell unter Einbezug von N:1- oder 1:1-Relationships - oder genau einer Relationship-Menge zwischen Objekten.
  - Redundanz ist eliminiert, alle Informationen sind repräsentierbar, und es treten keinerlei "Änderungsanomalien" auf
    - Änderungen können bei Beachtung der Primärschlüssel- und Fremdschlüsselbedingung keine Inkonsistenzen hervorrufen
    - alle Informationen lassen sich unter Wahrung der Primärschlüssel- und Fremdschlüsselbedingung (ohne "Kunstgriffe") einfügen
    - Informationen können einzeln wieder gelöscht werden, ohne die Primärschlüssel- oder Fremdschlüsselbedingung zu verletzen
- Algorithmische Herleitung solcher "guter" Schemata

Negativbeispiel:

Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge)

"Typische" Ausprägung:

Datum	**KNr**	**PNr**	**Bez**	**Preis**	**Gewicht**	**Menge**
16.7.	1	1	Papier	20.00	2.000	100
21.7.	1	1	Papier	20.00	2.000	200
26.10.	2	1	Papier	20.00	2.000	100
26.10.	2	5	Disketten	20.00	0.500	50

11.1 Funktionalabhängigkeiten (Functional Dependencies)

Definition:
Sei R eine Relation mit Attributmenge sch(R). Für X = {X1, ..., Xm} $\subseteq$ sch(R) und Y = {Y1, ..., Yn} $\subseteq$ sch(R) gilt die *Funktionalabhängigkeit* $X \rightarrow Y$, gesprochen: X bestimmt Y, wenn zu jedem Zeitpunkt für je zwei Tupel r, s $\in$ val(R) gelten muß:

(r.X1 = s.X1 $\wedge$ ... $\wedge$ r.Xm = s.Xm) $\Rightarrow$ (r.Y1 = s.Y1 $\wedge$ ... $\wedge$ r.Yn = s.Yn)

(in Kurznotation: r.X = s.X $\Rightarrow$ r.Y = s.Y).

Beispiel:
Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge)
Es gelten beispielsweise die Funktionalabhängigkeiten:

Datum, KNr, PNR $\rightarrow$ Menge

PNr $\rightarrow$ Bez, Preis, Gewicht

Es gelten nicht die Funktionalabhängigkeiten:

KNr, PNR $\rightarrow$ Menge

PNr $\rightarrow$ Datum

Achtung: Funktionalabhängigkeiten sind Integritätsbedingungen, die für alle möglichen Ausprägungen gelten sollen. Insofern kann man aus einer Ausprägung nur schließen, welche Funktionalabhängigkeiten nicht gelten. Eine "typische" Ausprägung kann freilich Anhaltspunkte für Funktionalabhängigkeiten liefern.

Konkretisierung des Ziels der relationalen Entwurfstheorie:
Alle Funktionalabhängigkeiten sollen in Form von relationalen Primärschlüsselbedingungen verkörpert werden.
Beispiel:
Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge)
verletzt diese Forderung, da beispielsweise PNr $\rightarrow$ Bez keine Primärschlüsselbedingung ist.

Ableitungsregeln für Funktionalabhängigkeiten

Definition:
Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von spezifizierten Funktionalabhängigkeiten. Die Menge aller aus F logisch ableitbaren Funktionalabhängigkeiten wird als *transitive Hülle* F^+ von F bezeichnet.

Regeln:

Sei R eine Relation mit Attributmenge sch(R). Seien $X \subseteq sch(R)$, $Y \subseteq sch(R)$, $Z \subseteq sch(R)$ Mengen von Attributen von R.

Regel 1 (Reflexivität): Wenn $Y \subseteq X$, dann gilt: $X \rightarrow Y$.
Regel 2 (Erweiterung): Wenn $X \rightarrow Y$, dann gilt: $XZ \rightarrow YZ$
(ausführliche Notation: $(X \cup Z) \rightarrow (Y \cup Z)$).
Regel 3 (Transitivität): Wenn $X \rightarrow Y$ und $Y \rightarrow Z$, dann gilt: $X \rightarrow Z$.

Diese drei Regeln werden auch als Armstrong-Regeln oder Armstrong-Axiome (eines Inferenzsystems für Funktionalabhängigkeiten) bezeichnet. Aus ihnen folgen weitere Ableitungsregeln, beispielsweise:

Regel 4 (Vereinigung): Wenn $X \rightarrow Y$ und $X \rightarrow Z$, dann gilt: $X \rightarrow YZ$.
Regel 5 (Pseudotransitivität): Wenn $X \rightarrow Y$ und $WY \rightarrow Z$ mit $W \subseteq sch(R)$, dann gilt: $XW \rightarrow Z$.
Regel 6 (Zerlegung): Wenn $X \rightarrow Y$ und $Z \subseteq Y$, dann gilt: $X \rightarrow Z$.

Beispiel:
Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge)
Sei F = { Datum, KNr, PNR $\rightarrow$ Menge,
PNr $\rightarrow$ Bez, Preis, Gewicht }.

Es folgen:	Datum, KNr, PNR $\rightarrow$ PNr	nach der Reflexivitätsregel,
	Bez, Preis, Gewicht $\rightarrow$ Bez	nach der Reflexivitätsregel,
	PNr $\rightarrow$ Bez	nach der Transitivitätsregel und schließlich
	Datum, KNr, PNr $\rightarrow$ Bez	nach der Transitivitätsregel.

Satz:
Sei R eine Relation mit einer Menge F von Funktionalabhängigkeiten. Durch endlichmalige Anwendung der drei Armstrong-Regeln lassen sich genau alle Funktionalabhängigkeiten in F^+ herleiten.
Die Armstrong-Regeln bilden also einen korrekten und vollständigen Ableitungskalkül für die Funktionalabhängigkeiten in F^+.

Beweis:
siehe Vorlesung

Ableitbarkeitstest für Funktionalabhängigkeiten

Definition:

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten, und sei $X \subseteq sch(R)$. Die *Hülle* X^+ der Attributmenge X ist die Menge aller Attribute $A \in sch(R)$, für die gilt $X \rightarrow A \in F^+$.

Algorithmus zur Bestimmung von X^+:

Eingabe: Attributmenge sch(R)
Funktionalabhängigkeiten F
Attributmenge $X \subseteq sch(R)$

Ausgabe: X^+

```
procedure xplus (X: set of attribute): set of attribute;
var   H: set of attribute;
      newattr: Boolean;
begin
   H := X; newattr := true;
   while newattr do (* Invariante: X → H *)
      newattr := false;
      for each Y → Z ∈ F do
         if Y ⊆ H then
            if not (Z ⊆ H) then H := H ∪ Z; newattr := true; fi
         fi
      od
   od
   return H
end xplus;
```

Korrektheitsskizze:

Es gilt die Schleifeninvariante, was durch Induktion über die Anzahl der Schleifendurchläufe gezeigt werden kann.

Verankerung:

Vor Eintritt in die Schleife ist H = X, und es gilt $X \rightarrow X$ nach Armstrong-Regel 1.

Induktionsschluß:

Es gelte $X \rightarrow H(i-1)$ vor dem i-ten Schleifendurchlauf; dabei sei H(i-1) der Wert von H nach dem (i-1)-ten Durchlauf. Nach dem i-ten Schleifendurchlauf gilt dann:
$X \rightarrow H(i-1)$ und $H(i-1) \rightarrow Y \rightarrow Z$ sowie nach Regel 3 auch $X \rightarrow Z$ und nach Regel 4 auch $X \rightarrow H(i-1)\ Z$.
Mit der Zuweisung H(i) := H(i-1) Z am Ende des Schleifenrumpfes folgt also $X \rightarrow H(i)$.

Beispiel:

sch(R) = {A, B, C, D, E, G}
F = {AB → C, ACD → B, BC → D, BE → C, C → A, CG → BD, CE → AG, D → EG}
X = {B, D}

nach Schleifendurchlauf	H
0	{B, D}
1	{B, D, E, G}
2	{B, D, E, G, C, A}
3	{B, D, E, G, C, A}

Definition:

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten, und sei $X \subseteq sch(R)$. Eine Attributmenge $X \subseteq sch(R)$ ist ein *Schlüsselkandidat*, wenn $X \rightarrow sch(R) \in F^+$ gilt und es keine echte Teilmenge von X gibt, die diese Eigenschaft hat.
Ein Attribut $A \in sch(R)$ heißt Schlüsselattribut, wenn es in mindestens einem Schlüsselkandidaten von R vorkommt.

Bemerkung:
Diese Definition eines Schlüsselkandidaten ist konsistent zur Definition von Kapitel 2.

Algorithmus zur Bestimmung aller Schlüsselkandidaten einer Relation:

Eingabe: Attributmenge sch(R)
Funktionalabhängigkeiten F
Ausgabe: alle Schlüsselkandidaten von R

```
procedure findkeys: set of set of attribute;
var   K: set of set of attribute;
      iskey: Boolean;
begin
   K := ∅;
   for each S ∈ 2^sch(R) do
      iskey := true;
      if xplus (S) ≠ sch(R) then iskey := false fi;
      for each A ∈ S while iskey do
         if xplus (S - {A}) = sch(R) then iskey := false fi
      od
      if iskey then K := K ∪ {S} fi
   od
   return K
end findkeys;
```

11.2 Relationale Normalformen

Definition Boyce-Codd-Normalform (BCNF):

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. R ist in *Boyce-Codd-Normalform (BCNF)*, wenn für jede Funktionalabhängigkeit $X \rightarrow A \in F^+$ mit $X \subseteq sch(R)$, $A \in sch(R)$ und $A \notin X$ gilt, daß X einen Schlüsselkandidaten enthalten muß.

Definition 3. Normalform (3NF):

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. R ist in *3. Normalform (3NF)*, wenn für jede Funktionalabhängigkeit $X \rightarrow A \in F^+$ mit $X \subseteq sch(R)$, $A \in sch(R)$ und $A \notin X$ gilt, daß X einen Schlüsselkandidaten enthalten muß oder A ein Schlüsselattribut ist.

Satz:

Jede Relation in BCNF ist auch in 3NF.

Algorithmus zum Testen, ob eine Relation in BCNF ist

Eingabe: Attributmenge sch(R)
Funktionalabhängigkeiten F
Ausgabe: true g.d.w. die Relation in BCNF ist

```
procedure BCNFtest: Boolean;
var   isbcnf: Boolean;
begin
   isbcnf := true;
   for each X → A ∈ F while isbcnf do
      if A ∉ X then if XPlus(X) ≠ sch(R) then isbcnf := false fi fi
   od
   return isbcnf;
end BCNFtest;
```

Beispiele:

1) Best (Datum, KNr, PNr, Menge) mit F = {Datum, KNr, PNr $\rightarrow$ Menge}
und
Prod (PNr, Bez, Preis, Gewicht) mit F = {PNr $\rightarrow$ Bez, Preis, Gewicht}
sind in BCNF.

2) Vorlesungen (Titel, Zeit, Raum)
mit F = { Titel $\rightarrow$ Raum,
Zeit, Raum $\rightarrow$ Titel}
ist in 3NF, nicht aber in BCNF.
Schlüsselkandidaten sind {Titel, Zeit} sowie {Zeit, Raum}.
Die Funktionalabhängigkeit Titel $\rightarrow$ Raum verletzt die BCNF-Bedingung.
Die Anomalie, daß eine Vorlesung zu verschiedenen Zeiten in verschiedenen Räumen stattfindet, wäre also nur durch die Primärschlüsselbedingung nicht auszuschließen.

Beispielausprägung zu Beispiel 2:

Titel	**Zeit**	**Raum**
Datenbanken	Di 9-11	HS 2
Datenbanken	Do 9-11	HS 2
Compiler	Di 9-11	HS 3
Compiler	Mi 13-15	HS 3
Betriebssysteme	Mi 13-15	HS 2

11.3 Relationendekomposition

Definition Abhängigkeitsbewahrung:

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. Eine Zerlegung von R in R1 (X1), ..., Rk (Xk) mit Xi $\subseteq$ sch(R) und
X1 $\cup$ .. $\cup$ Xk = sch(R) heißt *abhängigkeitsbewahrend*, wenn gilt:

$$\left((F^+ \mid_{X1})^+ \cup ... \cup (F^+ \mid_{Xk})^+ \right)^+ = F^+,$$

wobei $F^+ \mid_{Xi}$ diejenigen Funktionalabhängigkeiten aus F^+ bezeichnen, deren linke und rechte Seite in Xi enthalten sind.

Beispiele:

1) Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge}
   mit F = { Datum, KNr, PNR $\rightarrow$ Menge,
   PNr $\rightarrow$ Bez, Preis, Gewicht}
   Zerlegung in
   Best (Datum, KNr, PNr, Menge) mit F1 = {Datum, KNr, PNR $\rightarrow$ Menge} und
   Prod (PNr, Bez, Preis, Gewicht) mit F2 = {PNr $\rightarrow$ Bez, Preis, Gewicht}
   ist abhängigkeitsbewahrend.

2) Vorlesungen (Titel, Zeit, Raum)
   mit F = { Titel $\rightarrow$ Raum,
   Zeit, Raum $\rightarrow$ Titel}
   Zerlegung in
   Vorlesungsräume (Titel, Raum) mit F1 = {Titel $\rightarrow$ Raum} und
   Vorlesungszeiten (Titel, Zeit) mit F2 = $\varnothing$
   ist nicht abhängigkeitsbewahrend.

Definition Verlustfreiheit:

Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. Eine Zerlegung von R in R1 (X1), ..., Rk (Xk) mit Xi $\subseteq$ sch(R) und X1 $\cup$ ... $\cup$ Xk = sch(R) heißt *(projektion-join-) verlustfrei*, wenn für alle möglichen Ausprägungen, die F erfüllen, gilt:

π[X1](R) |×| ... |×| π[Xk](R) = R.

Beispiel:

Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge)
mit F = { Datum, PNr, KNr $\rightarrow$ Menge,
PNr $\rightarrow$ Bez, Preis, Gewicht} und
der Beispielausprägung:

Datum	**KNr**	**PNr**	**Bez**	**Preis**	**Gewicht**	**Menge**
16.7.	1	1	Papier	20.00	2.000	100
16.7.	1	5	Disketten	20.00	0.500	50

Zerlegung in
Best1 Best2

Datum	**KNr**	**PNr**	**Menge**
16.7.	1	1	100
16.7.	1	5	50

und

Datum	**Bez**	**Preis**	**Gewicht**
16.7.	Papier	20.00	2.000
16.7.	Disketten	20.00	0.500

ist nicht verlustfrei, denn Best1 |×| Best2 hat die folgende Ausprägung:

Datum	**KNr**	**PNr**	**Bez**	**Preis**	**Gewicht**	**Menge**
16.7.	1	1	Papier	20.00	2.000	100
16.7.	1	5	Disketten	20.00	0.500	50
16.7.	1	1	Disketten	20.00	0.500	100
16.7.	1	5	Papier	20.00	2.000	50

Satz:
Zu jeder Relation R mit Funktionalabhängigkeiten F gibt es eine Zerlegung von R in 3NF-Relationen, die abhängigkeitsbewahrend und verlustfrei ist.

Satz:
Zu jeder Relation R mit Funktionalabhängigkeiten F gibt es eine Zerlegung von R in BCNF-Relationen, die verlustfrei ist.

Satz:
Sei R eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. Sei $X \rightarrow Y \in F^+$ mit $X \cap Y = \emptyset$.
Die Zerlegung von R in R' (X, Y) und R" (X, sch(R) - Y) ist verlustfrei.

Algorithmus für verlustfreie BCNF-Dekomposition

Eingabe: Relationenschema sch(R)
Funktionalabhängigkeiten F
Ausgabe: verlustfreie Zerlegung in R1, ..., Rk, so daß R1, ..., Rk alle in BCNF sind

Algorithmus:
Teste, ob R in BCNF ist
Falls R nicht in BCNF ist
(* es gibt eine Funktionalabhängigkeit $X \rightarrow Y \in F^+$ mit $X \cap Y = \emptyset$ und $X \rightarrow sch(R) \notin F^+$ *)
dann
zerlege R in R' (X, Y) und R" (X, sch(R) - Y)
wiederhole den Zerlegungsalgorithmus für R' und R"

Beispiel:

R (FlugNr, Datum, Abflugzeit, FSNr, SitzNr, TicketNr, Name, Adresse, GNr, Gewicht)
F = { FlugNr, Datum → Abflugzeit, FSNr,
FlugNr, Datum, TicketNr → SitzNr,
TicketNr → Name, Adresse,
GNr → Gewicht }

Schlüsselkandidat: FlugNr, Datum, TicketNr, GNr

Zerlegungsschritte:

1. Zerlegung von R entlang der Funktionalabhängigkeit: FlugNr, Datum → Abflugzeit, FSNr:
   R1 (FlugNr, Datum, Abflugzeit, FSNr) in BCNF und
   R2 (FlugNr, Datum, SitzNr, TicketNr, Name, Adresse, GNr, Gewicht)

2. Zerlegung von R2 entlang der Funktionalabhängigkeit: FlugNr, Datum, TicketNr → SitzNr:
   R21 (FlugNr, Datum, TicketNr, SitzNr) in BCNF und
   R22 (FlugNr, Datum, TicketNr, Name, Adresse, GNr, Gewicht)

3. Zerlegung von R22 entlang der Funktionalabhängigkeit: TicketNr → Name, Adresse:
   R221 (TicketNr, Name, Adresse) in BCNF und
   R222 (TicketNr, FlugNr, Datum, GNr, Gewicht)

4. Zerlegung von R222 entlang der Funktionalabhängigkeit: GNr → Gewicht:
   R2221 (GNr, Gewicht) in BCNF und
   R2222 (GNr, FlugNr, Datum, TicketNr) in BCNF

Umbenennungen:

R1	in Flüge
R21	in Sitzreservationen
R221	in PassagiereMitTicket
R2221	in Gepäckstücke
R2222	in Check-in

Achtung:
Das Resultat der Relationendekomposition ist abhängig von der Reihenfolge der Zerlegungsschritte und der Wahl der Funktionalabhängigkeiten für die Zerlegungen.
Ein intuitiv guter Entwurf wird u.U. nur durch geschickte Wahl der Zerlegungsschritte erreicht.

11.4 Relationensynthese

Definition Überdeckung:
Sei eine Relation mit Attributmenge sch(R) und einer Menge F von Funktionalabhängigkeiten. Eine Menge F' von Funktionalabhängigkeiten über sch(R) heißt Überdeckung (engl. cover) von F, wenn gilt: $(F')^+ = F^+$.

Definition minimale Überdeckung:
Eine Überdeckung F' von F heißt minimal, wenn gilt:

1) In jeder Funktionalabhängigkeit von F' hat die rechte Seite ein Attribut.
2) Keine echte Teilmenge von F' ist eine Überdeckung von F.
3) Für jede Funktionalabhängigkeit $X \rightarrow A \in F'$ und jede echte Teilmenge X' von X ist $F'' := F' - \{ X \rightarrow A\} \cup \{X' \rightarrow A\}$ keine Überdeckung mehr von F.

Beispiel:
sch(R) = {Ang(estellten)Nr, Name, Gehalt, Leistung(sbeurteilung), Tarif(klasse),
Abt(eilungs)Nr, ProjektNr, Projektleiter, Abt(eilungs)leiter}
F = { AngNr → Name,
AngNr → Gehalt,
AngNr → Tarif,
AngNr → AbtNr,
AngNr → Abtleiter,
ProjektNr → Projektleiter,
AbtNr → Abtleiter,
AngNr, Abtleiter → Leistung,
Tarif, Leistung → Gehalt}

Schritt 1: Entferne redundante Attribute auf den linken Seiten
→ { AngNr → Name,
AngNr → Gehalt,
AngNr → Tarif,
AngNr → AbtNr,
AngNr → Abtleiter,
ProjektNr → Projektleiter,
AbtNr → Abtleiter,
AngNr → Leistung,
Tarif, Leistung → Gehalt}

Schritt 2: Entferne redundante Funktionalabhängigkeiten
→ { AngNr → Name,
AngNr → Tarif,
AngNr → AbtNr,
ProjektNr → Projektleiter,
AbtNr → Abtleiter,
AngNr → Leistung,
Tarif, Leistung → Gehalt} = F'

Algorithmus zur Berechnung einer minimalen Überdeckung:

Eingabe: Attributmenge sch(R)
Funktionalabhängigkeiten F
(Annahme: alle rechten Seiten von F bestehen aus einem Attribut)
Ausgabe: minimale Überdeckung von F

```
procedure mincover: set of FDs;

  var G: set of FDs;
  procedure xplus (arg1: set of FDs, arg2: set of attribute): set of attribute;
  procedure reduce (arg: set of FDs): set of FDs;
    var H: set of FDs;
    begin
    H := arg;
    for each X→A∈H do
      for each C∈X do
        Hred := H - { X→A} ∪ {(X-{C})→A};
        if A ∈ xplus (H, X-{C})) then H := Hred fi;
      od;
    od;
    return H
    end (* reduce *);

begin
G := reduce(F);
for each X→A∈G do
  G := G - { X→A};
  if not (A ∈ xplus (G, X)) then G := G ∪ { X→A} fi;
od;
return G
end mincover;
```

Achtung:
Es kann mehrere, verschiedene, minimale Überdeckungen von F geben. Das Ergebnis des Algorithmus ist also abhängig von der Reihenfolge, in der die Funktionalabhängigkeiten inspiziert werden.

Algorithmus zur Relationensynthese:

Eingabe: Attributmenge sch(R)
Funktionalabhängigkeiten F
Ausgabe: Menge von Relationen R1, ..., Rn mit Schemata sch(R1), ..., sch(Rn) in 3NF
die eine abhängigkeitsbewahrende Zerlegung von R bilden.

Schritt 1: Bereche eine minimale Überdeckung F' von F.
Schritt 2: Partitioniere die Funktionalabhängigkeiten in F' nach gleichen linken Seiten, d.h. für jede Attributmenge Xi, die als linke Seite einer oder mehrerer Funktionalabhängigkeiten in F' vorkommt, alle Funktionalabhängigkeiten $Xi \rightarrow A1$, ..., $Xi \rightarrow Ak_i$ zusammen.
Schritt 3: Bilde für jede Partition von Schritt 2 eine Relation Ri mit $sch(Ri) = Xi \cup \{Ai, ..., Ak_i\}$.
Schritt 4: Falls $sch(R) - (\cup_i sch(Ri))$ nichtleer ist (es also Attribute in sch(R) gibt, die in keiner Funktionalabhängigkeit von F' vorkommen), bilde eine weitere Relation R' mit $sch(R') = sch(R) - (\cup_i sch(Ri))$.
Schritt 5: Falls keine der erzeugten Relationen Ri, R' einen Schlüssel von R enthält, erzeuge eine weitere Relation R", deren Schema aus einem Schlüssel von R besteht.

Satz:
Der Algorithmus zur Relationensynthese erzeugt eine abhängigkeitsbewahrende und verlustfreie 3NF-Zerlegung von R.

Beispiel:
sch(R) = {Ang(estellten)Nr, Name, Gehalt, Leistung(sbeurteilung), Tarif(klasse),
Abt(eilungs)Nr, ProjektNr, Projektleiter, Abt(eilungs)leiter}
mit minimaler Überdeckung
F' = { AngNr $\rightarrow$ Name,
AngNr $\rightarrow$ Tarif,
AngNr $\rightarrow$ AbtNr,
ProjektNr $\rightarrow$ Projektleiter,
AbtNr $\rightarrow$ Abtleiter,
AngNr $\rightarrow$ Leistung,
Tarif, Leistung $\rightarrow$ Gehalt}

Ergebnis der Relationensynthese:
Angestellte (<u>AngNr</u>, Name, Tarif, Leistung, AbtNr)
Projekte (<u>ProjektNr</u>, Projektleiter)
Abteilungen (<u>AbtNr</u>, Abtleiter)
Lohn (<u>Tarif, Leistung</u>, Gehalt)
Projektmitarbeit (<u>AngNr, ProjektNr</u>)

11.5 Ergänzungen zum algorithmischen Entwurf

Das (algorithmische) Resultat der Relationendekomposition oder der Relationensynthese sollte stets einer kritischen Prüfung unterzogen werden.

Beispiele für sinnvolle "Nachbesserungen von Hand":

- weitere Zerlegung von BCNF-Relationen:
  Angestellte (ANr, Informatikkenntnisse, Kinder) mit F = ∅
  Beispielausprägung:

ANr	Informatikkenntnisse	Kinder
1	Leda	Sabrina
1	Oracle	Sabrina
2	Leda	Pierre
2	Leda	Sabrina

  sollte weiter zerlegt werden in
  AngInfKenntnisse (ANr, Informatikkenntnisse) und
  AngKinder (ANR, Kinder)
  → Theorie der mehrwertigen Abhängigkeiten (Multivalued Dependencies)
  → 4. Normalform (4NF)

- unnötig feine Zerlegungen:
  Produkte (PNr, Bez, Preis)
  mit F = { PNr → Bez,
  PNr → Preis}
  sollte nicht zerlegt werden in
  Produktbez (PNr, Bez) und
  Produktpreise (PNr, Preis)

- Zerlegung aufgrund irrelevanter Funktionalabhängigkeiten:
  Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge, Summe)
  mit F = { Datum, KNr, PNr → Menge,
  PNr → Bez, Preis, Gewicht,
  Preis, Menge → Summe}
  sollte nicht zur folgenden Relation führen
  Preise (Preis, Menge, Summe)

- Nicht-BCNF-Relationen, die bezüglich Redundanz und potentieller Inkonsistenz unproblematisch sind:
  Kunden (KNr, Name, Postleitzahl, Stadt, Strasse)
  mit F = { KNr → Name, Postleitzahl, Stadt, Strasse,
  Postleitzahl → Stadt }
  muß nicht notwendigerweise zerlegt werden in
  Kundenadressen (KNr, Name, Postleitzahl, Strasse) und
  Postleitzahlenverzeichnis (Postleitzahl, Stadt)

- M:N-Relationships ohne Attribute:
  Sitze (FlugNr, Datum, SitzNr)
  muß bei der Relationendekomposition u.U. manuell hinzugefügt werden (falls diese Information relevant ist)

Ergänzende Literatur zu Kapitel 11:

D. Maier, The Theory of Relational Databases, Computer Science Press, 1983

H. Mannila, K.-J. Raiha, The Design of Relational Databases, Addison-Wesley, 1992

C. Fleming, B. von Halle, Handbook of Relational Database Design, Addison-Wesley, 1988

P. Atzeni, C. de Antonellis, Relational Database Theory, Benjamin Cummings, 1992

Kapitel 12: Datenmodellierung mit ERM und UML

Ziel der Datenmodellierung (des konzeptionellen Datenbankentwurfs):

Modellierung von Sachverhalten der realen Welt mittels weniger Grundkonzepte
mit dem Zweck, eine computergestützte Datenbank aufzubauen.
Dazu sind notwendig:

- eine *Modellierungssprache* (ein "semantisches" Datenmodell)
- eine *Entwurfsmethodologie*
- computergestützte *Entwurfswerkzeuge* (für große Entwürfe)

Rolle der Datenmodellierung bei der Entwicklung von Informationssystemen:

Informationssysteme werden in mehreren Stufen entwickelt:

1) Analyse der Anwendungswelt (Informationsbedarf, zu automatisierende Abläufe)
2) Datenmodellierung (sowie Funktions- und Prozeßmodellierung)
3) Abbildung des Datenmodells auf ein (relationales) Datenbankschema

Notwendigkeit einer systematischen Datenmodellierung (Entwurfstheorie):

Relationale Datenbankschemata können "gut" oder "schlecht" entworfen sein.
Es sollte Konstruktionsrichtlinien für (komplexere) Datenbankschemata geben.

Beispiel eines schlechten bzw. debattierbaren Entwurfs:

Bestellungen (Datum, KNr, PNr, Bez, Preis, Gewicht, Menge, Summe, Status)
Produkte (PNr, Bez, Lagerort, Vorrat)
Kunden (KNr, Name, Adresse, Rabatt, Saldo)

→ Probleme: Datenredundanz, potentielle Inkonsistenz,
Nichtdarstellbarkeit bestimmter Informationen

Beispiel:
Wie wäre das Schema der Bestellungs-DB aufzubauen, wenn mit einer Bestellung mehrere Produkte auf einmal bestellt werden können, wenn diese u.U. über mehrere Lieferungen verteilt geliefert werden, entsprechend mehrere Teilzahlungen eingehen, etc. ?

12.1 Datenmodellierung mit dem Entity-Relationship-Modell (ERM)

Entities und Entity-Mengen (Entitäten, Objekte, Gegenstände)

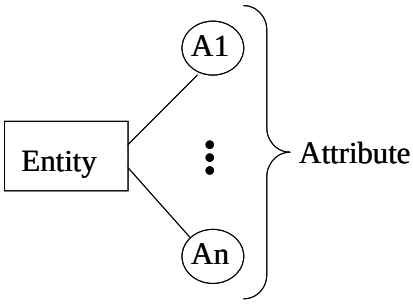
Ein Entity ist ein eindeutig identifizierbarer Gegenstand der realen Welt oder unserer Vorstellung (eine Person, ein Kunde, ein Buch, ein Bankkonto, etc.).
Gleichartige Entities werden zu einem *Entity-Typ* zusammengefaßt (z.B. alle Bücher, alle Bankkonten). Alle Entities desselben Entity-Typs werden durch eine einheitliche Menge von *Attributen* (Eigenschaften, Merkmalen) beschrieben, die jeweils einem Entity einen Wert eines bestimmten Wertebereichs (Domains) zuordnen.
Im Gegensatz zum Relationenmodell sind die Wertebereiche von Attributen nicht auf primitive Datentypen beschränkt, sondern können beispielsweise mengenwertig sein (z.B. Informatikkenntnisse als Attribut von Studenten oder Angestellten).

Beispiele:
- ISBN, Titel, Autoren, Verlag, Erscheinungsjahr als Attribute von Büchern
- Kontonummer, Kontoinhaber, Kontostand, Zinssatz als Attribute von Bankkonten
- Matrikelnummer, Name, Adresse, Semester, Prüfungen als Attribute von Studenten

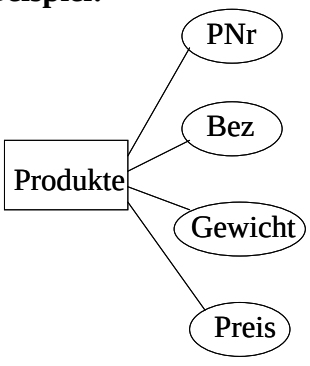
Eine Menge von Entities desselben Typs heißt Entity-Menge (Objektmenge, Objektklasse). Es kann mehrere, voneinander verschiedene Entity-Mengen desselben Entity-Typs geben (z.B. Studenten der Informatik und Studenten der Elektrotechnik, Bücher der Informatikbibliothek und Bücher der Universitätsbibliothek). Entity-Mengen müssen nicht disjunkt sein (z.B. Ärzte und Patienten, beide vom Entity-Typ Personen).

Bemerkungen:
- Bei vielen Anwendungen gibt es pro Entity-Typ genau eine relevante Entity-Menge. In diesen Fällen wird in der Regel nicht explizit zwischen Entity-Typ und Entity-Menge unterschieden.
- Der (Analyse-) Schritt von der Betrachtung einzelner Entities zur Betrachtung von Entity-Mengen wird auch als Abstraktion oder Klassifikation bezeichnet. Bei der Informationsmodellierung ist nur diese (Schema-) Ebene von Bedeutung.

Graphische Darstellung:



Beispiel:



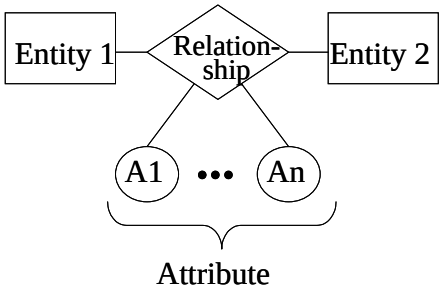
Eine Attributmenge K einer Entity-Menge E heißt *Schlüsselkandidat*, wenn zu jedem Zeitpunkt für je zwei verschiedene Entities e1, e2 $\in$ E gelten muß, daß sich e1 und e2 bezüglich K unterscheiden, und es keine echte Teilmenge von K gibt, die diese Eigenschaft hat (z.B. Kontonummer bei Bankkonten einer bestimmten Bank, Kontonummer und Name der Bank bei allen Bankkonten, Personalnummer bei Angestellten).
Der *Primärschlüssel* einer Entity-Menge ist ein explizit ausgewählter Schlüsselkandidat.

Relationships und Relationship-Mengen (Beziehungen, Assoziationen)

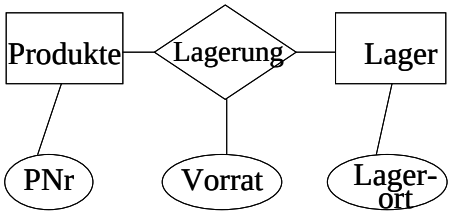
Eine *Relationship-Menge* R zwischen Entity-Mengen E1, ..., En ist eine Teilmenge des kartesischen Produkts E1 × ... × En (meist ist n=2). Ein Element <e1, ..., en> einer Relationship-Menge bildet eine Relationship zwischen den Entities e1, ..., en.
Alle möglichen Relationship-Mengen zwischen denselben Entity-Typen bilden zusammen einen Relationship-Typ.
Die Elemente einer Relationship-Menge können durch Attribute zusätzlich charakterisiert werden (z.B. die Vorratsmenge, die von einem Produkt in einem Lager vorhanden ist, oder die Note, die ein Student in einem Prüfungsfach erreicht hat). Die Primärschlüssel der an einer Relationship-Menge beteiligten Entity-Mengen gelten als implizite Attribute der Relationship-Menge.

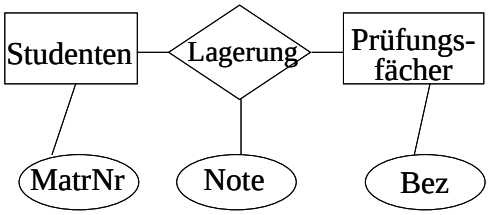
Bemerkungen:
- Bei vielen Anwendungen gibt es pro Relationship-Typ genau eine relevante Relationship-Menge. In diesen Fällen wird in der Regel nicht explizit zwischen Relationship-Typ und Relationship-Menge unterschieden.
- Die Modellierung von Relationships zwischen Entities wird auch als (spezielle Form der) Aggregation bezeichnet.

Graphische Darstellung (ERM-Diagramm):



Beispiele:



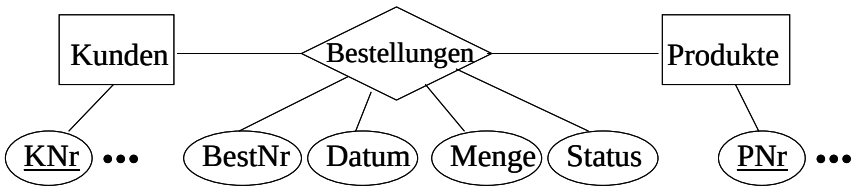


Eine Teilmenge K der Vereinigung $\bigcup_{i=1}^{n} Ki$ der Primärschlüssel K1, ..., Kn von E1, ..., En heißt *Schlüsselkandidat* der Relationship-Menge R, wenn sich zu jedem Zeitpunkt je zwei verschiedene Elemente r=<e1, ..., en> und r'=<e1', ..., en'> von R bezüglich K unterscheiden müssen und es keine echte Teilmenge von K gibt, die diese Eigenschaft hat.
Der *Primärschlüssel* einer Relationship-Menge ist ein explizit ausgewählter Schlüsselkandidat.

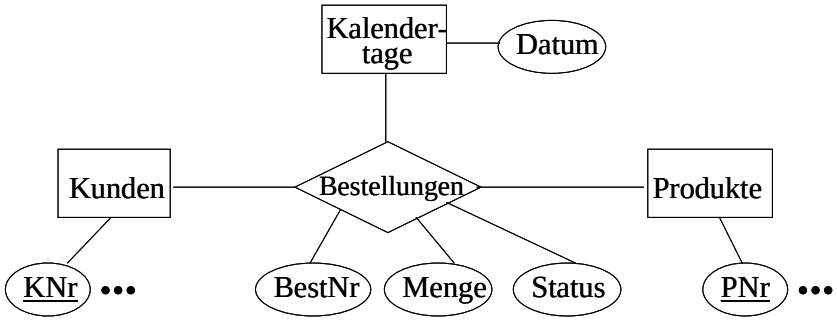
Beispiel:
PNr ist ein Schlüsselkandidat der Relationship-Menge Lagerung, falls ein Produkt immer nur an einem Ort gelagert wird. Andernfalls ist {PNr, Lagerort} ein Schlüsselkandidat von Lagerung.

Modellierung eines Sachverhalts als Entity oder als Relationship?

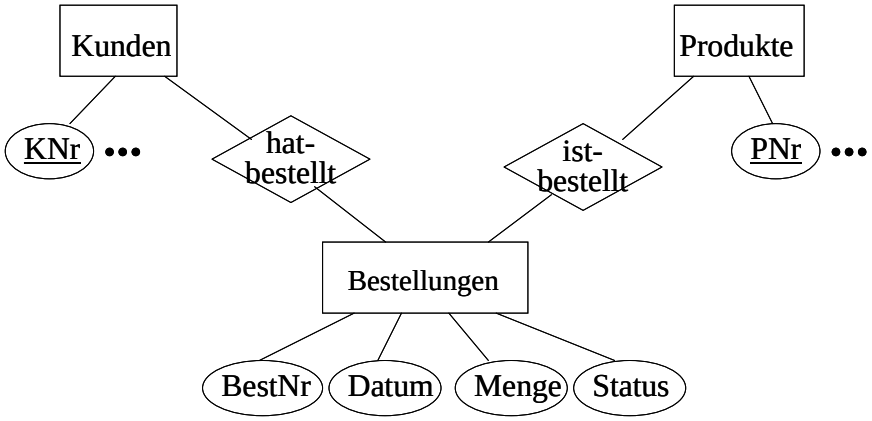
Variante 1:



Variante 2:



Variante 3:



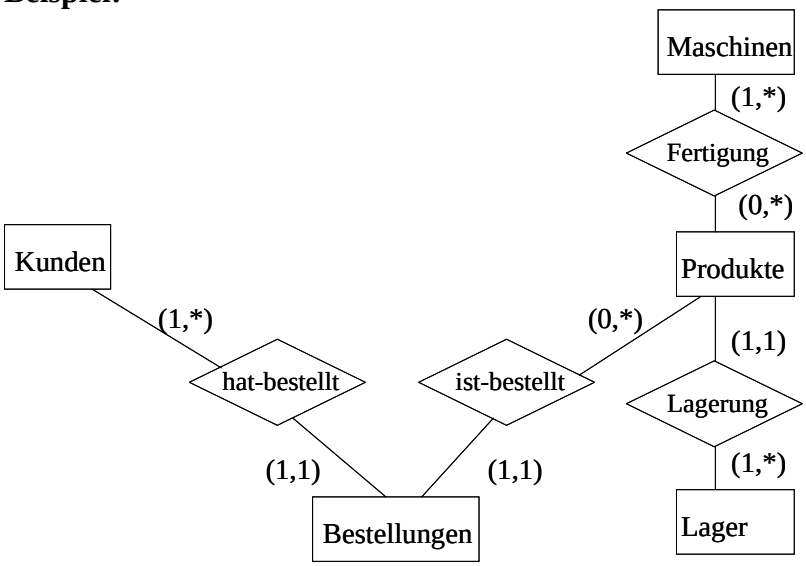
Variante 1 ist inadäquat bzw. inkorrekt, da es mehrere Bestellungen mit demselben Wert für <KNr, PNr> geben kann.
Variante 2 gibt dem Kalender eine unangemessen prominente Position.
Variante 3 ist hier wohl die angemessenste Modellierung.

Kardinalitätsbedingungen für (Entities in) Relationship-Mengen

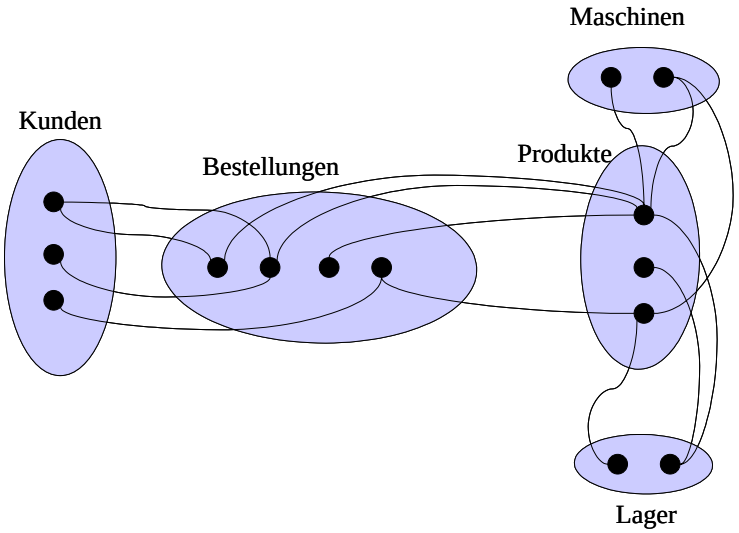
Eine Kardinalitätsbedingung für die Rolle (Bindung, Assoziation) einer Entity-Menge E in einer Relationship-Menge R spezifiziert, wie häufig ein Entity von E in R mindestens vorkommen muß und/oder wie häufig es höchstens vorkommen darf.

Notation:
Die Kante zwischen E und R in einem ERM-Diagramm wird mit (x,y) annotiert, wobei x, y natürliche Zahlen sind und für y auch das Symbol "*" (für "don't care") stehen darf.
x bezeichnet die Untergrenze für die Anzahl der Beziehungen eines Entities, y die Obergrenze. Falls für y das Symbol "*" angegeben ist, ist keine Obergrenze spezifiziert; in diesem Fall wird statt "*" gelegentlich auch "N" geschrieben.

Beispiel:

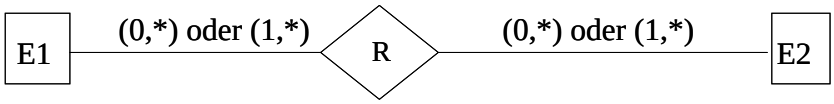


Beispielausprägung:

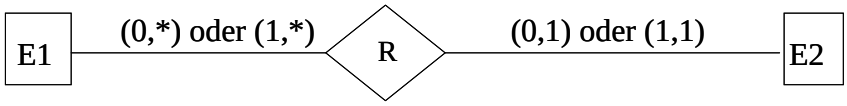


Verschiedene Arten von Relationships je nach Art der Kardinalitätsbedingung

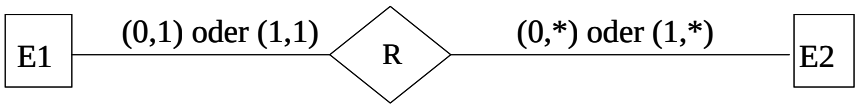
M:N-Relationship (Viele-zu-viele-Beziehung, komplexe Beziehung)



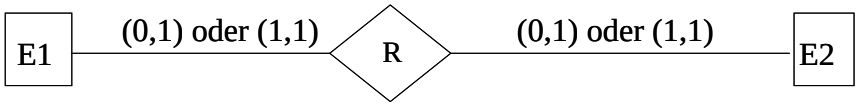
1:N-Relationship (Eins-zu-viele-Beziehung, hierarchische Beziehung)



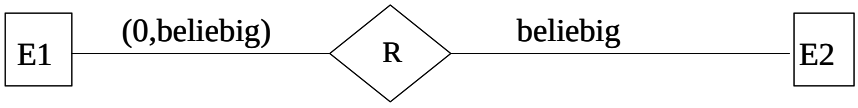
N:1-Relationship (Viele-zu-eins-Beziehung, hierarchische Beziehung)



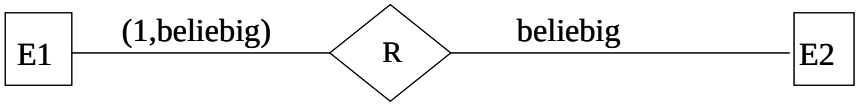
1:1-Relationship (Eins-zu-eins-Beziehung, injektive Beziehung)



Optionale Rolle eines Entities E1 in einer Relationship R



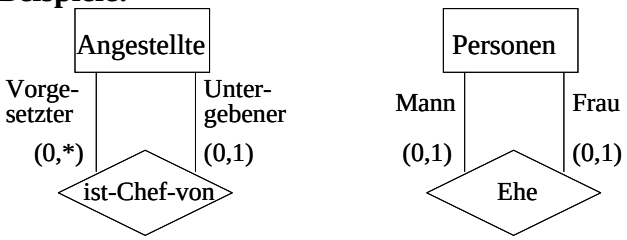
Verpflichtende Rolle eines Entities E1 in einer Relationship R



Rekursive Relationship-Mengen

Eine Relationship-Menge zwischen einer Entity-Menge E und sich selbst heißt *rekursiv*. Um Kardinalitätsbedingungen richtig ausdrücken zu können, werden bei rekursiven Relationship-Mengen die verschiedenen "Rollen" von E benannt. Im ERM-Diagramm werden die entsprechenden Kanten mit den Rollennamen annotiert.

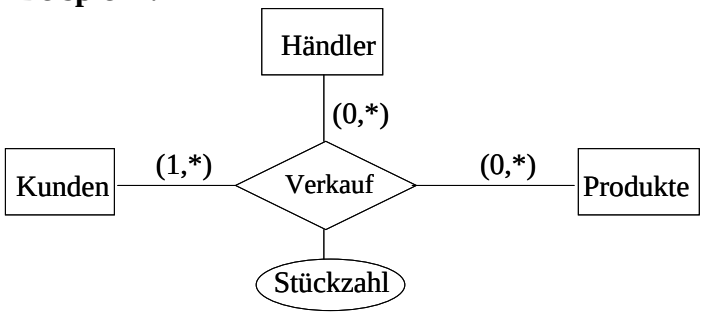
Beispiele:



Relationship-Mengen zwischen mehr als zwei Entity-Mengen

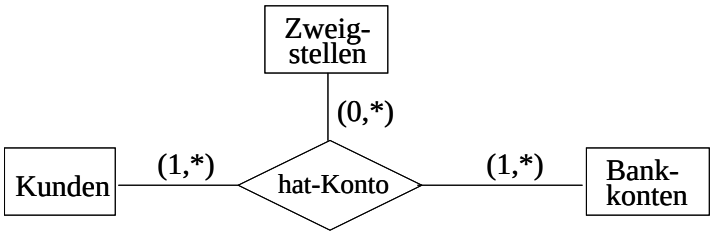
Relationship-Mengen sind nicht notwendigerweise binär. In vielen Anwendungen kommen ternäre Relationship-Mengen vor. Einige davon lassen sich in mehrere binäre Relationship-Mengen zerlegen, bei anderen ist eine solche Zerlegung nicht möglich.

Beispiel 1:

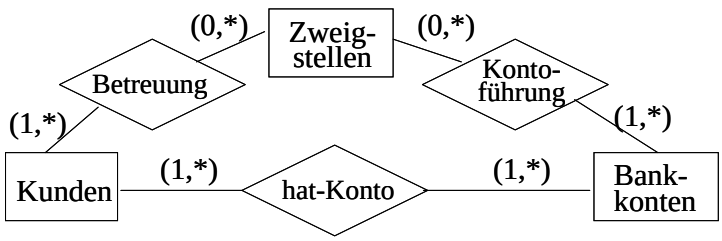


Verkauf ist nicht in mehrere binäre Relationship-Mengen zerlegbar.

Beispiel 2:



Die Relationship-Menge hat-Konto ist zerlegbar in drei binäre Relationship-Mengen.
Achtung: Die Zerlegung verändert aber die Semantik der Relationships in subtiler Weise. Die Betreuung wäre jetzt unabhängig davon, ob der Kunde ein Konto bei einer Zweigstelle hat.

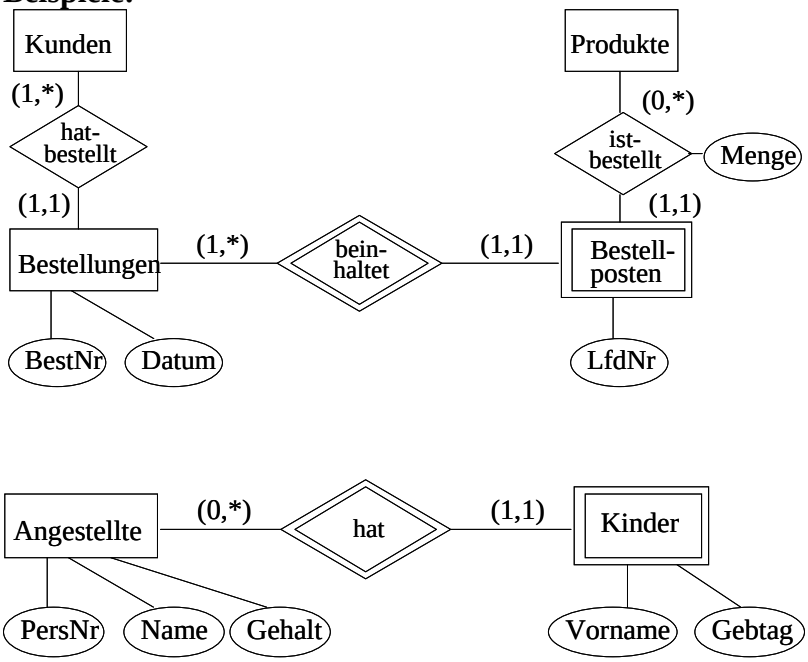


Schwache Entity-Mengen und Schwache Relationships

Eine schwache Entity-Menge S ist eine Entity-Menge, die in einer hierarchischen N:1-Relationship R zu einer anderen (nichtschwachen) Entity-Menge E steht, so daß der Primärschlüssel von S den Primärschlüssel von E enthält. R heißt dann auch schwache Relationship, wobei S eine verpflichtende Rolle in R hat.
Ein Entity einer schwachen Entity-Menge ist nur in Verbindung mit dem entsprechenden nichtschwachen Entity für die modellierte Anwendungswelt interessant.

Darstellung:
Die schwache Entity-Menge S und die schwache Relationship R werden in ERM-Diagrammen speziell markiert. Der Primärschlüssel von E wird bei S nicht explizit aufgeführt.

Beispiele:

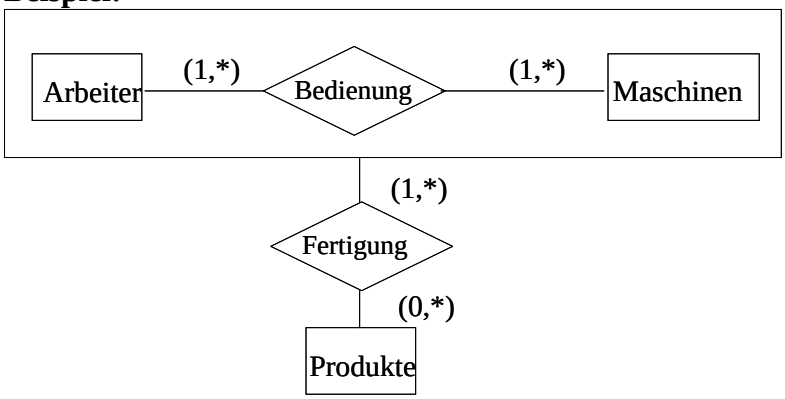


Aggregation

Mehrere Entities mit ihren Relationships können selbst wiederum zu einem *zusammengesetzten Entity* (komplexen Objekt) zusammengefaßt werden. Die entsprechende Entity-Menge heißt aggregierte Entity-Menge.

Bemerkung: Aggregation ist in vielen „semantischen“ Datenmodellen nur in eingeschränkter Form vorgesehen, wird aber z.B. von der ERM-Erweiterung SERM (Structured ERM) unterstützt.

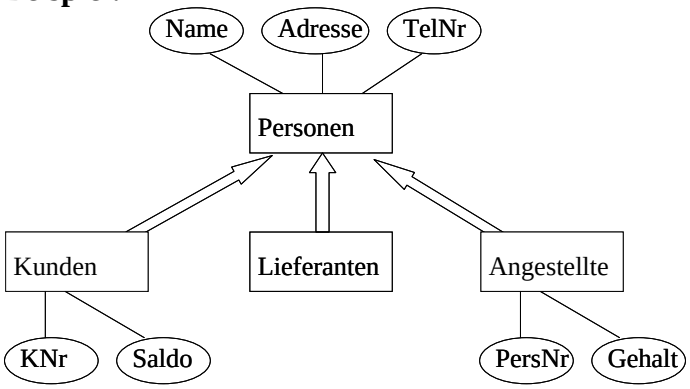
Beispiel:



ISA-Relationships (Generalisierungsbeziehungen)

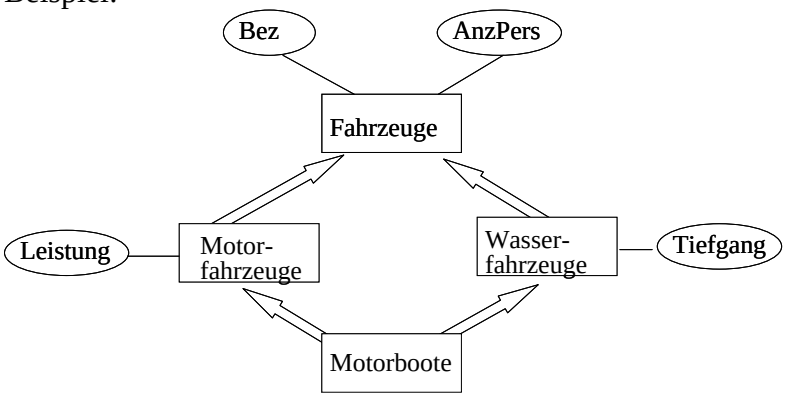
Eine binäre 1:1-Relationship-Menge R zwischen zwei Entity-Mengen E und F heißt *ISA-Relationship-Menge,* wenn die Attributmenge von F eine Obermenge der Attributmenge von E ist und die Teilmengenbeziehung $F \subseteq E$ gilt.
Man sagt dann "F is a E" und bezeichnet F als *Spezialisierung* (Subklasse) von E und E als *Generalisierung* (Superklasse) von F. Ferner bezeichnet man den Entity-Typ von F als einen Subtyp des Entity-Typs von E und den Entity-Typ von E als einen Supertyp des Entity-Typs von F.

Beispiel:



Mit der Generalisierung lassen sich ganze Hierarchien aufbauen, die häufig die morphologische Gliederung eines Anwendungsfelds modellieren.
Beispiele:
1) Wirbeltiere - Säugetiere - Katzen - Tiger - Sibirische Tiger
2) Bankkonten - Sparkonten - Jugendsparkonten
3) Finanzinstrumente - Derivative Instrumente - Optionen - Aktienindexoptionen - DAX-Optionen

Generalisierungshierarchien müssen nicht notwendigerweise Bäume sein.
Beispiel:



Verschiedene Arten von Generalisierungsbeziehungen

Seien F1, ..., Fn Spezialisierungen der Entity-Menge E.
Die Generalisierungsbeziehungen zwischen F1, ..., Fn und E heißen *disjunkt,* wenn für je zwei verschiedene Spezialisierungen Fi, Fj gilt: $Fi \cap Fj = \varnothing$.

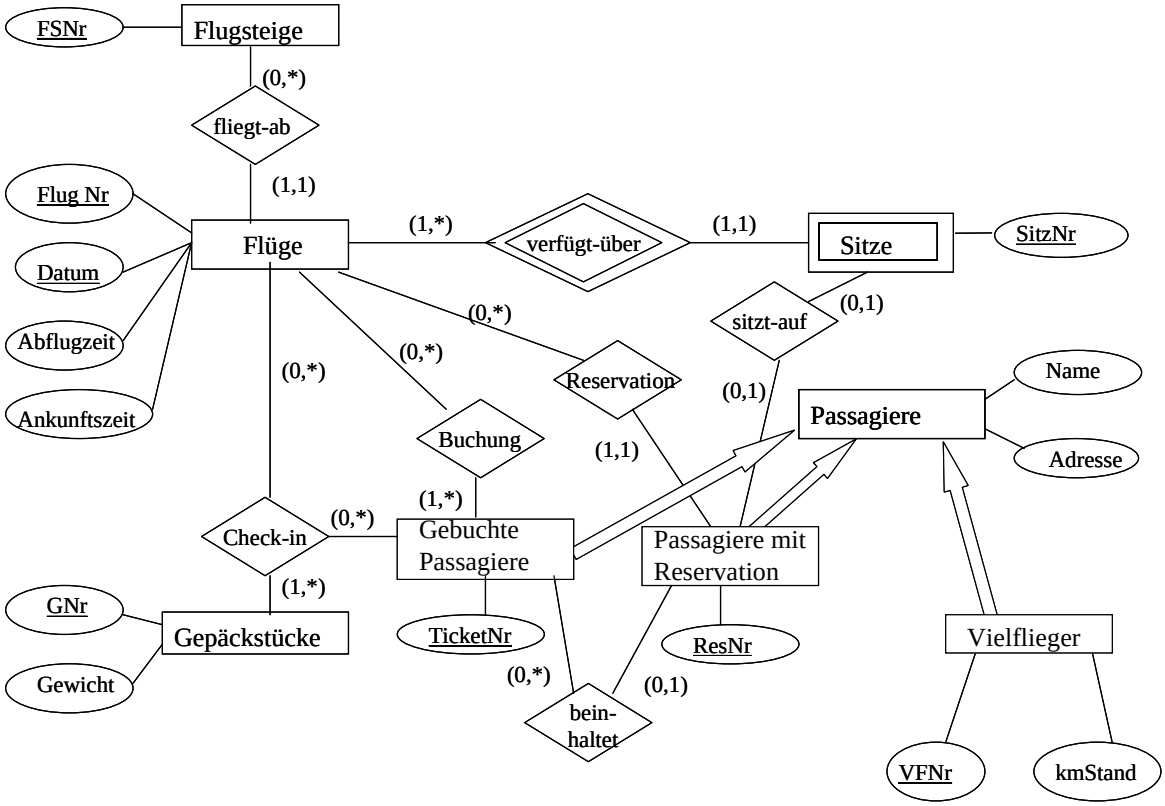
Sie heißen *vollständig-überdeckend,* wenn gilt: $\bigcup_{i=1}^{n} Fi = E$.

12.2 Entwurfsmethodologie (grob)

1) Abstecken des Problemrahmens (Analyse des Informationsbedarfs)
2) Spezifikation von Entity-Mengen mit ihren Attributen
3) Festlegung von Primärschlüssel für die Entity-Mengen
4) Bildung von Generalisierungsbeziehungen
5) Spezifikation der relevanten Relationship-Mengen
6) Spezifikation von Kardinalitätsbedingungen und ggf. weiteren Integritätsbedingungen

Beispiel:

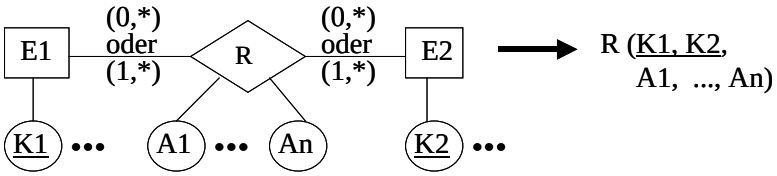
Es sollen die Informationszusammenhänge für ein Flugbuchungssystem einer Fluggesellschaft modelliert werden.
Flüge werden durch eine Flugnummer identifiziert, die für Flüge am selben Tag eindeutig ist.
Passagiere können einen Flug reservieren, was durch eine Reservationsnummer bestätigt wird. Eine Reservation wird zu einer festen Buchung, indem man ein Ticket käuft.
Bei der Reservation oder später können Passagiere auch eine Sitzplatzreservierung vornehmen.
Für Teilnehmer des Vielfliegerprogramms ist die gesamte mit der Fluggesellschaft geflogene Kilometerzahl von Bedeutung.
Flüge fliegen von einem bestimmten Flugsteig ab. Passagiere müssen vor dem Abflug eine Check-in-Prozedur durchlaufen. Dabei können sie auch Gepäckstücke aufgeben.



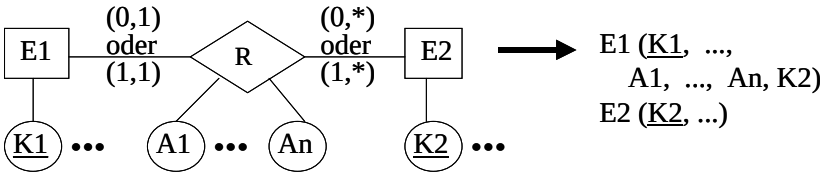
12.3 Abbildung eines ERM-Entwurfs auf Relationen

Regel 1: Jede Entity-Menge bildet eine Relation

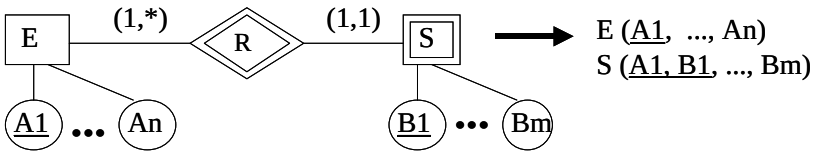
Regel 2: M:N-Relationships bilden eigene Relationen



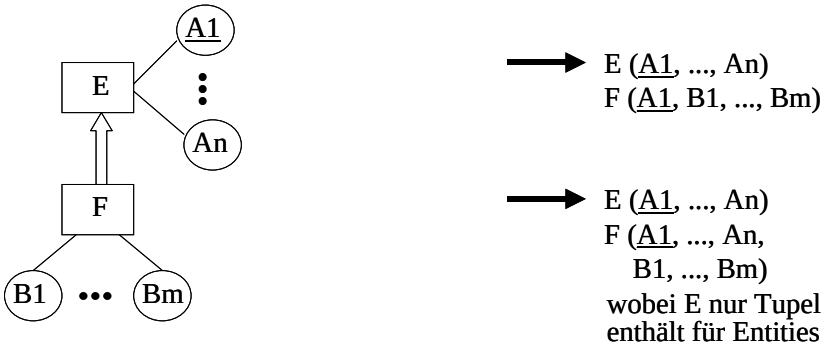
Regel 3: N:1-, 1:N und 1:1-Relationships können in eine "Entity-Relation" integriert werden



Regel 4: Eine schwache Entity-Menge bildet eine eigene Relation



Regel 5: Generalisierung und Spezialisierung bilden jeweils eine eigene Relation



Flugbuchungs-Beispiel:

Flüge (FlugNr, Datum, Abflugzeit, Ankunftszeit, FSNr)
Flugsteige (FSNr)
Sitze (FlugNr, Datum, SitzNr)
PassagiereMitReservation (ResNr, Name, Adresse, FlugNr, Datum, SitzNr, TicketNr)
GebuchtePassagiere (TicketNr, Name, Adresse)
Buchungen (TicketNr, FlugNr, Datum)
Vielflieger (VFNr, Name, Adresse, kmStand)
Gepäckstücke (GNr, Gewicht)
Check-In (FlugNr, Datum, TicketNr, GNr)

12.4 Datenmodellierung mit UML

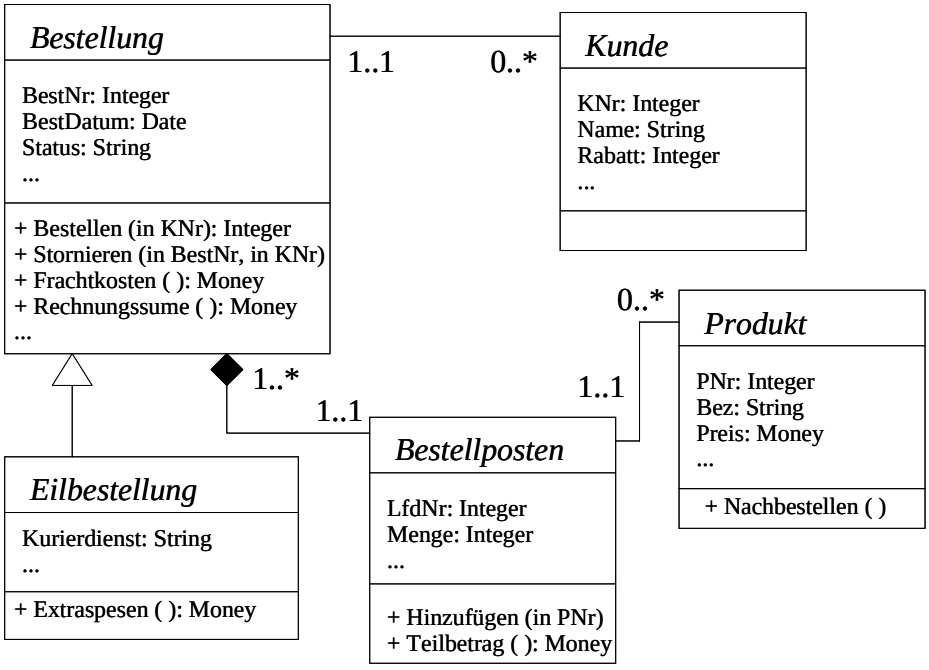
Datenbankentwürfe, die mit einer „semantischen“ Datenmodellierungsmethode, z.B. als Entity-Relationship-Diagramm, erstellt worden sind, lassen sich auch in Form von *Objektklassendiagrammen* darstellen (vgl. auch Kapitel 8 über ODMG). Für die Darstellung hat sich der Industriestandard *UML (Unified Modeling Language)* durchgesetzt. Vorteile gegenüber ERM-ähnlichen Darstellungen liegen angeblich darin, daß

1) Objektaggregationen, d.h. die Bildung komplexer Objekte, zu gekapselten Objekten und
2) anwendungsspezifische Funktionen, d.h. Methoden auf Objekten,

in einfacher Weise direkt in einem *Klassendiagramm (Class Diagram)* dargestellt werden können.

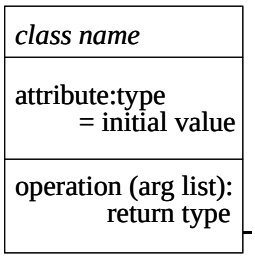
Darüber hinaus wird häufig proklamiert, daß die Fokussierung auf Objekte (und das “Zurückdrängen" von Relationships) den Entwurfsprozeß selbst besser strukturiert und zu einer übersichtlicheren Darstellung führt.

Beispiel eines Klassendiagramms nach UML:



Klassendiagramme basieren auf den folgenden Piktogrammen, deren Bedeutung jeweils nur informal definiert ist. Es gibt z.Zt. keine formal-präzise Beschreibung der Semantik aller UML-Konstrukte.

Klassen und Objekte



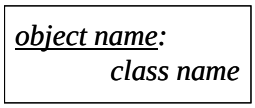
Klasse (Objekttyp, Komponente)

Attribute

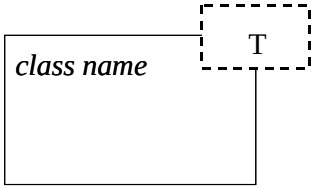
Methoden mit in-, out- und inout-Parametern
+: public, -: private, #: protected

constraints

Konsistenzbedingungen, Vor- und Nachbedingungen (constraints) in Textform

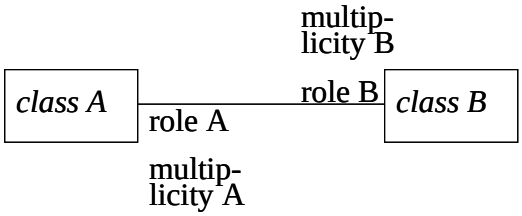


Objekt (Instanz einer Klasse)

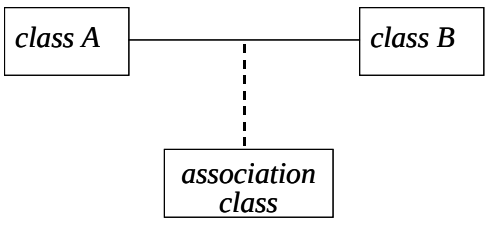


Parametrisierte Klasse (Template Class)
z.B. List <T>

Assoziationen, Aggregationen, Kompositionen

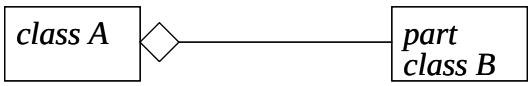


Assoziation / Relationship (assocations) zwischen Klassen mit Rollennamen und Kardinalitäten (min..max) und ggf. der Option ordered

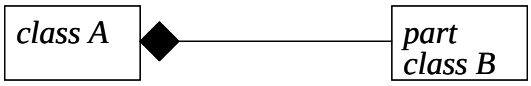


Beziehungsklasse (assocation class): entspricht den ERM-Relationships mit eigenen Attributen

Abhängigkeit (dependeny)
der Klasse A von Klasse B
(z.B. Server-API-Klasse B und
Client-Klasse A)

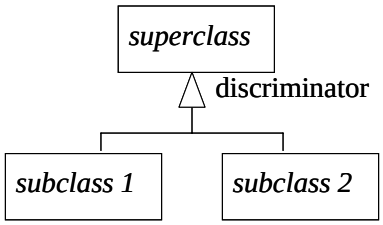


Aggregation (aggregation)
mit shared objects
(Sonderfall der Assoziation)

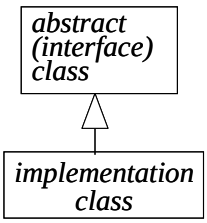


Komposition (composition)
mit non-shared objects
(Sonderfall der Assoziation)

Generalisierungen / Spezialisierungen



Generalisierungshierarchie
(generalization)

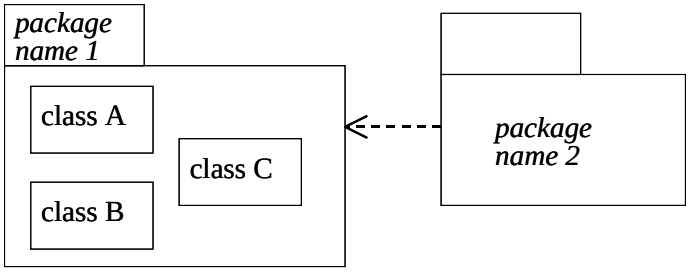


Beziehung zwischen Schnittstelle (ADT)
und Implementierung
(Sonderfall der Generalisierung /
Spezialisierung)

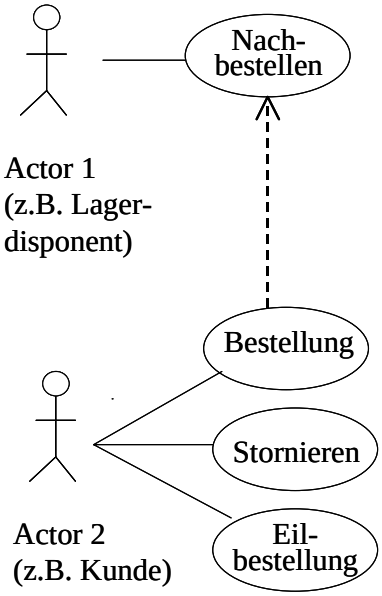
UML beinhaltet eine Reihe weiterer Diagrammarten zur Darstellung des gesamten Anwendungsentwurfs (also nicht nur der Datenaspekte). Dazu zählen:

- *Paketdiagramme (Package Diagrams)* als vergröberte Sicht auf große Klassendiagramme,
- *Anwendungsfalldiagramme (Use-Case Diagrams)* zur Grobdarstellung der Beziehung zwischen Benutzern und Funktionen in bestimmten Anwendungsabläufen,
- *Interaktionsdiagramme (Sequence Diagrams)* zur Darstellung der Methodenaufrufe zwischen Objektklassen,
- *Zustandsübergangsdiagramme (State-Transition Diagrams)* zur feineren Darstellung des Verhaltens von Objekten in bestimmten Anwendungsabläufen,
- *Aktivitätsdiagrame (Activity Diagrams)* zur Darstellung des Kontroll- und Datenflusses zwischen Objekten.

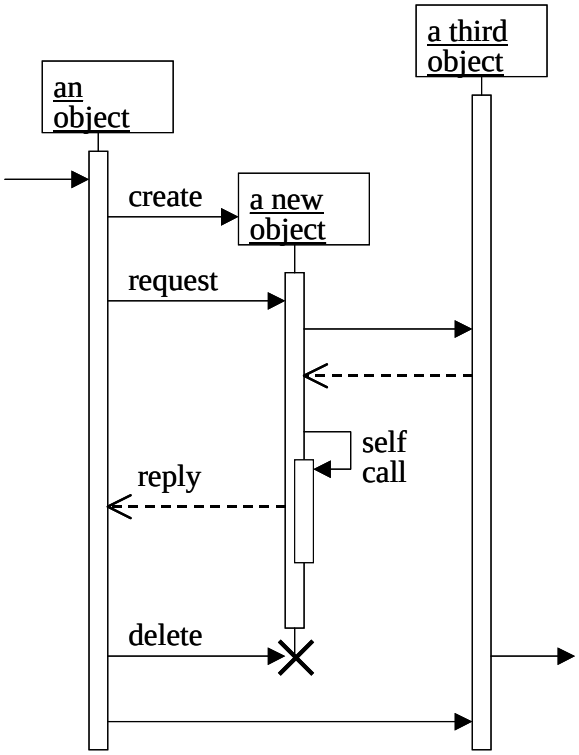
Insbesondere Zustandsübergangsdiagramme sind für die Spezifikation der “Dynamik" eines Informationssystems, also des Verhaltens der Datenobjekte im Kontext von *Geschäftsprozessen (Business Processes)* von Bedeutung (siehe Kapitel 11).



Paketdiagramm
(package diagram)



Anwendungsfalldiagramm
(use-case diagram):
Zusammenhang zwischen
Benutzern bzw. Rollen
und Funktionen / Ereignissen



Interaktionsdiagramm
(sequence diagram):
Nachrichtenfluss zwischen
Objekten (bzw. Threads)

Insgesamt dient UML als Beschreibungsmittel für den gesamten Entwurfs- und Entwicklungsprozeß von der informellen Kommunikation zwischen Anwendern, Fachkonzeptexperten und Informatikern in den frühen Phasen der Anforderungsanalyse und des Grobentwurfs bis hin zur Dokumentation der Implementierung. Dies erklärt die etwas barocke Vielfalt der Diagrammtypen und auch den erkennbaren Mangel an formaler Durchdringung. An der Beschreibung einer präzisen Semantik für die im Feindesign und der Implementierung wichtigen UML-Konstrukte wird aktiv gearbeitet.

12.5 Objektorientierte Analyse- und Entwurfsmethodologie

Für die Anforderungsanalyse und den Anwendungsentwurf wird - basierend auf der objektorientierten Darstellungsmethode (insbesondere den Klassendiagrammen) - häufig auch eine bestimmte *Vorgehensmethodologie* verfochten. Diese stellt - mehr oder weniger grobe - Richtlinien zusammen, in welchen Schritten die Analyse bzw. der Entwurf vorangetrieben werden sollte. Die wohl bekannteste unter diesen Schulen ist die *Object Modeling Technique (OMT)* von Rumbaugh, deren diagrammbasierte Notation zusammen mit den Notationen von Booch und Jacobson zum Industriestandard UML geführt hat.

OMT propagiert die folgenden Analyse- bzw. Entwurfsschritte, die zur schrittweisen Verfeinerung ineinander verzahnt iterierbar sind:

1) *strukturelle Modellierung* der Datenobjekte:
   Objektklassen mit ihren Assoziationen, insbesondere Aggregationen
2) *Dynamikmodellierung*:
   Zustandsübergangsdiagramme zur Darstellung des dynamischen Verhaltens von Objekten bzw. Geschäftsprozessen
3) *Funktionsmodellierung*:
   Beschreibung der Objektmethoden, ihrer gegenseitigen Aufrufe und des Datenflusses zwischen ihnen

Diese Grobstruktur kann weiter detailliert werden, bleibt jedoch letzten Endes nur ein Leitfaden und darf keinesfalls als Patentlösung für die Praxis angesehen werden.

Werkzeuge

Es gibt etliche Softwarewerkzeuge (z.B. Erwin, Rational Rose, etc.), mit denen ERM- oder UML-artige Modelle graphisch erstellt werden können; diese Werkzeuge bilden solche Modelle automatisch auf relationale oder objektorientierte Schemata oder Interface-Beschreibungen ab und können u.a. SQL-Anweisungen für Create Table inkl. Constraints generieren.

Ergänzende Literatur zu Kapitel 12:

Teorey, T.J., Database Modelling and Design, Morgan Kaufmann, 1999

Batini, C., Ceri, S., Navathe, S.B., Conceptual Database Design, Benjamin/Cummings, 1992

Muller, R.J., Database Design for Smarties: Using UML for Data Modeling, Morgan Kaufmann, 1999

Fowler, M., Scott, K., UML Distilled, Addison-Wesley, 2000

Maciaszek, L.A., Requirements Analysis and System Design: Developing Information Systems with UML, Addison-Wesley, 2001

Object Management Group (OMG), Unified Modeling Language (UML) Documentation, http://www.omg.org/technology/uml/index.htm oder http://www.rational.com/uml/index.jsp

Scheer, A.-W., ARIS - Business Process Modeling, 2nd Edition, Springer-Verlag, 1999

Balzert, H., Lehrbuch der Software-Technik, Band 1: Software-Entwicklung, Spektrum-Verlag, 1996

Kapitel 13: Prozeßmodellierung und Workflow-Management

13.1 Prozessmodellierung

Beim Entwurf von Informationssystemen müssen außer den Daten auch Funktionen und Prozesse der Anwendung modelliert werden. Unter Funktionen verstehen wir dabei elementare Bausteine der Anwendungsfunktionalität aus Benutzersicht, also mächtige Methoden auf „Business Objects“ wie z.B. die betriebswirtschaftliche Verbuchung einer Bestellung oder die Prämienberechnung für eine Versicherungspolice. Mehrere solcher Funktionen können zu einem Geschäftsprozeß kombiniert werden, und ein Geschäftsprozeß kann u.U. selbst wieder als umfassendere Funktion angesehen werden. Die primitiveren Funktionen der Anwendung schließlich werden auf Methoden der Datenobjekte und programmiertechnische Klassen abgebildet, die z.B. mit UML modelliert werden. Entscheidend für die Anpassungsfähigkeit eines Unternehmens (oder auch einer Behörde) an die Dynamik der sich ständig wandelnden Kundenbedürfnisse und Marktanforderungen ist eine saubere Trennung von Funktionen, den betrieblichen Bausteinen, und Prozessen, den angepaßten Abläufen im Unternehmen und auch (z.B. in Logistikketten) gegenüber anderen Unternehmen.

Die Computerunterstützung betrieblicher Abläufe kann sich auf die einzelnen Funktionen beschränken, so daß die Koordination langlebiger Geschäftsprozesse beim menschlichen Sachbearbeiter verbleibt, oder sie kann auch die (weitgehend) automatisierte Steuerung der Prozesse umfassen. Für den zweiten Fall hat sich der Begriff *Workflow-Management* eingebürgert. Beispielanwendungen sind etwa die Bearbeitung von Kreditanträgen in einer Bank oder Schadensfällen in einer Versicherung, die Planung und Durchführung einer Marketingkampagne, die elektronische Einreichung, Begutachtung und Publikation wissenschaftlicher Zeitschriftenartikel oder - heute noch fiktiv - die elektronische Einreichung und automatisierte Verarbeitung von Steuererklärungen oder die Abwicklung aller bei Immobilienkäufen notariell, behördlich sowie finanz- und versicherungstechnisch erforderlichen Schritte. Derartige Workflows bestehen aus mehreren, automatisierten oder intellektuellen Arbeitsschritten - in der Literatur häufig *Aktivitäten* genannt -, die verschiedenen Ausführungsorganen - Sachbearbeitern oder Computerapplikationen - zugeordnet werden. Die von Aktivitäten aufgerufenen Applikationsprogramme sind die eigentlichen Funktionen der Anwendung, d.h. Aufrufe von Methoden auf *„Business Objects“* wie zum Beispiel Konto- oder Bestellungs-Objekten. Insofern ergibt sich aus der objektorientierten Datenmodellierung und der funktionsorientierten Modellierung von Geschäftsprozessen eine integrierte Darstellung der Anwendung.

In den folgenden Abschnitten werden verschiedene Spezifikationsmethoden für die Modellierung von Prozessen vorgestellt.

13.1.1 Prozeßmodellierung mit State-Charts (bzw. State– und Activity–Charts)

Der State– und Activity–Chart–Formalismus wurde in Arbeiten von David Harel (ursprünglich für “reaktive", eingebettete Steuerungssysteme) entwickelt und inzwischen in den Industrie–Standard UML übernommen (in UML heißen sie Zustandsübergangsdiagramme, englisch: State Transition Diagrams). State– und Activity–Charts stellen duale Sichtweisen einer Spezifikation dar.

Ein *Activity–Chart* spezifiziert den Datenfluß zwischen den Aktivitäten in Form eines gerichteten Graphen mit Aktivitäten als Knoten und Datenelementen als Kantenbeschriftung.

(Achtung: Diese Art von Activity-Chart hat nichts mit den Aktivitätsdiagrammen von UML zu tun. Letztere sind eine Variante von Petrinetzen und beschreiben Kontrollfluß.)

Ein *State–Chart* beschreibt das Verhalten einer Spezifikation, indem es den Kontrollfluß zwischen Aktivitäten als Transitionen zwischen Zuständen spezifizieren. Ein State–Chart ist im wesentlichen ein endlicher Automat mit einem ausgewiesenen Anfangszustand und Transitionen, die mit *Event–Condition–Action Regeln (ECA–Regeln)* beschriftet sind. Eine Transition von Zustand *X* nach Zustand *Y*, die mit der ECA–Regel *E[C]/A* beschriftet ist, feuert genau dann, wenn sich der Automat in Zustand *X* befindet, das Ereignis *E* eintritt und die Bedingung C erfüllt ist. Der Automat wechselt dann in den Zustand *Y* und die Aktion *A* wird ausgeführt. Bedingungen sind aussagenlogische Formeln über elementaren Bedingungen auf Zuständen (z.B. *in(S)*, die wahr ist, wenn Zustand S betreten ist) und - kontrollflußrelevanten - globalen Variablen (u.a. denjenigen Variablen, die für die Datenflußspezifikation im Activity-Chart verwendet werden). Eine Aktion kann sowohl explizit eine Aktivität starten (durch den Ausdruck *st!(aktivität))*, als auch Ereignisse erzeugen oder Bedingungen setzen (z.B. *fs!(C)* setzt die Bedingung *C* auf den Wert falsch). Jeder der drei Teile des *E[C]/A*–Tripels kann auch leer sein.

Zwei weitere Besonderheiten unterscheiden State–Charts von herkömmlichen Zustandsautomaten. Zum einen ist dies die im Hinblick auf schrittweise Verfeinerung und Abstraktion wichtige Möglichkeit zur *Schachtelung von Zuständen*. Ein Zustand kann ein komplettes State–Chart beinhalten, wobei beim Betreten eines übergeordneten Zustands implizit der jeweilige initiale Unterzustand betreten wird und beim Verlassen eines Elternzustandes alle betretenen Kinderzustände verlassen werden. Zum anderen erlauben State–Charts *orthogonale Zustandskomponenten*, mit denen Parallelität ausgedrückt wird, so daß Transitionen in verschiedenen orthogonalen Komponenten gleichzeitig feuern können.

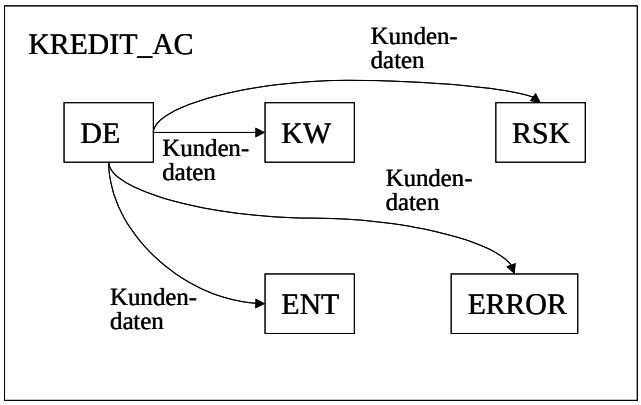
Einfaches Beispiel 1: Simplifizierte Kreditantragsprüfung

Eine simplifizierte Kreditantragsprüfung bestehe aus den Aktivitäten
- Dateneingabe (DE) zur Erfassung der Kundendaten (KD),
- Kreditwürdigkeitsprüfung (KW) des Kunden,
- Risikoabschätzung (RSK) aufgrund des Betrags, der Währung und Zahlungsart des Kredits,
- Entscheidung (ENT) über den Kreditantrag und einer
- generischen Fehlerbehandlungsaktivität (ERROR).

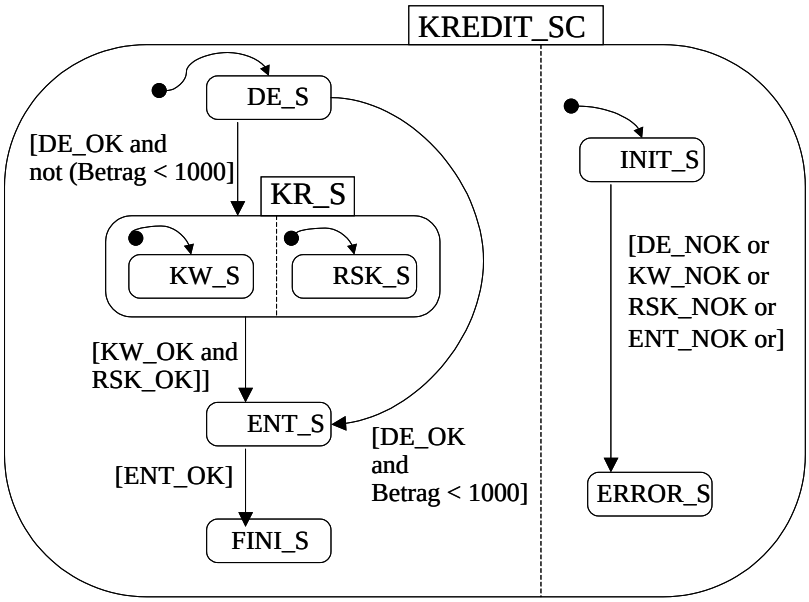
Die Aktivitäten KW und RSK sind nur bei Krediten über einer bestimmten Höhe notwendig; sie sind außerdem parallel ausführbar.

Jede Aktivität ist genau einem Zustand zugeordnet und soll mit dem Betreten des Zustands gestartet und mit dem Verlassen des Zustands beendet werden. Die Aktivitäten können als Methoden eines Objekts “Kreditantrag" angesehen werden. Insofern beschreiben State- und Activity-Charts das dynamische Verhalten eines solchen Objekts.

Activity-Chart:



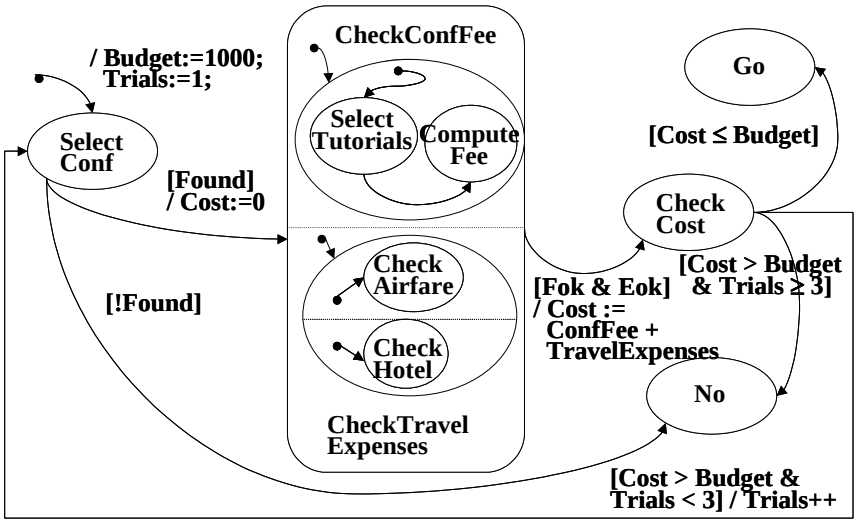
State-Chart:



Einfaches Beispiel 2: Planung einer Konferenzreise

Bei der – fiktiven - Planung einer Konferenzreise (z.B. eines wissenschaftlichen Mitarbeiters eines Universitätslehrstuhls) wird zunächst eine thematisch interessante Konferenz gewählt. Dann werden parallel die Konferenzgebühren und die Reisekosten für Hotel und Flug ermittelt. Wenn die Gesamtkosten im Rahmen des vorgesehenen Budgets liegen, fällt eine positive Entscheidung für die Konferenzteilnahme (Endzustand „Go"); ansonsten werden bis zu drei alternative Konferenzen analog evaluiert. Nach drei erfolglosen Versuchen, gibt man auf (Endzustand „No").

Statechart:



Komplexeres Beispiel: Bestellungen via Internet (Electronic Commerce)

Kontrollfluß. Der Workflow beginnt mit einer Aktivität, die eine *neue Bestellung* in einer Datenbank erfaßt. Das Versandhaus stellt zwei Arten von Zahlungsmodalitäten zur Verfügung. Der Kunde kann entweder bei der Bestellung seine Kreditkartennummer angeben oder eine Rechnung anfordern, nach deren Erhalt er die angefallene Rechnungssumme überweisen muß. Der Kontrollfluß muß an dieser Stelle aufgespalten werden. Wurde eine *Kreditkartenzahlung* gewünscht, wird zunächst eine Aktivität zur Bonitätsprüfung der Kreditkarte gestartet. Nach der Prüfung der Kreditkarte werden die beiden Kontrollfluß–Pfade wieder zusammengeführt.

Nach der Registrierung der Bestellung werden der *Versand* sowie das Senden einer *Bestätigungs–Mail* an den Kunden initiiert. Dazu werden parallel zwei Sub–Workflows gestartet. Einer der beiden Sub–Workflows startet eine Aktivität, die eine Bestätigung an den Kunden versendet. Das Versenden von Waren erfolgt extern aus einem oder mehreren Lagern, die u. U. sogar von separaten Großhändlern autonom verwaltet werden. Daher startet der zweite Sub–Workflow zunächst eine Aktivität, die für jedes Lager die auszuliefernden Artikel bestimmt. Die Zuordnung wird in Artikellisten festgehalten. Innerhalb eines jeden Lagers, aus dem Artikel für die aktuelle Bestellung ausgeliefert werden müssen, wird eine Anwendung gestartet, die als Parameter die Artikelliste erhält und eine Versendegarantie zurückliefert. In einer Schleife werden so oft einzelne Lager angesprochen, bis alle Artikel auf der Bestellung bearbeitet wurden und die Versendung garantiert wurde. Wenn beide Subworkflows beendet sind, wird der Kontrollfluß erneut aufgespalten, abhängig davon, welche Zahlungsmodalität gewünscht wurde. Wurde Zahlung per Kreditkarte gewählt, wird die Kreditkarte nun belastet und der Workflow beendet. Auf diesem Kontrollfluß–Pfad werden keinerlei menschliche Eingriffe benötigt.

Soll eine Rechnung gestellt werden, könnte zusätzlich eine Aktivität gestartet, die auf den *Eingang der Zahlung* wartet. Diese Aktivität wäre intellektuell–interaktiver Natur, da die Eingabe von Daten, z.B. ob der Rechnungsbetrag vollständig eingegangen ist, durch einen Mitarbeiter des Versandhauses erfolgt. Ist die Zahlung erfolgt, ist das Szenario beendet. Sollte die Rechnung nicht innerhalb einer Frist von 14 Tagen beglichen sein, wird dem Kunden eine Mahnung geschickt. Dieser Vorgang wiederholt sich maximal dreimal im Abstand von 14 Tagen, bis die Rechnung beglichen ist. Wurde die Zahlung 14 Tage nach der dritten Mahnung noch immer nicht geleistet, so wird die Rechtsabteilung des Versandhauses informiert. Das Mahnverfahren könnte mit einer Schleife spezifiziert werden. Innerhalb der Schleife würde eine Aktivität gestartet, die eine Mahnung versendet, der Stichtag für das Versenden der nächsten Mahnung ermittelt und der Zähler für die bereits versendeten Mahnungen inkrementiert.

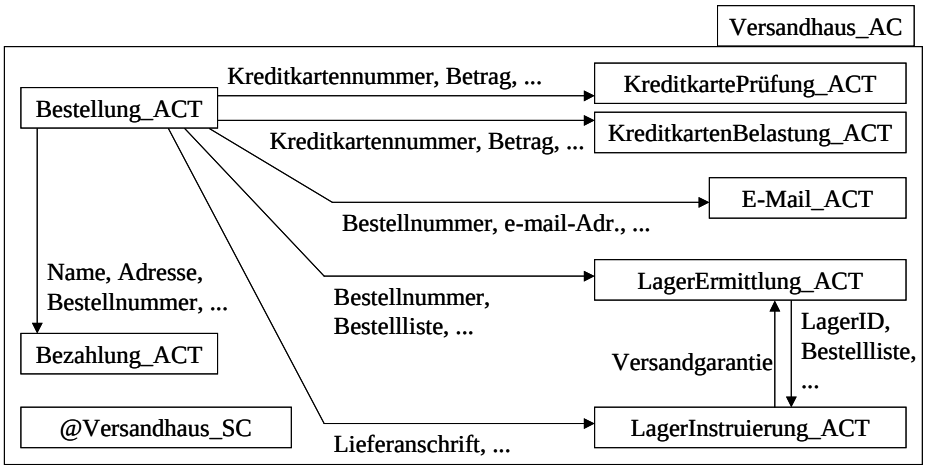
Datenfluß. Alle Eingabedaten werden in der Aktivität gesammelt, die die Bestellung erfaßt, und von ihr (bedarfsweise) an die übrigen Aktivitäten verteilt. Es ergibt sich ein sternförmiger Datenfluß, ausgehend von dieser Aktivität. Lediglich innerhalb der Schleife, die die Artikel auf die Lager verteilt, gibt es zusätzlichen Datenfluß, nämlich die jeweiligen Artikellisten in die eine und die Versandgarantien in die andere Richtung.

Spezifikation als State– und Activity-Chart (ohne das Mahnverfahren). Im State-Chart wird der Übersichtlichkeit halber auf die Darstellung von Ausnahmebehandlungen verzichtet (z.B. Nichtlieferbarkeit bestellter Artikel). Initial wird die Aktivität *BestellungErfassen* gestartet (hier explizit mit der Aktion st!(Bestellung_ACT)) und der Workflow geht in den Zustand *NeueBestellung_S* über. Die Bedingung *KreditkartenZahlung* spezifiziert die gewünschte Zahlungsmodalität und wird von der Aktivität *BestellungErfassen* gesetzt. Ist die Bedingung wahr, wird die Aktivität *KreditkartePrüfen* gestartet, die bei ihrer Terminierung die Bedingung *KreditkarteOk* auf wahr oder falsch setzt. Schlägt der Test fehl, wird der Workflow beendet, ansonsten geht er in den geschachtelten Zustand *Versand_S* über. Innerhalb dieses geschachtelten Zustandes werden zwei Sub–Workflows ausgeführt. Die Parallelität der Ausführung wird durch die gestrichelte Linie spezifiziert. In den Sub–Workflows werden die Aktivitäten *Bestätigen*, *LagerErmitteln* und *LagerInstruieren* gestartet. Sie setzen jeweils die Bedingungen *BestätigungGesendet*, *LagerErmittelt* und *VersandGarantiert* auf wahr. Dabei durchlaufen die Sub–Workflows die Zustände *Bestätigen_S*, *LagerErmitteln_S* und *LagerInstruieren_S* und terminieren in den Zuständen *BestätigungGesendet_S* beziehungsweise *Abgeschickt_S*.

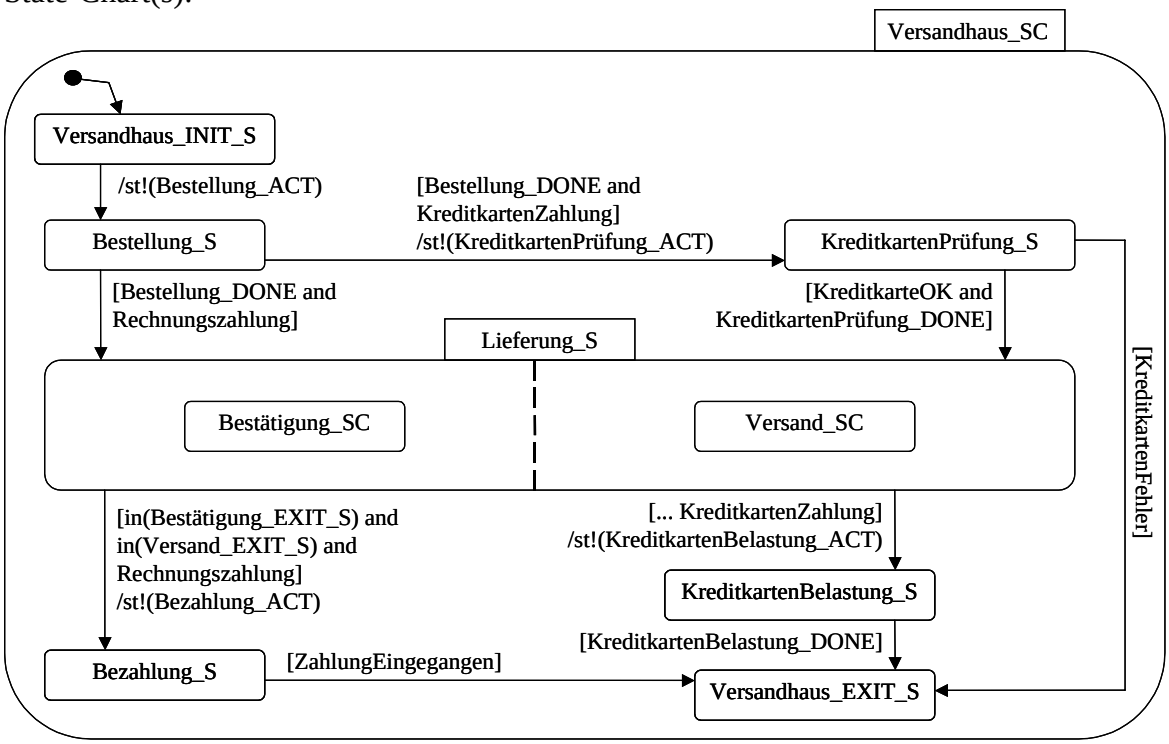
Haben beide Sub–Workflows ihren Endzustand erreicht, angezeigt durch die Bedingungen *in(Abgeschickt_S)* und *in(BestätigungGesendet_S)*, wird der geschachtelte Zustand *Versand_S* verlassen. Abhängig vom Wert der Bedingung *KreditkartenZahlung* wird die Kreditkarte durch Starten der Aktivität *KreditkarteBelasten* belastet und der Workflow endet im Zustand *Ende_S*, oder die Aktivität *Bezahlung* wird gestartet und der Workflow geht in den Zustand *Bezahlung_S* über. Bei der Aktivität *Bezahlung* wird die Bedingung *ZahlungEingegangen* auf wahr gesetzt, sobald die Rechnung beglichen wurde. Dadurch geht der Workflow in den Zustand *Ende_S* über.

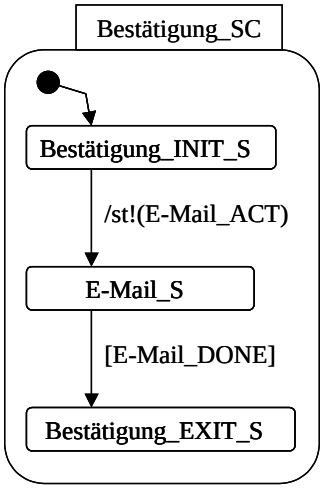
Das Activity–Chart enthält alle an dem Workflow beteiligten Aktivitäten. Alle Eingabedaten werden in der Aktivität *BestellungErfassen* gesammelt und von ihr bedarfsweise an die übrigen Aktivitäten verteilt. Zusätzlich übermittelt die Aktivität *LagerErmitteln* der Aktivität *LagerInstruieren* eine *LagerID* mit der zugehörigen Artikelliste. Die Aktivität *LagerInstruieren* schickt an ihrem Ende der Aktivität *LagerErmitteln* die erhaltenen Versandgarantien zurück.

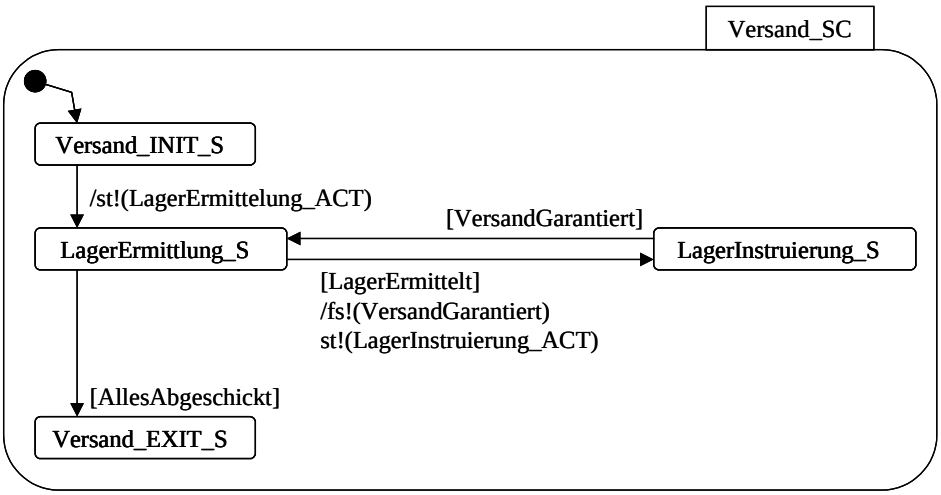
Activity-Chart:



State-Chart(s):



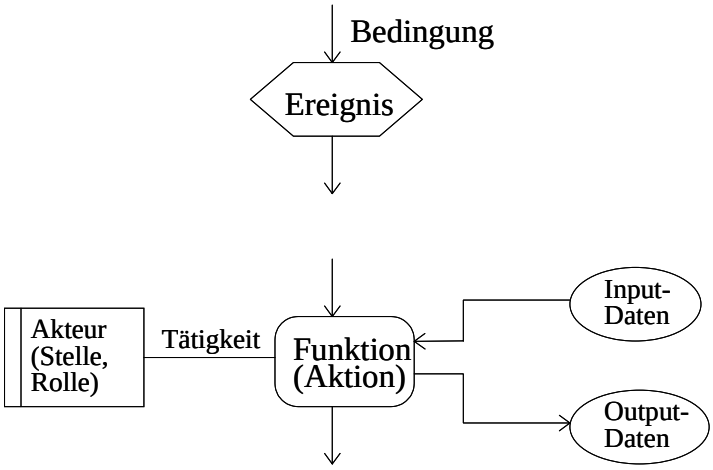


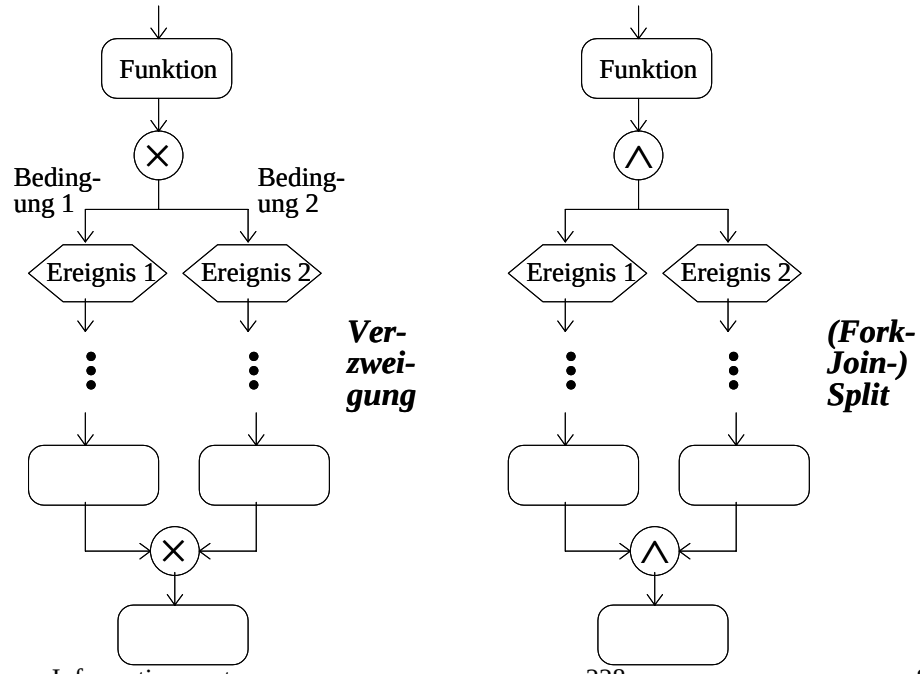


13.1.2 Prozeßmodellierung mit Ereignis-Prozeß-Ketten (EPKs)

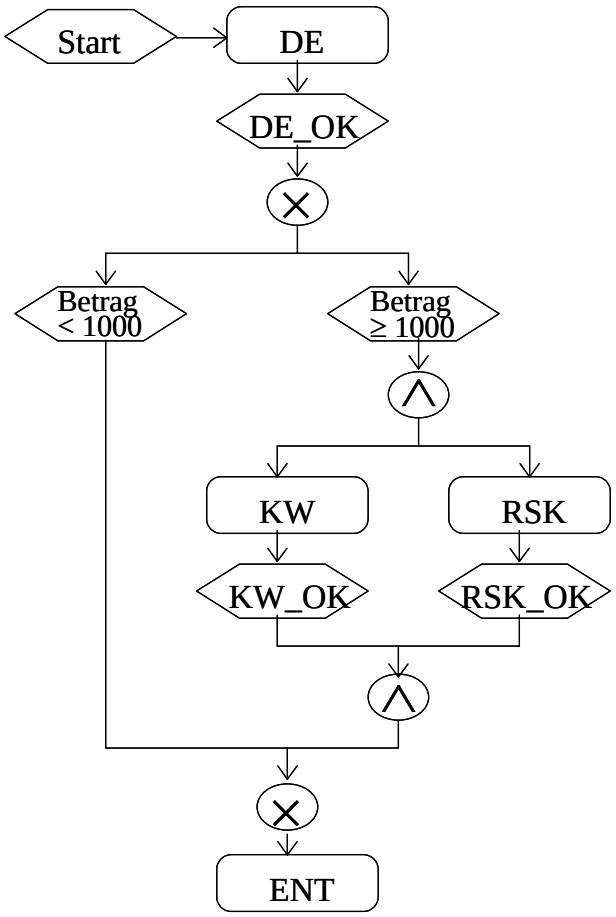
Zur Modellierung von Geschäftsprozessen gibt es zahlreiche weitere “semiformale" oder auch stärker formalisierte Methoden, z.B. solche, die auf Varianten von Petrinetzen beruhen. Eine in der Praxis weit etablierte Methode ist die der *Ereignis-Prozeß-Ketten (Event-Process Chains), kurz: EPKs*. Dabei werden Aktivitäten mit Kontrollfluß-Konnektoren der Typen XOR (Verzweigung), AND (Parallelausführung) und OR (Nichtdeterminismus) verknüpft. Sofern die logischen Bedingungen für den Kontrollfluß, z.B. Verzweigungsbedingungen, hinreichend präzise und formal spezifiziert werden, lassen sich EPKs automatisch in State-Charts übersetzen. Dies kann nützlich sein, wenn Anwender beispielsweise mit EPKs zunächst grob modellieren, dann schrittweise verfeinern und präzisieren, bis sie schließlich eine übersetzbare Spezifikation erhalten, die sie schlußendlich dem StateChart-Interpreter eines Workflow-Management-Systems zur Ausführung übergeben können. EPKs können über die Beschreibung des Kontroll- und Datenflusses hinaus auch organisatorische Zuordnungen der Aktivitäten (Funktionen) zu Abteilungen, Sachbearbeiterrollen, etc. in Form von Annotationen darstellen.

Die wichtigsten grafischen Elemente von EPKs sind im folgenden kurz zusammengestellt:





Einfaches Beispiel eines EPK-Modells (Kreditantragsprüfung) unter Weglassung von Ausnahmebehandlungen, Datenfluß, und Aspekten der Arbeitsorganisation:

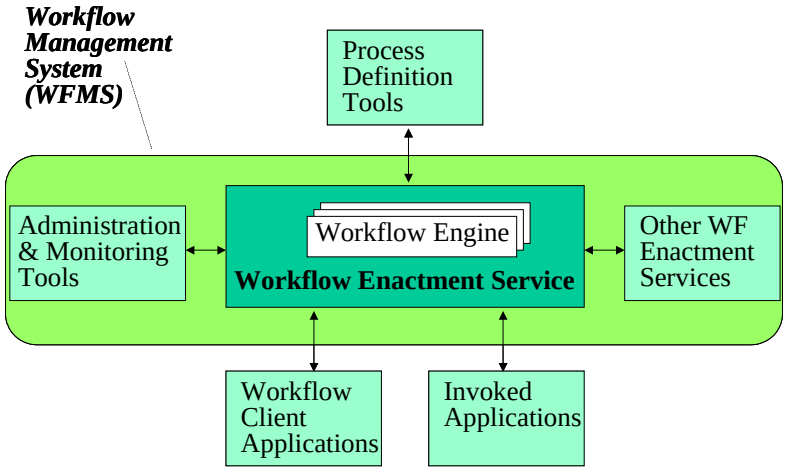


13.2 Workflow-Management für Geschäftsprozesse

Heute verfügbare *Workflow-Management-Systeme (WFMS)* unterstützen die automatische Steuerung von Geschäftsprozessen auf drei Ebenen:

- durch eine Methode zur *Spezifikation des Kontroll- und Datenflusses* zwischen den Aktivitäten eines Workflows,
- durch eine *Workflow-Engine zur Ausführung von Workflows* (im wesentlichen durch Interpretation der Workflow-Spezifikation) und damit systemunterstützten Koordination der Aktivitäten sowie
- durch eine Palette von *Administrationswerkzeugen* zur Simulation, Analyse und Visualisierung von Workflows sowie zur Festlegung und Überwachung von Regeln für die Zuteilung von Arbeitsschritten zu den Ausführungsorganen (dem sog. Worklist-Management zur dynamischen Rollenauflösung).

Die typische Architektur eines WFMS sieht folgendermaßen aus:



Dies ist eine Referenzarchitektur, die von der sog. Workflow Management Coalition (WfMC), einem Industriekonsortium, entwickelt wurde. Kommerzielle Produkte entsprechen mehr oder weniger diesem Framework. Führende WFMS-Produkte sind z.B. IBM MQ Series Workflow (Teil von IBM WebSphere), Staffware, MS BizTalk, HP E-Speak, BEA WebLogic, SAP Workflow, SNI WorkParty, Dialogika MultiDesk, etc.

13.3 Semantik von Statecharts

13.3.1 Operationale Semantik

Im folgenden wird eine operationale Semantik einer simplifizierten Variante von Statecharts formal entwickelt, also das dynamische Verhalten einer Statechart-Spezifikation mathematisch beschrieben. Zur Vereinfachung werden Events nicht betrachtet, und es wird angenommen, daß die einem Zustand entsprechende Aktivität jeweils mit Betreten des Zustands gestartet und mit dem Verlassen des Zustands beendet wird.

Die formale Semantik bildet die Grundlage für die Analyse und Verifikation von Prozeßspezifikationen. Dabei ist man an Invarianten des Verhaltens eines Prozesses unabhängig von seinen Eingaben interessiert. Diese Invarianten gehören in der Regel zu einer der folgenden Kategorien:

- *Sicherheitseigenschaften (safety properties):* zu jedem Zeitpunkt gelten bestimmte wünschenswerte Eigenschaften; unzulässige Prozeßzustände werden immer vermieden („es passiert nie etwas Schlimmes“).
- *Lebendigkeitseigenschaften (liveness properties):* ab einem bestimmten Zeitpunkt wird eine wünschenswerte Eigenschaft erreicht; Zielzustände des Prozesses werden irgendwann erreicht („es passiert schlußendlich etwas Gutes“).

Im Konferenzreiseplanungsbeispiel etwa wäre das Nichtüberschreiten des Budgets eine Sicherheitseigenschaft und das Terminieren des Entscheidungsprozesses (also Erreichen eines Endzustands) eine Lebendigkeitseigenschaft.

Definition:

Ein ***Statechart*** ist ein Tripel (S, V, T), wobei

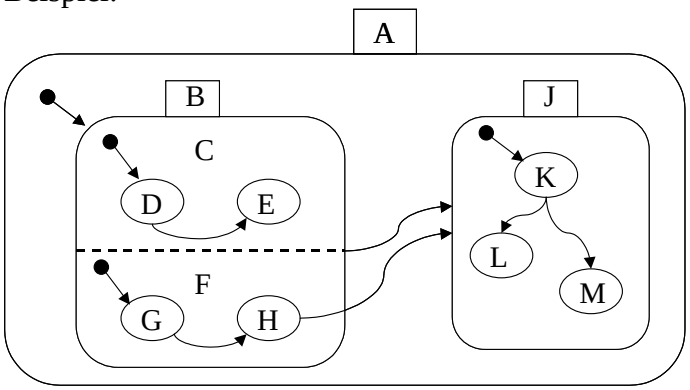
- S eine endliche Menge von ***Zuständen*** ist,
- V eine endliche Menge von typisierten ***Variablen*** ist (mit den Typen Boolean, Integer, etc.),
- T eine endliche Menge von ***Transitionen*** ist, wobei jede Transition ein Paar von Zuständen aus S verbindet oder eine initiale Transition in einen Zustand ist und mit einer Condition-Action-Regel der Form [C] / A annotiert ist. Für $t \in T$ bezeichnet source(t) den Ausgangszustand und target(t) den Zielzustand. Bei einer ***initialen Transition*** ist source(t) undefiniert.

In einer ***Condition-Action-Regel*** [C] / A ist

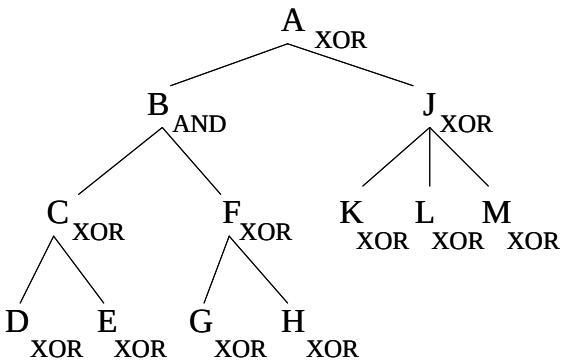
- C eine aussagenlogische Formel (mit den Junktoren and, or, not) über elementaren Vergleichen der Form Var1 θ Wert oder Var1 θ Var2 (Var1, Var2 $\in$ V) mit einem Vergleichsoperator θ aus $\{=, \neq, <, >, \leq, \geq\}$ oder der Form in(s) mit einem Zustand $s \in S$ und
- A eine Liste von Zuweisungen der Form Var := expr (Var $\in$ V) mit einem – je nach Typ von Var – arithmetischen oder Booleschen (oder sonstigem) Ausdruck expr.

Die Zustandsmenge S bildet einen Baum, bei dem $s1 \in S$ Vater von $s2 \in S$ ist, wenn s2 ein direkter Unterzustand von s1 ist. Ein Zustand ist vom Typ AND, wenn seine Kinder orthogonale Komponenten sind, ansonsten vom Typ XOR. Für jeden XOR-Zustand wird verlangt, daß unter seinen Kindern genau eines Ziel einer initialen Transition ist.

Beispiel:



Zustandsbaum:



Definition:

Ein ***Ausführungszustand*** states eines Statecharts ist eine Teilmenge von S, die Menge der aktuell betretenen Zustände. Dabei muß gelten:

- Die Wurzel von S ist in jedem Ausführungszustand states enthalten.
- Wenn s in states enthalten ist und s ist vom Typ AND, dann sind alle Kinder von s in states.
- Wenn s in states enthalten ist und s ist vom Typ XOR, dann ist genau ein Kind von s in states.

Ein ***Ausführungskontext*** eines Statecharts ist eine Wertebelegung der Variablen in V, d.h. eine (typgerechte) Abbildung val: V $\rightarrow$ Dom, wobei Dom die Menge aller möglichen Werte aller vorgesehenen Typen ist.

Eine (Ausführungs-) ***Konfiguration*** conf eines Statecharts ist ein Paar (states, val) von Ausführungszustand und Ausführungskontext.

Die initiale Konfiguration $conf_0$ hat die Zustandsmenge states, die

- die Wurzel von S enthält,
- für jeden AND-Zustand s in $conf_0$ alle Kinder von s und
- für jeden XOR-Zustand s in $conf_0$ das Kind von s, das Ziel einer initalen Transition ist,

und den Ausführungskontext mit val(Var1) = false für alle Var1 $\in$ V vom Typ Boolean und val(Var2) = 0 für alle Var2 $\in$ V vom Typ Integer usw.

Definition:

Die Auswertung eval(expr, conf) eines Ausdrucks expr in einer Konfiguration conf = (states, val) ist induktiv wie folgt definiert:

- eval (c, conf) = c für alle Konstanten c
- eval (not expr, conf) = not eval(expr, conf),
- eval (expr1 and expr2, conf) = eval(expr1, conf) and eval(expr2, conf); analog für or
- eval (expr1 θ expr2, conf) = eval(expr1, conf) θ eval(expr2, conf)
- eval (in(s), conf) = true falls s $\in$ states, false sonst

Die Wirkung einer Zuweisung Var1 := expr im Kontext val ist eine Kontextveränderung mit neuem Kontext val', so daß val'(Var1) = eval(expr, conf) und val'(Var) = val(Var) für alle anderen Variablen Var.

Definition:

Sei conf = (states, val) eine Konfiguration eines Statecharts (S,V,T), und sei t eine Transition mit Annotation [cond] / action.

t ***kann feuern***, wenn source(t) $\in$ states und eval(cond, conf) = true ist. Die Menge aller Transitionen, die in Konfiguration conf feuern können, wird mit fire(conf) bezeichnet.

Sei a = lca(source(t), target(t)) der kleinste gemeinsame Vorfahre von source(t) und target(t) im Zustandsbaum von S, und seien src(t) und tgt(t) diejenigen Kinder von a, die in den Teilbäumen source(t) bzw. target(t) liegen. Die Mengen source*(t) und target*(t) der durch das Feuern von t ***verlassenen*** bzw. ***neu betretenen Zustände*** sind wie folgt induktiv definiert:

- src(t) ist in source*(t)
- falls s in source*(t), dann sind auch alle Kinder von s in source*(t)
- tgt(t) und target(t) sind in target*(t)
- falls s in target*(t) und vom Typ AND ist, dann sind auch alle Kinder von s in target*(t)
- falls s in target*(t) und vom Typ XOR ist und nicht Vater von target(t) ist, dann ist das Kind von s, das Ziel einer initialen Transition ist, auch in target*(t)

Eine ***Folgekonfiguration*** conf' = (states', val') von conf entsteht durch (nichtdeterministische) Auswahl einer Transition t aus fire(conf) wie folgt:

- states' = states – source*(t) $\cup$ target*(t)
- val' unterscheidet sich von val dadurch, daß alle Wirkungen der Zuweisungen in action berücksichtigt werden.

Die operationale Semantik eines Statecharts (S,V,T) ist die Menge seiner möglichen Ausführungen mit Konfigurationen $conf_0$, $conf_1$, $conf_2$, ..., wobei $conf_0$ die initiale Konfiguration ist und $conf_{i+1}$ eine Folgekonfiguration von $conf_i$ sein muß.

13.3.2 Abbildung von Statecharts auf endliche Automaten

Ein alternativer und komplementärer Ansatz, die Semantik von Statecharts formal zu definieren, ist die Abbildung auf endliche Automaten. Endliche Automaten sind ein Standardmodell der Informatik mit wohldefinierter Semantik und einer reichen Theorie; insbesondere im Hinblick auf Verifikationsfragen ist die Automatentheorie ein nützliches Instrument.

Definition:

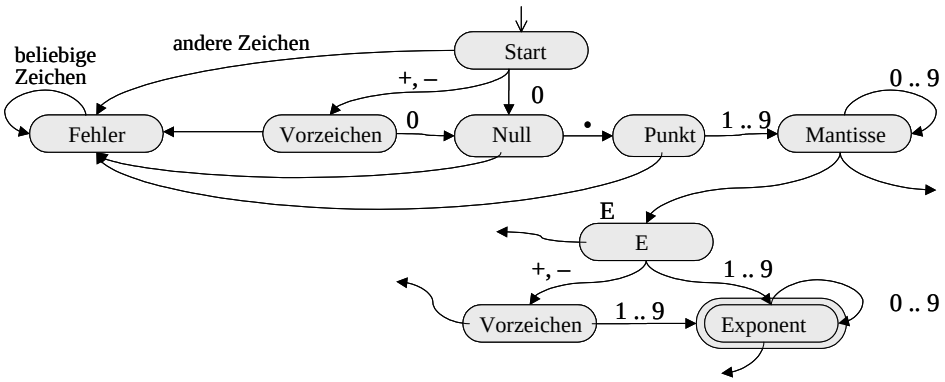
Ein endlicher Automat (finite state automaton) ist ein 5-Tupel M = (Z, Σ, δ, z0, E) mit

- einer endlichen Zustandsmenge Z
- einem Alphabet (d.h. einer endlichen Menge von Zeichen) Σ
- einer Transitionsfunktion δ: $Z \times \Sigma \rightarrow Z$
- einem Startzustand z0
- einer Menge von Endzuständen $E \subseteq Z$

Wir sagen, der Automat geht im Zustand $z \in Z$ mit dem Lesen des Eingabezeichens $x \in \Sigma$ in den Zustand $\delta(z,x) \in Z$ über. Die Transitionsfunktion δ wird von einzelnen Zeichen homomorph auf ganze Zeichenketten $w \in \Sigma^*$ zur Funktion δ^*: $Z \times \Sigma^* \rightarrow Z$ erweitert: $\delta^*(z, au) = \delta^*(\delta(z,a),u)$ mit $z \in Z$, $a \in \Sigma$, $u \in \Sigma^*$. Die Menge $L(M) = \{w \in \Sigma^* \mid \delta^*(z0,w) \in E\} \subseteq \Sigma^*$ ist die vom Automat M akzeptierte Sprache.

Endliche Automaten sind ein in der Informatik weit verbreitetes Modellierungs- und Spezifikationsinstrument.

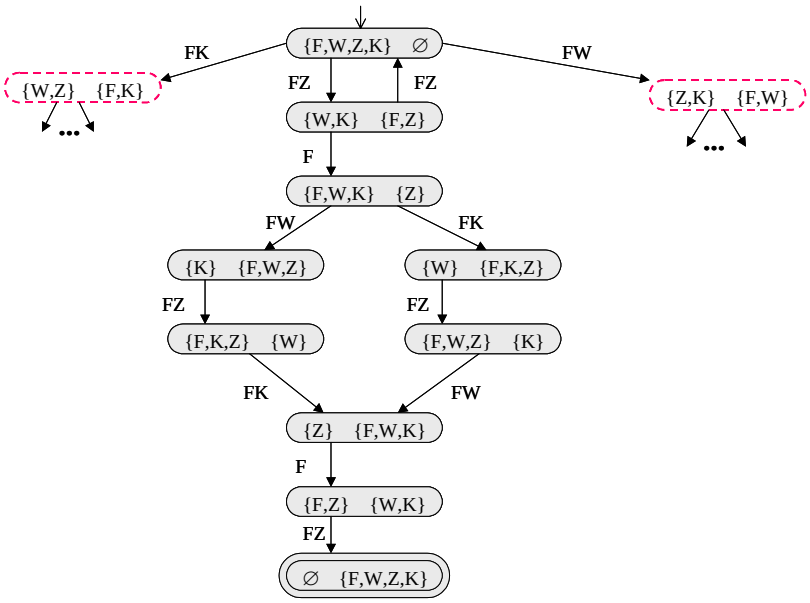
Beispiel 1:
Wir können die Menge der Gleitkommazahlen – in korrekter wissenschaftlicher Notation, also mit normalisierter Mantisse und Exponent – durch einen Automaten beschreiben. Beispielsweise sind 0.901E-42, -0.123E+1 und +0.909E10 korrekte Angaben, die der Automat akzeptieren soll, und 0.011E9, 15E-4, AB12XYZ sind inkorrekte Eingaben, für die der Automat in einen Fehlerzustand übergehen soll. Im folgenden Automatendiagramm ist der Anfangszustand durch einen kleinen Pfeil dargestellt und Endzustände sind doppelt umrandet. Zur Übersichtlichkeit sind nicht alle Übergänge in den Fehlerzustand angegeben.



Allgemeiner können beliebige sog. reguläre Sprachen durch endliche Automaten spezifiziert werden, und der einer solchen Sprache entsprechende Automat kann verwendet werden, um ein Eingabewort auf Zugehörigkeit zur Sprache zu testen. Dies hat Anwendungen u.a. bei der Erkennung von Mustern, wie man sie etwa im bekannten Unix-Tool grep angeben kann.

Beispiel 2:
Wir können die bekannte Knobelei Wolf-Ziege-Kohl durch einen endlichen Automaten modellieren. Bei dieser Knobelei muss ein Fährmann (F) Wolf (W), Ziege (Z) und Kohl (K) von einem Ufer zum anderen übersetzen, kann aber immer nur eines dieser drei „Objekte“ auf einmal mitnehmen. Erschwerend dabei ist, dass die Ziege den Kohl und der Wolf die Ziege fressen würde, wenn sie ohne Aufsicht gelassen werden. Die Zustände des Automaten modellieren jeweils Paare (L,R), wobei L angibt, welche „Objekte“ am linken Ufer sind und R, welche am rechten Ufer sind. Die Transitionen entsprechen also dem Übersetzen von einem Ufer ans andere; die Transitionsbeschriftungen geben an, welche zwei der insgesamt vier „Objekte“ jeweils übersetzen. Die Transitionsbeschriftungen haben hier aber nur mehr oder weniger Kommentarcharakter; es gibt es keine eigentlichen Eingabewörter, vielmehr kommt es nur auf die Zustandsfolge vom Startzustand zum Endzustand an (man könnte z.B. eine triviale Eingabe wählen und einfach alle Transitionen mit demselben Zeichen eines einelementigen Alphabets beschriften). Die Abbildung des Automaten lässt „uninteressante“, weil nicht zum Ziel führende, Zustände und Übergänge weg.

Allgemeiner spielen endliche Automaten in der Modellierung und Analyse kombinatorischer Spiele eine wichtige Rolle.



Bei der **Abbildung von Statecharts auf endliche Automaten** müssen drei Aspekte betrachtet werden:

1) die Abstraktion von Variablen mit unendlichen Wertebereichen (z.B. ganze Zahlen) durch einen endlichen Zustandsraum,

2) die Abbildung der aktuell aktiven Zustandsmenge eines Statecharts auf einen Zustand eines Automaten,

3) die Codierung der Übergangsannotationen (d.h. der ECA-Regeln) eines Statecharts in Zustände eines Automaten.

Zu Aspekt 1:

Die Aspekte 2 und 3 sind eher technischer Natur, während Aspekt 1 fundamentalen Charakter hat: offensichtlich können unendliche Strukturen nicht ohne weiteres auf endliche Strukturen ohne Informationsverlust abgebildet werden. Wir lösen dies, indem wir jede Bedingung, die sich auf Variablen mit unendlichem Wertebereich bezieht, durch eine Boolesche Variable ersetzen; die Variable ist auf True gesetzt, wenn die Originalbedingung wahr ist. Zuweisungen zu den Originalvariablen mit unendlichem Wertebereich werden in dieser Abstraktion durch Zuweisungen zu den Booleschen Variablen ersetzt; wenn dabei offen ist, ob die Boolesche Variable den Wert True oder False erhalten soll (z.B. weil die Originalbedingungen so komplex sind, dass sich dies gar nicht entscheiden lässt, oder weil der Wahrheitswert von Eingabeparametern abhängt), müssen beide Varianten berücksichtigt werden.

Im Reiseplanungsbeispiel von Abschnitt 11.1 etwa wird die Bedingung Cost > Budget durch eine Boolesche Variable BudgetOK ersetzt. Nach Zuweisungen wie Cost := ConfFee + TravelCost müssen beide möglichen Wahrheitswerte für BudgetOK berücksichtigt werden (d.h. wir kennen nach einer solchen Zuweisung den Wert von BudgetOK nicht).

Formal handelt es sich bei diesem Abbildungsschritt also um eine nichtinjektive Abbildung $\psi 1$: val $\rightarrow$ B1 $\times$ B2 $\times$ ... $\times$ Bm des Ausführungskontextes eines Statecharts auf eine endliche Menge Boolescher Variablen B1, ..., Bm.

Zu Aspekt 2:

In einem Statechart mit Zustandsmenge S können wegen der Hierarchisierung einerseits und vor allem wegen der orthogonalen Komponenten andererseits mehrere Zustände gleichzeitig aktiv sein; in einem endlichen Automaten ist zu einer Zeit jeweils nur ein Zustand betreten.

Die Abbildung des Statecharts auf endlichen Automaten mit Zustandsmenge Z muss also Teilmengen von S „codieren“.

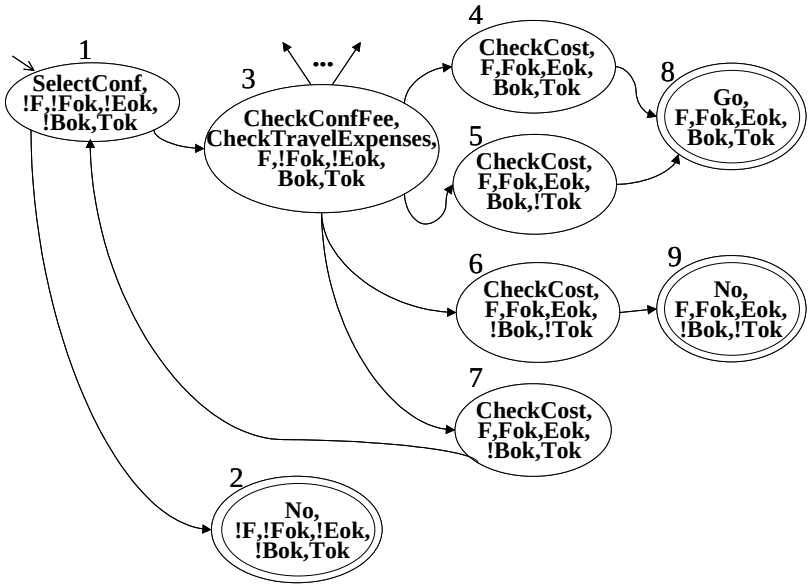
Im Reiseplanungsbeispiel von Abschnitt 11.1 etwa ist die Zustandsmenge {CheckConfFee, SelectTutorials, CheckTravelExpenses, CheckAirfare, CheckHotel} ein einziger Zustand des entsprechenden Automaten.

Formal entspricht diese Art der Zustandscodierung einer Abbildung $\psi 2$: states $\rightarrow 2^S =: Z$ der Ausführungszustände des Statecharts auf Zustände eines „Potenzmengenautomaten“. Wenn im Statechart Zustände s1, ..., sk aktiv sind, ist der entsprechende Automat gerade im Zustand $\{s1, ..., sk\} \in Z$. Startzustand des Automaten ist der Zustand $\{sj \mid sj \in conf_0.states\}$, wobei $conf_0$ die initiale Konfiguration des Statecharts ist. Man sieht sofort, dass der überwiegende Teil der Zustände in Z nie erreichbar ist und bei weiteren Analysen des Automaten weggelassen werden kann.

Zu Aspekt 3:

Nach der Abstraktion zu Aspekt 1 sind alle Bedingungen in den Transitionsannotationen eines Statecharts aussagenlogische Formeln über Booleschen Variablen B1, ..., Bm und die Aktionen sind Zuweisungen zu den Booleschen Variablen. Um dies auf einen einfachen Automaten abzubilden, wird die Wertebelegung der Booleschen Variablen in den Zustandsraum des Automaten hineincodiert. Formal entspricht dies einer injektiven Abbildung $\psi 3$: $Z \times B1 \times B2 \times ... \times Bm \rightarrow Z'$, wobei Z' den so erweiterten Zustandsraum des Automaten bezeichnet. Wir erzeugen dann im Automaten eine – unbeschriftete – Transition von Zustand (z, b1, ..., bm) nach (z', b1', ..., bm'), wenn es im Statechart eine Transition von dem z entsprechenden Ausführungszustand mit Transitionsannotation [cond] / action in den z' entsprechenden Ausführungszustand gibt, so dass cond bei Auswertung mit der Variablenbelegung b1, ..., bm wahr wird und action die Variablenbelegung b1', ..., bm' erzeugt.

Ein Ausschnitt des aus diesen drei Abbildungsschritten resultierenden Automaten für eine vereinfachte Variante des Reiseplanungsbeispiels von Abschnitt 11.1 sieht wie folgt aus. Gegenüber dem vollständigen Statechart werden dabei die Zustände CheckConfFee und CheckTravelExpenses nicht weiter verfeinert. Die Bedingung Cost > Budget wird zur Booleschen Variablen Bok abstrahiert und die Bedingung Trials < 3 zu Tok.



Insgesamt repräsentiert der aus einem Statechart abgeleitete Automat die Menge der möglichen Konfigurationen bei der Ausführung des Statecharts, so dass die Zustandsübergänge den möglichen Übergängen von einer Konfiguration zu einer Folgekonfiguration entsprechen.

13.4 Eigenschaften von Statecharts und deren Verifikation

Die im Hinblick auf verlässliche Informationsdienste kritischen Eigenschaften des Verhaltens von Statecharts und verwandten Prozessspezifikationen, die wir formal verifizieren sollten, können in zwei Kategorien eingeteilt werden:

Sicherheitseigenschaften (Safety Properties): logische Invarianten, die in jedem Zustand während der Prozessausführung gelten sollen. Im Reiseplanungsbeispiel von Abschnitt 11.1 beispielsweise wäre die Forderung, dass die Reisekosten das vorgesehene Budget nicht überschreiten dürfen, eine Sicherheitseigenschaft. Intuitiv kann man Sicherheitseigenschaften auch mit der Forderung gleichsetzen, dass bestimmte unerwünschte Konfigurationen (in denen die Sicherheitsinvariante verletzt wäre) während der Prozessausführung nicht betreten werden.

Lebendigkeitseigenschaften (Liveness Properties): logische Bedingungen, die bei einer Prozessausführung schlussendlich gelten sollen und damit den Fortschritt des Prozesses (Termination, Fairness, etc.) sicherstellen. Im Reiseplanungsbeispiel von Abschnitt 11.1 beispielsweise wäre die Forderung, dass der Reiseantrag schlussendlich entschieden wird, der Prozess also in einem der Zustände Go oder No terminiert, eine Lebendigkeitseigenschaft. Intuitiv kann man Lebendigkeitseigenschaften auch mit der Forderung gleichsetzen, dass bestimmte erwünschte Konfigurationen während der Prozessausführung irgendwann definitiv erreicht werden.

Um Eigenschaften dieser Art, also Bedingungen über das dynamische Verhalten eines Prozesses, formal ausdrücken zu können, wird eine erweiterte Art von Logik benötigt, die sog. **temporalen Logiken**. Der temporale Aspekt bezieht sich dabei in der Regel auf Ausführungsschritte eines Programms, Prozesses oder allgemeiner Transitionssystems, also eine abstrakte und diskrete Zeit. Temporallogiken zählen zu der Familie der sog. **Modallogiken**, bei denen der Wahrheitswert von logischen Aussagen relativ zu den Umständen (Modalitäten) dieser Aussagen verstanden wird. Beispielsweise können Aussagen wie „es regnet“ von Ort und Zeit der Aussage abhängen, und Aussagen wie „Gott existiert“ oder „das Weltall wird in seiner Expansion irgendwann stoppen und danach kollabieren“ hängen vom subjektiven Glauben oder dem relativen Wissensstand der jeweiligen Person ab. Ungeachtet der relativen Wahrheit solcher Aussagen kann man aber streng logische Schlüsse über die Konsequenzen der Aussagen führen. Für die Präzisierung von Programm- bzw. Prozessausführungseigenschaften haben sich vor allem die temporale Logik CTL (Computational Tree Logic) sowie deren Verallgemeinerungen (z.B. CTL*) bewährt.

11.4.1 Die Temporale Logik CTL

Die temporale Logik CTL (Computational Tree Logic) betrachtet mögliche Ausführungspfade eines Programms bzw. Prozesses, also Folgen von aufeinanderfolgenden Programm- bzw. Prozesszuständen (z.B. Konfigurationen einer Statechart-Ausführung). Das gedankliche Verfolgen aller möglichen Zustandsfolgen führt auf einen Baum von Zuständen, den Berechnungsbaum, der bei dem Namen CTL Pate gestanden hat. Diese Art von Temporallogik wird auch als „Branching Time Logic“ bezeichnet.

CTL umfasst **aussagenlogische Formeln**, die sich auf Zustände einer Programmausführung beziehen. Diese Formeln können quantifiziert werden – mit einem **Existenzquantor E** oder einem **Allquantor A** -, wobei sich die Quantifizierung auf Ausführungspfade bezieht: für einen Pfad oder für alle Pfade, die von einem bestimmten Zustand ausgehen, soll eine bestimmte Formel gültig sein. In den meisten Fällen ist der Zustand, von dem aus die mögli-

chen Ausführungspfade betrachtet werden, der initiale Zustand des gesamten Programms oder Prozesses.

Die beiden Quantoren E und A werden stets mit einem der folgenden **Modaloperatoren X (Next), G (Globally), F (Finally)** kombiniert. Dabei bedeutet:

EX P: eine Formel P soll in (mindestens) einem Nachfolgerzutand gültig sein.

AX P: eine Formel P soll in allen möglichen Nachfolgerzuständen gültig sein.

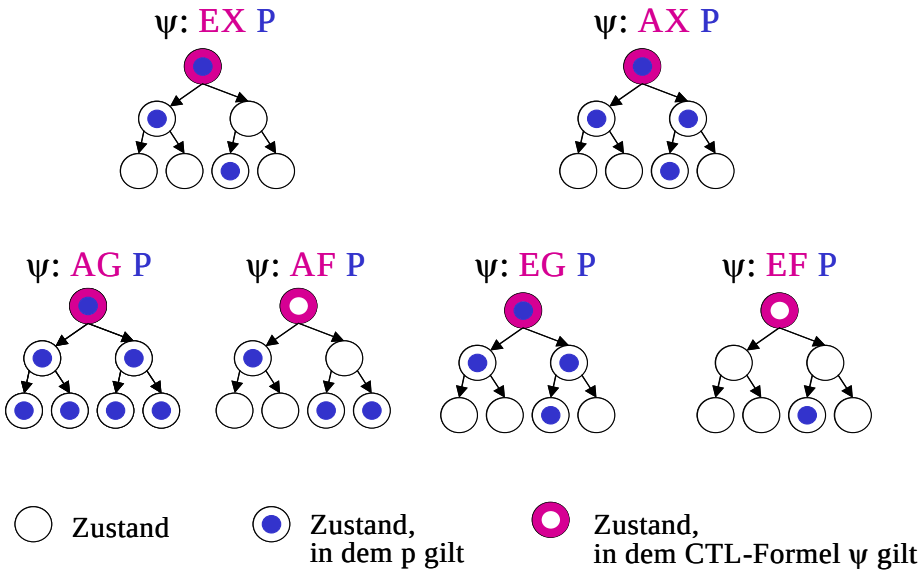
EG P: eine Formel P soll entlang (mindestens) eines Pfades global, d.h. in allen Zuständen des Pfades, gültig sein.

AG P: eine Formel P soll entlang aller möglichen Pfade global, d.h. in allen Zuständen der jeweiligen Pfade, gültig sein.

EF P: eine Formel P soll entlang (mindestens) eines Pfades irgendwann, d.h. in einem der Zustände des Pfades, gültig sein.

AF P: eine Formel P soll entlang aller möglichen Pfade irgendwann gültig sein.

Die Betrachtung der Wahrheitswerte aussagenlogischer Formeln in Zuständen und entlang von Pfaden eines Ausführungsbaums generell und die verschiedenen Kombinationen von Quantor und Modaloperator werden in der folgenden Abbildung visualisiert.



Definition:
Eine atomare CTL-Formel ist eine aussagenlogische Formel über elementaren Aussagen (bzw. Booleschen Variablen), inklusive Aussagen der Form „in(s)“, wobei s ein Zustand eines Transitionssystems ist.
Die Menge der in CTL erlaubten Formeln ist induktiv wie folgt definiert:

(i) Jede atomare CTL-Formel ist eine Formel.

(ii) Wenn P und Q Formeln sind, dann sind auch EX (P), AX (P), EG (P), AG (P), EF (P), AF (P), (P), $\neg$P, P$\wedge$Q, P$\vee$Q, P$\Rightarrow$Q und P$\Leftrightarrow$Q Formeln.

Formeln mit einer Quantor/Modalität-Kombination können also auch geschachtelt werden. Dabei dürfen jedoch Quantoren und Modaloperatoren immer nur in den genannten Kombinationen auftreten, niemals alleine. Diese Einschränkung fällt in der expressiveren Temporallogik CTL* weg, dafür hat diese aber eine höhere Komplexität.

Beispiele:

1) Die Sicherheitseigenschaft für das Reiseplanungsbeispiel, dass das Budget nicht überschritten werden darf, kann wie folgt in CTL formalisiert werden:

   AG ($\neg$Bok $\Rightarrow$ $\neg$in(Go))

   oder alternativ (äquivalent)

   $\neg$EF ($\neg$Bok $\wedge$ in(Go))
2) Die Lebendigkeitseigenschaft für das Reiseplanungsbeispiel, dass der Prozess in einem der Zustände Go oder No terminieren muss, kann in CTL folgendermaßen formalisiert werden:

   AF (in(Go) $\vee$ in(No))
3) Die Eigenschaft, dass wir nach einem ersten wegen Budgetüberschreitung gescheiterten Vorschlag immer noch die Reise genehmigen können, ist in CTL wie folgt ausdrückbar:

   EF ((in(CheckCost) and !Bok) => (EF (in(Go))))

Definition:
Gegeben sei eine Menge P atomarer aussagenlogischer Formeln. Eine *Kripke-Struktur* M über P ist ein 4-Tupel (S, s0, R, L) mit

- einer endlichen Zustandsmenge S,
- einem Startzustand s0 $\in$ S,
- einer binären Transitionsrelation R $\subseteq$ S $\times$ S,
- einer Funktion L: S $\rightarrow$ 2^P, die jedem Zustand die Menge der in dem Zustand wahren atomaren Aussagen zuordnet.

Die *Interpretation* ψ einer CTL-Formel F mit atomaren Aussagen P ist eine Abbildung auf eine Kripke-Struktur M=(S, s0, R, L) über atomaren Aussagen P, so dass die Wahrheitswerte von Teilformeln p bzw. p1, p2 von F in den Zuständen s von M, in Zeichen: M,s |= p, wie folgt festgelegt sind:

(i) M,s |= p mit einer aussagenlogischen Formel p gilt genau dann, wenn p $\in$ L(s);
(ii) M,s |= $\neg$p genau dann, wenn nicht M,s |= p gilt;
(iii) M,s |= p1 $\wedge$ p2 genau dann, wenn M,s |= p1 und M,s |= p2;
(iv) M,s |= p1 $\vee$ p2 genau dann, wenn M,s |= p1 oder M,s |= p2;
(v) M,s |= EX p genau dann, wenn es t$\in$S gibt mit (s,t)$\in$R und M,t |= p;
(vi) M,s |= AX p genau dann, wenn für alle t$\in$S mit (s,t)$\in$R gilt: M,t |= p;
(vii) M,s |= EG p genau dann, wenn es t1, ..., tk $\in$S gibt mit t1=s, (ti, ti+1)$\in$R für alle i und tk=tj für ein j:1$\leq$j<k oder tk ohne Nachfolger bzgl. R, so dass M,ti |= p für alle i;
(viii) M,s |= AG p genau dann, wenn für alle t$\in$S mit (s,t)$\in$R* gilt: M,t |= p;
(ix) M,s |= EF p genau dann, wenn es t$\in$S gibt mit (s,t)$\in$R* und M,t |= p;
(x) M,s |= AF p genau dann, wenn es für alle t$\in$S mit (s,t)$\in$R* einen Zustand t'$\in$S gibt mit a) (t,t')$\in$R* oder b) (s,t')$\in$R* und (t',t)$\in$R*, so dass M,t' |= p gilt.

Eine Kripke-Struktur M = (S, s0, R, L) ist ein *Modell* einer CTL-Formel F, wenn M,s0 |= F. Eine Formel heißt erfüllbar, wenn sie mindestens ein Modell hat, ansonsten unerfüllbar. Eine Formel F heißt allgemeingültig (oder Tautologie), wenn jede Kripke-Struktur über den atomaren Aussagen von F ein Modell von F ist.

13.4.2 Verifikation durch Modellprüfen (Model Checking)

Deduktionsverfahren für CTL haben sehr hohen Rechenaufwand. Insbesondere das Testen einer CTL-Formel auf Allgemeingültigkeit, also das Theorembeweisen in einem CTL-Kalkül, ist für die meisten praktischen Belange zu aufwändig (mindestens exponentiell in der Formelgröße). Dieses Vorgehen ist jedoch zum Verifizieren oder Falsifizieren kritischer Eigenschaften von Transitionssystemen weder angemessen noch notwendig. Vielmehr genügt es zu testen, ob das vorliegende Transitionssystem ein Modell einer gegebenen CTL-Formel ist. Diese Technik nennt man Modellprüfen oder Model-Checking. Sie hat eine Laufzeit, die polynomiell in der Größe der Formel und des Transitionssystems ist.

Die Vorgehensweise beim Modellprüfen orientiert sich eng an der Definition der Interpretation einer CTL-Formel F, indem man prüft, ob das vorliegende Transitionssystem eine Kripke-Struktur ist, in der die Formel wahr ist. Der Algorithmus des Modellprüfens arbeitet induktiv über dem Aufbau der Formel F, indem er bottom-up für jede Teilformel q von F die Menge der Zustände des Transitionssystems mit markiert, in denen q gilt. Wenn dann am Ende der initiale Zustand des Transitionssystems mit F markiert ist, wissen wir, dass das Transitionssystem ein Modell von F ist.

Die Prozeduren für das Markieren der Zustände $Q \subseteq S$, in denen eine Teilformel q gilt, hängen von der Form von q ab und sehen wie folgt aus. Dabei seien p, p1, p2 Teilformeln von q, für die gemäß der strukturellen Induktion des Gesamtalgorithmus bereits markierte Zustandsmengen P, P1, P2 bestimmt wurden.

(i) q ist eine atomare Aussage (Boolesche Variable):
Markiere alle Zustände s mit $q \in L(s)$ mit q

(ii) q hat die Form $\neg p$:
Markiere S – P mit q

(iii) q hat die Form $p1 \wedge p2$:
Markiere $P1 \cap P2$ mit q

(iv) q hat die Form $p1 \vee p2$:
Markiere $P1 \cup P2$ mit q

(v) q hat die Form EX p:
Markiere alle Vorgänger von P mit q, also alle $s \in S$, für die es ein $x \in P$ gibt mit R(s,x)

(vi) q hat die Form AX p:
Markiere s mit q, wenn alle Nachfolger von s mit p markiert sind

(vii) q hat die Form EF p:
Dieser Fall wird auf die Rekursions- bzw. Fixpunktgleichung $Q = P \cup pred(Q)$ zurückgeführt. Dabei ist pred(Q) die Menge der Vorgänger der Zustände in Q.
Diese Fixpunktgleichung beruht auf der logischen Äquivalenz $EF\ p \Leftrightarrow p \vee EX\ (EF\ p)$.
Die Lösung der Fixpunktgleichung, d.h. die Berechnung des kleinsten Fixpunktes, erfolgt mit der Prozedur:

```
Q := P;
Qnew := Q ∪ pred(Q);
while not (Q = Qnew) do
   Q := Qnew;
   Qnew := Q ∪ pred(Q);
od;
```

(viii) q hat die Form EG p:

Die Äquivalenz EG p $\Leftrightarrow$ p $\wedge$ EX (EG p) weist den Weg zur iterativen Berechnung:

```
Q := P;
Qnew := Q ;
repeat
      for each s in Q do
          if s has successors and no successor of s is in Q
          then Qnew := Q - {s}; fi;
      od;
until (Q = Qnew);
```

(ix) q hat die Form AG p:

Die Äquivalenz AG p $\Leftrightarrow$ p $\wedge$ AX (AG p) weist den Weg zur iterativen Berechnung:

```
Q := P;
repeat
     Qnew := Q;
     for each s in Q do
          if s has successors and one successor of s is not in Q
          then Q := Q - {s} fi;
     od;
until (Q = Qnew);
```

Alternativ kann dieser Fall auf die Äquivalenz AG p $\Leftrightarrow \neg$ EF ($\neg$p) zurückgeführt werden.
Wir berechnen also zunächst die Zustandsmenge Q' zur Formel EF ($\neg$p) und markieren dann die Zustandsmenge S – Q' mit q.

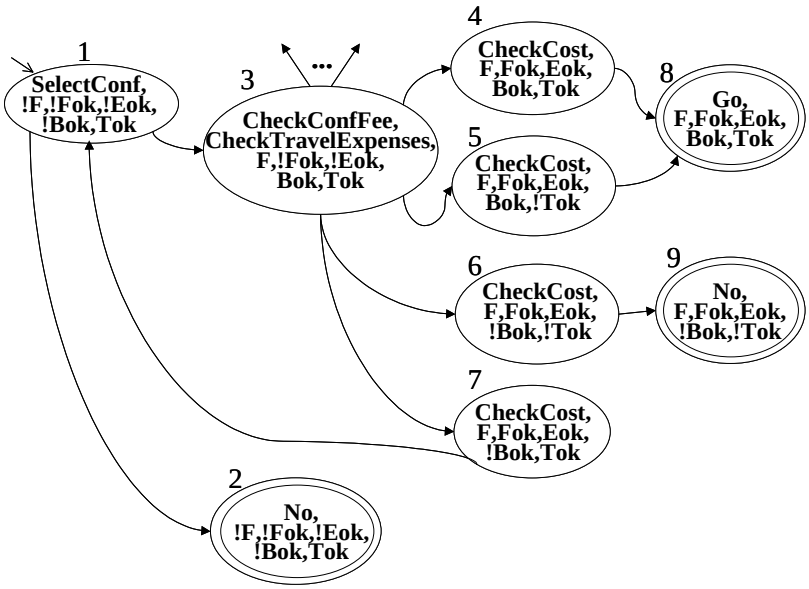
(x) q hat die Form AF p:

Die Äquivalenz AF p $\Leftrightarrow$ p $\vee$ AX (AF p) weist den Weg zur iterativen Berechnung:

```
Q := P;
repeat
     Qnew := Q;
     for each s in pred(Q) do
          if all successors of s are in Q
          then Q := Q ∪ {s}; fi;
     od;
until (Q = Qnew);
```

Alternativ kann dieser Fall auf die Äquivalenz AF p $\Leftrightarrow \neg$ EG ($\neg$p) zurückgeführt werden.
Wir berechnen also zunächst die Zustandsmenge Q' zur Formel EG ($\neg$p) und markieren dann die Zustandsmenge S – Q' mit q.

Beispiel:
Wir wollen Modellprüfen auf das Reiseplanungsbeispiel bzw. den daraus abgeleiteten endlichen Automaten anwenden. Der Automat ist hier – mit durchnummerierten Zuständen – nochmals abgebildet.



Zur Verifikation der Eigenschaft AG ($\neg$Bok $\Rightarrow$ $\neg$in(Go)) wird wie folgt markiert:
mit Bok die Zustände 3, 4, 5, 8
mit in(Go) der Zustand 8
mit $\neg$in(Go) die Zustände 1, 2, 3, 4, 5, 6, 7, 9
mit ($\neg$Bok $\Rightarrow$ $\neg$in(Go)) die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
mit AG ($\neg$Bok $\Rightarrow$ $\neg$in(Go))
in der 0. Iteration die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
in der 1. Iteration die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
Damit gilt die Eigenschaft im Startzustand 1.

Zur Verifikation der Eigenschaft AF (in(Go) $\vee$ in(No)) wird wie folgt markiert:
mit in(Go) der Zustand 8
mit in(No) der Zustand 2, 9
mit in(Go) $\vee$ in(No) die Zustände 2, 8, 9
mit AF (in(Go) $\vee$ in(No))
in der 0. Iteration die Zustände 2, 8, 9
in der 1. Iteration die Zustände 2, 4, 5, 6, 8, 9
in der 2. Iteration die Zustände 2, 4, 5, 6, 8, 9
Damit kann die Eigenschaft für den Startzustand 1 nicht nachgewiesen werden.

Zur Verifikation der Eigenschaft EF ((in(CheckCost) and !Bok) => (EF (in(Go)))) wird wie folgt markiert:
mit in(Go): der Zustand 8
mit EF (in(Go)): die Zustände 1, 3, 4, 5, 6, 7, 8, 9
mit not in(CheckCost) or Bok: die Zustände 1, 2, 3, 4, 5, 8, 9
mit (in(CheckCost) and !Bok) => (EF (in(Go)): die Zustände 1, 2, 3, 4, 5, 6, 7, 8, 9
mit EF ((in(CheckCost) and !Bok) => (EF (in(Go))))
in der 0. Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9
in der 1. Iteration: 1, 2, 3, 4, 5, 6, 7, 8, 9
Damit gilt die Eigenschaft im Startzustand 1.

Diese grundsätzliche algorithmische Vorgehensweise kann in ihrer Effizienz noch deutlich verbessert werden, wenn das Modellprüfen statt auf dem eigentlichen Transitionssystem auf einer geeigneten symbolischen Repräsentation des Systems arbeitet; man spricht dann auch von symbolischem Modellprüfen. Die erfolgreichste Variante dieser Richtung, mit der sehr große Systeme verifiziert werden konnten, codiert die Zustände des Transitionssystems in Form von m binären Variablen, also praktisch als m-stellige Bitstrings, und die Transitionen als 2m-stellige Boolesche Funktion, deren Funktionswert 1 ist, wenn es eine mögliche Transition zwischen den entsprechenden Zuständen gibt, und 0 sonst. Diese Boolesche Funktion wiederum kann mit sog. BDDs (Binary Decision Diagrams) bzw. OBDDs (Ordered Binary Decision Diagrams) sehr kompakt repräsentiert werden, und symbolisches Modellprüfen arbeitet dann als Markierungsverfahren auf OBDDs.

Ergänzende Literatur zu Kapitel 13:

D. Harel, M. Politi, Modeling Reactive Systems with Statecharts: the Statemate Approach, McGraw-Hill, 1998

D. Harel, E. Gery, Executable Object Modeling with Statecharts, IEEE Computer Vol.30 No.7, 1997

E.A. Emerson, Temporal and Modal Logic, in: J. van Leeuwen (Editor), Handbook of Theoretical Computer Science, Elsevier, 1990

E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, MIT Press, 1999

G. Weikum, D. Wodtke, A. Kotz-Dittrich, P. Muth, J. Weißenfels, Spezifikation, Verifikation und verteilte Ausführung von Workflows in MENTOR, in: T. Härder (Hrsg.), Informatik Forschung und Entwicklung Band 12 Heft 2, Springer-Verlag, 1997, Themenheft über Workflow-Management

G. Weikum, Towards Guaranteed Quality and Dependability of Information Services, 8. GI-Fachtagung über Datenbanksysteme in Büro, Technik und Wissenschaft, Freiburg, 1999, Springer-Verlag, 1999.

Scheer, A.-W., ARIS - Business Process Modeling, 2nd Edition, Springer-Verlag, 1999

Object Management Group (OMG), Unified Modeling Language (UML) Documentation, http://www.omg.org/ oder http://www.rational.com/uml/

F. Leymann, D. Roller, Production Workflow – Concepts and Techniques, Prentice Hall, 2000

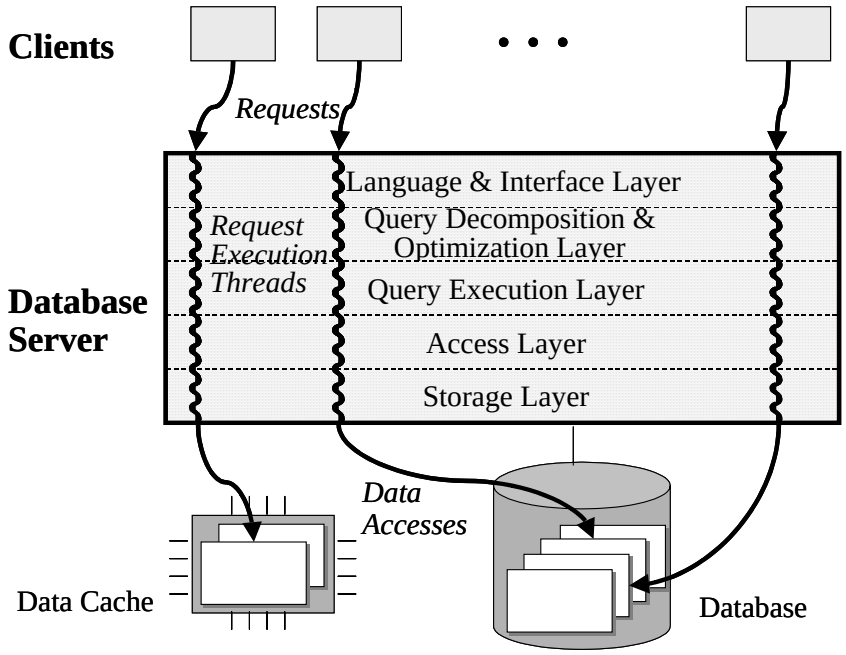
A. Dogac, L. Kalinichenko, T. Özsu, A. Sheth (Editors), Workflow Management Systems and Interoperability, Springer, 1998

G. Vossen, J. Becker (Hrsg.), Geschäftsprozeßmodelle und Workflow-Management - Modelle, Methoden, Werkzeuge, International Thomson Publishing, 1995

S. Jablonski, M. Böhm, W. Schulze (Hrsg.), Workflow-Management - Entwicklung von Anwendungen und Systemen, dpunkt-Verlag, 1997

Kapitel 14: Datenspeicherung, Indexstrukturen und Anfrageauswertung

Praktisch alle Datenbanksysteme haben - zumindest in idealisierter Betrachtung - eine interne Schichtenarchitektur. Jede Schicht bietet an ihrer Schnittstelle bestimmte Methoden und Dienste an, mit deren Hilfe die Schnittstelle der nächsthöheren Schicht implementiert ist. Alle hier gemeinten Schichten liegen im selben Server; für Client-Programme ist nur die Schnittstelle der obersten Schicht sichtbar. Wenn ein Client-Auftrag (z.B. ein SQL-Kommando eines ESQL-Programms oder eines Web-Servlets) beim Server eintrifft, erzeugt oder verwendet er einen eigenen Thread, innerhalb dessen der Auftrag von Schicht zu Schicht in mehrere, primitivere Operationen zerlegt wird, bis er schließlich eine Reihe von Plattenzugriffen oder Seitenzugriffen auf einem hauptspeicherresidenten Daten-Cache erzeugt. Alle Threads zusammen bilden in der Regel einen gemeinsamen virtuellen Adressraum, in dem ein oder mehrere Betriebssystemprozesse auf einer CPU oder einem Parallelrechner (typischerweise einem SMP = Symmetric Multiprocessor) ablaufen.



Auf der ***Sprach- und Schnittstellenebene (language and interface layer)*** werden APIs für den Aufruf von Datenbankoperationen wie SQL-Kommandos bereitgestellt, also z.B. JDBC, ODBC oder auch produktspezifische Schnittstellen wie OCI (siehe auch Kapitel 6). Diese Kommandos werden zerlegt in interne Operatorbäume, die auf der nächsttieferen Ebene, der ***Anfragezerlegungs- und -optimierungsschicht (query decomposition and optimization layer)*** behandelt werden. Diese Operatorbäume entsprechen in erster Näherung relationenalgebraischen Ausdrücken, die aufgrund von Äquivalenzregeln sowie Kostenschätzungen für verschiedene Ausführungsplanvarianten optimiert werden. Diese Optimierung erfolgt möglichst zur Compile-Zeit der Anwendung, so daß einmal gene-

rierte Ausführungspläne bei wiederholter Ausführung der SQL-Anweisung bzw. des gesamten Anwendungsprogramms nicht jedes Mal erneut optimiert werden müssen.

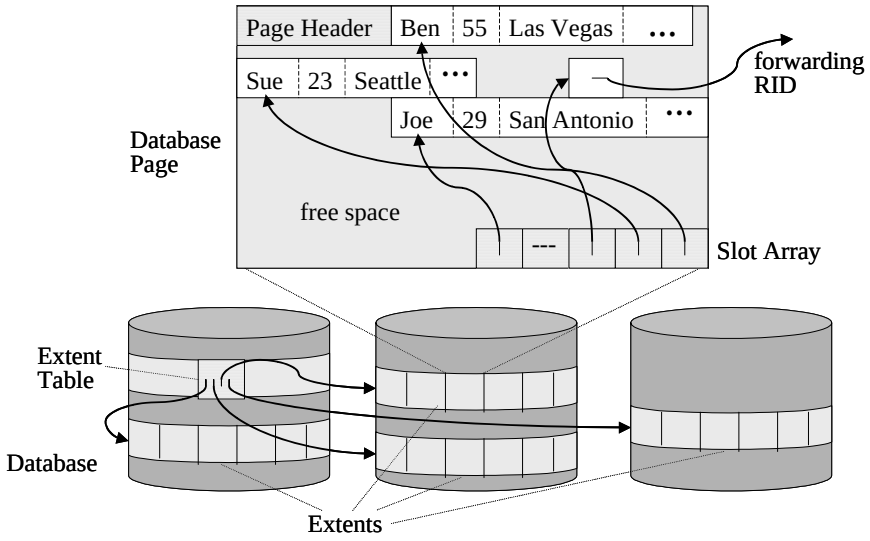
Jeder Operator in einem als Operatorbaum vorliegenden, optimierten Ausführungsplan wird auf einen bestimmtes Servermodul mit entsprechenden Algorithmen (z.B. zur Berechnung eines Equijoins) abgebildet. Diese Algorithmen und die Koordination des Kontroll- und Datenflusses zwischen den Operatoren wird von der ***Anfrageausführungsschicht (query execution layer)*** übernommen. Die einzelnen Operatoren wiederum erzeugen während ihrer Ausführung Record-Zugriffe sowie - zur Beschleunigung der Anfrage - Zugriffe auf sogenannte Indexstrukturen. Records und Indexstrukturen, letztere typischerweise als magnetplattenresidente Suchbäume implementiert, werden von der ***Zugriffsschicht (access layer)*** verwaltet. Alle Zugriffs- und Speicherungsstrukturen sind schlußendlich in Seiten (Blöcken) fester Größe (z.B. 32 KBytes) abgelegt, die die Transporteinheiten zwischen Magnetplatten und Hauptspeicher, genauer dem Daten-Cache, bilden. Im Daten-Cache werden jeweils die in jüngster Vergangenheit am häufigsten benutzten bzw. wichtigsten Seiten zum schnelleren Zugriff gepuffert; ein sogenannter Seitenersetzungsalgorithmus wie z.B. LRU (least recently used) steuert dynamisch den aktuellen Cache-Inhalt. Für Caching und Plattenzugriffe ist die ***Speicherungsschicht (storage layer)*** zuständig.

14.1 Wie Daten gespeichert werden

Alle Daten werden intern in Form von Records (Datensätzen) gespeichert, die aus einer Reihe von Feldern (Fields) bestehen und zusammen einen - in der Regel physisch zusammenhängenden - Bytestring variabler Länge bilden. Records werden in Seiten (Blöcken) fester Länge (z.B. 32 KBytes) abgelegt, die als Transportcontainer zwischen Magnetplatten und Daten-Cache im Hauptspeicher dienen. Seiten sind innerhalb eines Tablespaces, der als plattenresidenter virtueller Adressraum für eine oder mehrere Tabellen einer (relationalen) Datenbank dient, fortlaufend nummeriert. Physisch besteht ein Tablespace aus vorab allozierten Blöcken auf einer oder mehreren Magnetplatten, wobei typischerweise eine kleine Anzahl von größeren, physisch zusammenhängenden Bereichen, sog. Extents, verwendet wird. Mittels einer Extent-Tabelle kann die "logische" Seitennummer in eine "phyische" Plattenblockadresse umgerechnet werden.

Jede Seite hat einen Seiten-Header fester Größe mit Freispeicherverwaltungsinformationen u.ä. sowie einen variabel langen Seiten-Trailer, der ein - häufig "Slot-Array" - genanntes Pointer-Array mit den Byte-Offsets der in der Seite abgelegten Records enthält. Die Records selbst liegen zwischen Header und Trailer an beliebigen Offsets; wegen dynamischer Erweiterungen und Verschiebungen von Records können zwischen Records auch Lücken mit freiem Speicher liegen. ein Record wird adressiert durch eine sogenannte Row-ID bzw. Record-ID, kurz RID (seltener auch TID = Tupel-ID genannt), die aus der Tablespace-Nummer, der Seitennummer und der Slot-Nummer im Seiten-Trailer besteht. Durch die Indirektion über das Slot-Array können Records innerhalb einer Seite beliebig verschoben werden, ohne daß sich ihre - in Indexstrukturen als Zeiger verwendeten - Adressen ändern. Gelegentlich kann es vorkommen, daß ein Record aus Platzmangel auf eine andere Seite verlagert werden muß; in diesem Fall wird auf der ursprünglichen Seite ein kurzer "Forwarding-Eintrag" angelegt, so daß die RID auch dann unverändert bleibt. Bei gelegentlichen Reorganisationen, die vom Datenbankadministrator während ruhigerer Betriebszeiten gestartet werden können, werden die Forwarding-Einträge eliminiert und Zeiger in Indexstrukturen angepasst.

Ein Beispiel für die beschriebenen Speicherungsstrukturen - mit Personen-Records, die Felder wie Name (name), Alter (age), Stadt (city), etc. haben - zeigt die folgende Abbildung.



14.2 Wie auf Daten effizient zugegriffen wird (Indexstrukturen)

Zur Beschleunigung von simplen Selektionen, als Operatoren innerhalb eines Ausführungsplans, können vom Datenbankadministrator mit dem SQL-Kommando

CREATE [UNIQUE] INDEX index-name ON table (column, {, column ...})

Indexstrukturen angelegt werden. Ein Index auf Attributen A1, A2, ..., Ak erlaubt die effiziente Bestimmung aller Treffer für

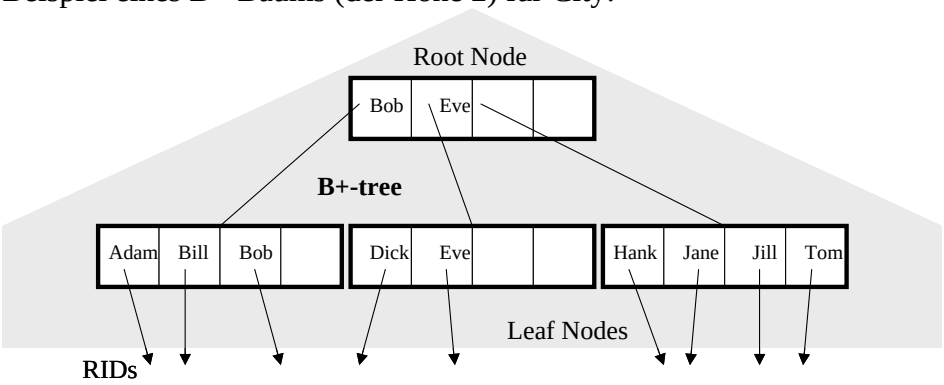
- Exact-Match-Selektionen der Form A1=wert1 $\wedge$ A2=wert2 $\wedge$ ... $\wedge$ Ak=wertk
  (z.B. City = 'Miami' (k=1) oder City = 'Paris' $\wedge$ State = 'Texas' (k=2)),
- Bereichs-Selektionen (Range Scans) der Form u1 $\leq$ A1 $\leq$ o1 $\wedge$ ... $\wedge$ uk $\leq$ Ak $\leq$ ok
  (z.B. 21 $\leq$ Age $\leq$ 30 (k=1) oder 21 $\leq$ Age $\leq$ 30 $\wedge$ Salaray $\geq$ 100 000 (k=2)) sowie
- Präfix-Match-Selektionen (mit einem Präfix der im Index verwalteten Felder)
  der Form u1 $\leq$ A1 $\leq$ o1 $\wedge$ u2 $\leq$ A2 $\leq$ o2 $\wedge$ ... $\wedge$ uj $\leq$ Aj $\leq$ oj (mit j < k)
  (z.B. 21 $\leq$ Age $\leq$ 30 bei einem Index über Age, Salary).

Indexstrukturen beschleunigen Anfragen, sofern die Anfrageoptimierung den Index entsprechend nutzt, belegen aber zusätzlichen Speicherplatz und bedeuten vor allem Mehraufwand für Änderungsoperationen. Wegen dieses Zielkonflikts obliegt es dem Datenbankadministrator, Indexstrukturen auf sinnvollen Feldkombinationen anzulegen; nur auf Primär- und Fremdschlüssel legen Datenbanksystem automatisch Indexstrukturen an. Bei dieser schwierigen Tuning-Aufgabe wird der Datenbankadministrator bei besseren Produkten von Optimierungswerkzeugen (z.B. sog. Index Wizards) unterstützt. Die Indexauswahl - als Teil der umfassenderen Aufgabe des sog. physischen Datenbankentwurfs - bleibt dennoch eine diffizile Fragestellung.

Ohne Index müssen Selektionen der o.a. Arten durch sequentiellen Scan ausgewertet werden, hätten also einen Berechnungsaufwand - gemessen in der Anzahl zu lesender Seiten der Datenbank -, der linear in der zugrundeliegenden Relationsgröße ist (in Kurznotation: O(n/C) mit n = Anzahl Records der Relation, C = Records pro Seite). Mit einem geeigneten Index verringert sich dieser Aufwand auf eine logarithmische Zeit in der Relationsgröße (in Kurnotation: $O(\log_k n/C)$ mit einer Basis k in der Größenordnung von einigen Hundert, s.u.). Um dies zu erreichen, sind Indexstrukturen intern als plattenresidente Suchbäume, genauer sog. B*-Bäume, implementiert.

Ein B*-Baum ist ein balancierter, hohler Mehrwegebaum für effizientes Suchen, Einfügen und Löschen von (Such-)Schlüsseln (und den damit verbundenen Records) auf seitenorientiertem Sekundärspeicher. Er beinhaltet auf Blattniveau Schlüssel-RID-Paare bzw. Schlüssel mit einer RID-Liste, die aus den RIDs derjenigen Records besteht, die den Schlüssel enthalten. Nichtblattknoten enthalten - im Schnitt - k geordnete Schlüssel und k+1 Sohnzeiger, wobei die Schlüssel in den Nichtblattknoten den Charakter von Wegweisern haben, um von der Wurzel ausgehend das Blatt zu finden, das den gesuchten Schlüssel enthält (oder enthalten müßte, wenn der Schlüssel überhaupt vorkäme). Der Baum ist jederzeit perfekt balanciert; alle Blätter haben also dieselbe Distanz zur Wurzel, nämlich $\text{ceil}(\log_k n/C) + 1$, woraus sich die logarithmische Suchzeit ergibt. k wird als der (mittlere) Fanout (Verzweigungsgrad) der Nichtblattknoten bezeichnet.

Beispiel eines B*-Baums (der Höhe 2) für City:



Formale Definition:
Ein Mehrwegebaum heißt B*-Baum der Ordnung (m,m*), $m \geq 1$, $m^* \geq 1$, wenn gilt:

- Jeder Nichtblattknoten außer der Wurzel enthält mindestens m und höchstens 2m Schlüssel.
- Ein Nichtblattknoten mit k Schlüssel x1, ..., xk hat genau k+1 Söhne t1, ..., t(k+1), so daß
  - für alle Schlüssel s im Teilbaum ti, $2 \leq i \leq k$, gilt $x(i-1) < s \leq xi$, und
  - für alle Schlüssel s im Teilbaum t1 gilt $s \leq x1$, und
  - für alle Schlüssel s im Teilbaum t(k+1) gilt $xk < s$.
- Alle Blätter haben dasselbe Niveau (d.h. Distanz von der Wurzel).
- Jedes Blatt enthält mindestens m* und höchstens 2m* Schlüssel
  bzw. Schlüssel-RID(-Listen)-Paare.

Die entscheidende Invariante eines B*-Baums besteht also darin, daß durch die Wegweiser eines Knotens der Suchbereich in disjunkte Teilbereiche partitioniert wird. Dies ermöglicht einen Divide-

and-Conquer-Ansatz für das effiziente Suchen. Die Suche nach einem Schlüssel s beginnt bei der Wurzel und arbeitet sich top-down bis zu einem Blatt herunter. Auf jeder Baumstufe wird ein Knoten nach dem kleinsten Schlüssel xi durchsucht, für den $s \leq xi$ gilt; dies erfolgt mittels binärer Suche auf den innerhalb eines Knotens sortiert abgelegten Wegweisern. Anschließend wird die Suche im Teilbaum ti fortgesetzt (bzw. t(k+1) falls es kein xi gibt mit $s \leq xi$), bis schlußendlich ein Blatt erreicht ist. Dort endet die Suche entweder erfolgreich - und liefert dann die zum gesuchten Schlüssel gehörenden RIDs zurück - oder erfolglos - mit einem Returncode, der anzeigt, daß der gesuchte Schlüssel überhaupt nicht vorhanden ist. Im o.a. B*-Baum etwa würde die Suche nach 'Dick' dem Wegweiser links vom Eintrag 'Eve' in der Wurzel folgen und dann im darunter hängenden Blatt erfolgreich enden. Bereichsanfragen suchen den kleinsten Schlüssel des spezifizierten Bereichs, also den "linken Rand", und traversieren dann alle Blätter, bis ein Schlüssel erreicht wird, der größer ist als der "rechte Rand" des Suchbereichs. Zum Zwecke dieser Traveriserung sind benachbarte Blätter durch Zeiger verkettet.

Pseudocode für Suchen von Schlüssel s in B*-Baum mit Wurzel t:

```
t habe k Schlüssel x1, ..., xk und k+1 Söhne t1, ..., t(k+1)
(letzteres sofern t kein Blatt ist)
Bestimme den kleinsten Schlüssel xi, so daß s ≤ xi
if s = xi (für ein i ≤ k) und t ist ein Blatt
then Schlüssel gefunden
else
    if t ist kein Blatt then
        if s ≤ xi (für ein i ≤ k)
        then suche s im Teilbaum ti
        else suche s im Teilbaum t(k+1) fi
    else Schlüssel s ist nicht vorhanden fi
 fi
```

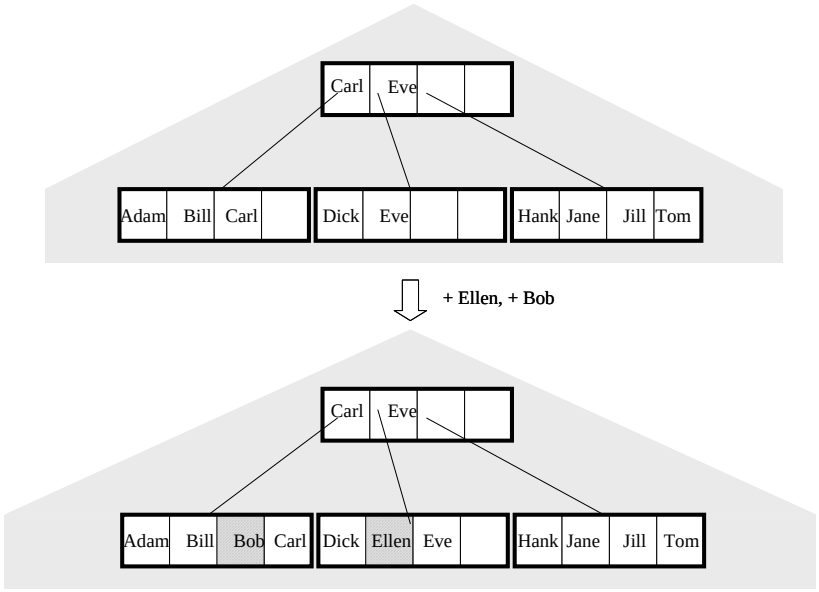
Statt einer festen Maximalanzahl 2m von Wegweisern in einem Knoten wird wegen der variablen Länge der Schlüssel eine variable Anzahl von Wegweisern gespeichert. Da die Einträge in den Nichtblattknoten nur der richtigen Verzweigung beim Suchen dienen, müssen diese nicht die vollständigen Schlüssel wiedergeben, sondern können auf geeignete Präfixe reduziert werden. Zum Beispiel könnte man im o.a. B*-Baum den Wegweiser 'Eve' in der Wurzel durch den kürzeren Separator 'F' ersetzen, ohne die Bauminvariante zu verletzen. Dadurch kann der Fanout des B*-Baums noch weiter erhöht werden, wodurch die Baumtiefe sinkt. Typische Fanouts in der Praxis liegen bei 100 bis 1000, so daß auch auf sehr großen Datenbeständen B*-Bäume selten höher als 3 sind. Dabei sind - bei sinnvoller Systemkonfiguration - alle Baumknoten außer den Blättern so gut wie cache-resident, so daß jede Exact-Match-Selektion mittels Index mit einem einzigen Plattenzugriff ausgeführt werden kann.

Beim Einfügen eines neuen Schlüssels wird zunächst das Blatt gesucht, in dem der Schlüssel abgelegt sein müßte, wenn er schon vorher vorhanden gewesen wäre. Wenn in diesem Blatt ausreichend freier Platz ist, wird der neue Schlüssel dort gespeichert. Wenn kein Platz ist, erfolgt ein sog. Knoten-Split. Dazu wird eine neue Seite von der Freispeicherverwaltung angefordert, die zum rechten Nachbarn des Blattes wird. Der Inhalt des "übergelaufenen" Blatts wird gleichmäßig zwischen altem und neuem Blatt aufgeteilt. Anschließend wird ein neuer Wegweiser-Eintrag erstellt, der einen Separator zwischen altem und neuem Blatt (z.B. den größten im alten Blatt verbleibenden Schlüssel) so-

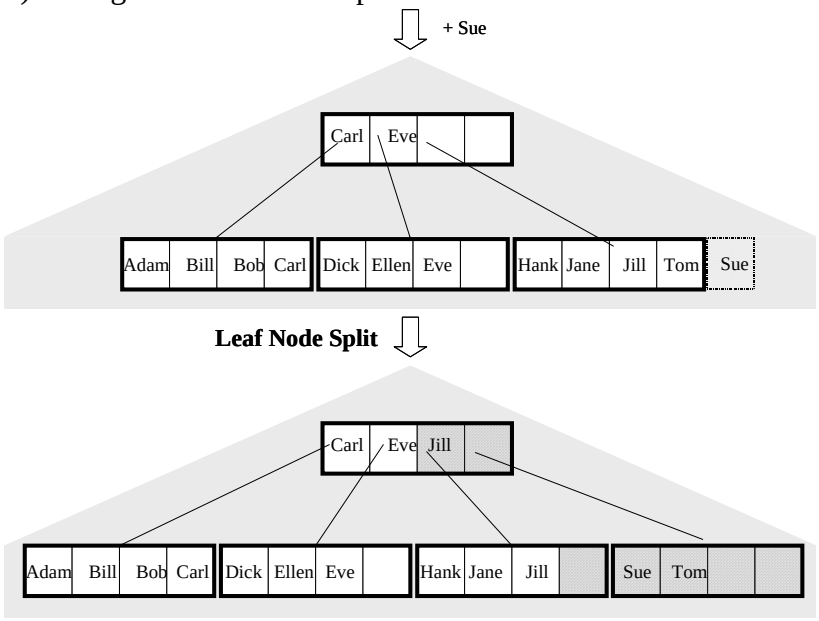
wie einen Zeiger auf das neue Blatt enthält. Dieser Wegweiser wird dann im Vater der beiden betroffenen Blätter eingefügt. Da es auch im Vater Platzknappheit geben kann, muß der Vaterknoten selbst evtl. analog geteilt werden, und ein Split kann sich rekursiv bis zur Wurzel fortpflanzen. Wenn die Wurzel geteilt werden muß, wird eine Wurzel erzeugt, so daß die Baumhöhe um eins wächst.

Die folgenden Abbildungen zeigen die verschiedenen Fälle beim Einfügen.

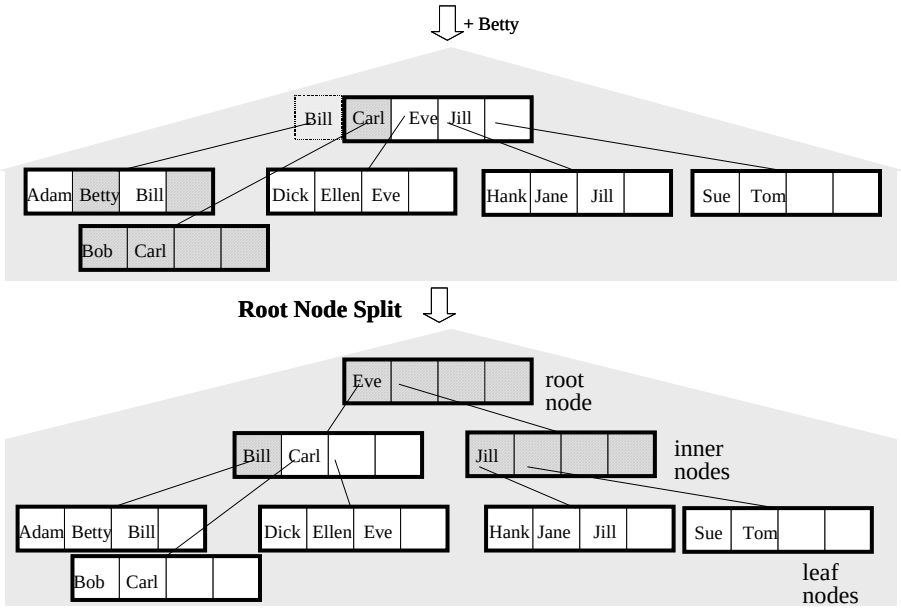
a) Einfügen von 'Ellen' und 'Bob' mit hinreichend freiem Platz im Blattknoten:



b) Einfügen von 'Sue' mit Split eines Blattknotens:



c) Einfügen von 'Betty' mit Blatt-Split und Wurzel-Split:



Pseudocode für das Einfügen eines neuen Schlüssels e in einen B*-Baum:

```
Suche nach einzufügendem Schlüssel e
if e ist noch nicht vorhanden then
  Sei t das Blatt, bei dem die Suche erfolglos geendet hat
  repeat
    if t hat weniger als 2m* bzw. 2m Schlüssel (d.h. ist nicht voll)
    then füge e in t ein
    else /* Knoten-Split */
      Bestimme Median s der 2m* +1 bzw. 2m+1 Schlüssel inkl. e
      Erzeuge Bruderknoten t‘ /* Grow-Phase */
      if t ist Blattknoten then
         Speichere Schlüssel ≤ s in t und Schlüssel > s in t‘
      else Speichere Schlüssel < s (mit Sohnzeigern) in t und
                        Schlüssel > s (mit Sohnzeigern) in t‘ fi
      if t ist Wurzel /* Post-Phase */
      then Erzeuge neue Wurzel r mit Schlüssel s und Zeigern auf t und t‘
      else Betrachte Vater von t als neues t und s (mit Zeiger auf t‘) als e fi
    fi
  until kein Knoten-Split mehr erfolgt
fi
```

14.3 Wie Anfragen ausgeführt werden

Anfragen werden übersetzt in Operatorbäume, die im wesentlichen relationenalgebraischen Ausdrücken auf Mengen, Multimengen und Listen. Gegenüber der Relationenalgebra aus Kapitel 2 gibt es weitere Operatoren, die sich an den Speicherungs- und Zugriffsstrukturen orientieren, z.B. einen Table-Scan-Operator zum sequentiellen Lesen aller Records einer Tabelle, einen Index-Scan-Operator zum Bestimmen aller RIDs zu einem Suchschlüssel, usw. Die wichtigsten Operatoren einer solchen DBS-internen Algebra sind im folgenden kurz zusammengestellt. In den meisten kommerziellen Datenbanksystemen kann man sich den für eine Anfrage erzeugten Operatorbaum mittels des Explain-Kommandos anschauen.

Table-Scan (TABLE ACCESS FULL):
Sequentielles Lesen aller Tupel einer Relation,
wobei auf jedem Tupel ein "Single-Scan"-Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.

RID-Zugriff (TABLE ACCESS BY ROWID):
Direkter Zugriff auf alle Tupel einer Liste von RIDs,
wobei auf jedem Tupel ein "Single-Scan"-Filterprädikat überprüft und
eine Attributprojektion (ohne Duplikateliminierung) vorgenommen werden kann.
Die RID-Liste sollte nach Seitennummern sortiert sein; Seiten mit kleinem "Abstand" auf derselben Platte (z.B. im selben Zylinder) sollten mit einem einzigen Plattenzugriff
gelesen werden (multi-block I/O, list prefetch).

Index-Scan (INDEX RANGE SCAN oder INDEX UNIQUE SCAN):
Zugriff auf alle TIDs von Tupeln, die ein Index-Suchprädikat der Form $wert1 \leq KEY \leq wert2$ erfüllen, wobei KEY der Suchschlüssel des Index ist (ggf. eine Attributkombination).
Realisiert ist dies durch eine Indexsuche nach wert1 mit einem nachfolgenden "Blatt-Scan" bis zum Erreichen des ersten Schlüssels, der größer als wert2 ist.

Sortieren (SORT):
Sortieren einer Tupelmenge nach einem Attribut bzw. einer Attributkombination.
Dabei werden Sortieralgorithmen für Externspeicher verwendet (z.B. Mehrwegemischen).

Durchschnitt, Vereinigung, Differenz (INTERSECTION, UNION, MINUS):
Anwenden von Mengenoperationen auf Tupelmengen oder TID-Listen.
Dies wird in der Regel erst nach vorhergehendem Sortieren angewandt.

Selektion bzw. Filterung (FILTER):
Überprüfen einer Booleschen Bedingung auf den Tupeln einer Tupelliste

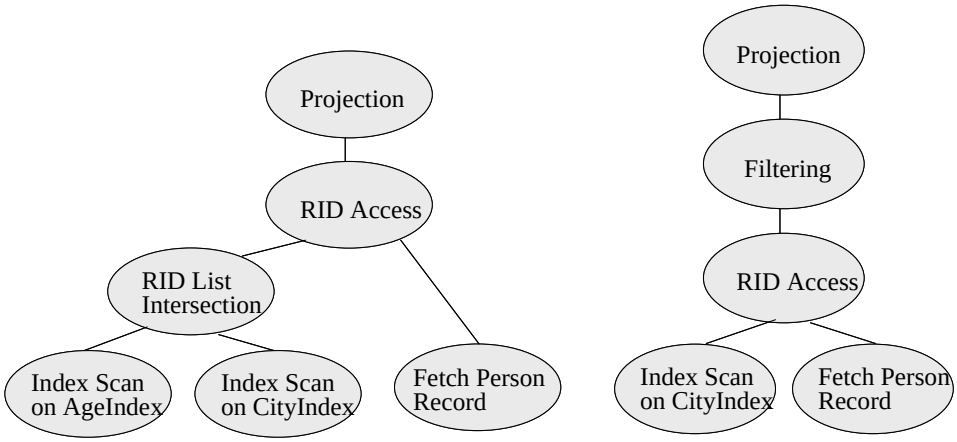
Projektion für Mengen (PROJ-SET):
Projektion von Attributen einer nach diesen Attributen sortierten Tupelliste mit Duplikateliminierung

Projektion für Multimengen (PROJ-MULTI):
Projektion von Attributen einer Tupelliste

Für eine SQL-Anfrage wie

```
Select Name, City, Zipcode, Street
From Person
Where Age < 30 And City = 'Austin'
```

erzeugt die Anfragezerlegungs- und -optimierungsschicht etwa einen der folgenden beiden Operatorbäume als Ausführungsplan:



In Oracle8i werden diese Ausführungspläne folgendermaßen dargestellt:

```
SELECT STATEMENT
        TABLE ACCESS            BY ROWID    Person
                INTERSECT
                        INDEX RANGE SCAN    AgeIndex
                        INDEX RANGE SCAN    CityIndex
```

bzw.

```
SELECT STATEMENT
        FILTER
                TABLE ACCESS    BY ROWID    Person
                        INDEX RANGE SCAN    CityIndex
```

Für Joins und Gruppierungen mit Aggregation gibt es weitere interne Operatoren als Implementierungsvarianten, beispielsweise die folgenden beiden Basisvarianten (für die es wiederum weitere optimierte Varianten gibt):

Nested-Loop-Join (NESTED LOOPS):
Berechnung eines Joins zwischen zwei Tupelmengen (mit Angabe einer expliziten Joinbedingung) mittels einer Doppelschleife.

Merge-Join (MERGE JOIN):
Berechnung eines Equi-Joins zwischen zwei nach dem Join-Attribut sortierten Tupellisten mittels Mischen der Listen.
Dieses Join-Berechnungs-Verfahren heißt häufig auch Sort-Merge-Join, weil u.U. vor dem Mischen noch Sortierschritte stattfinden müssen.

Hash-Join (HASH JOIN):
Berechnung eines Equi-Joins zwischen zwei Tupelmengen durch Abbilden beider Mengen auf eine Hash-Tabelle (mit einer für beide Tupelmengen identischen Hash-Funktion über dem Join-Attribut), so daß Tupel mit gleichem Attributwert auf denselben Eintrag (bzw. Bucket) der Hash-Tabelle abgebildet werden.

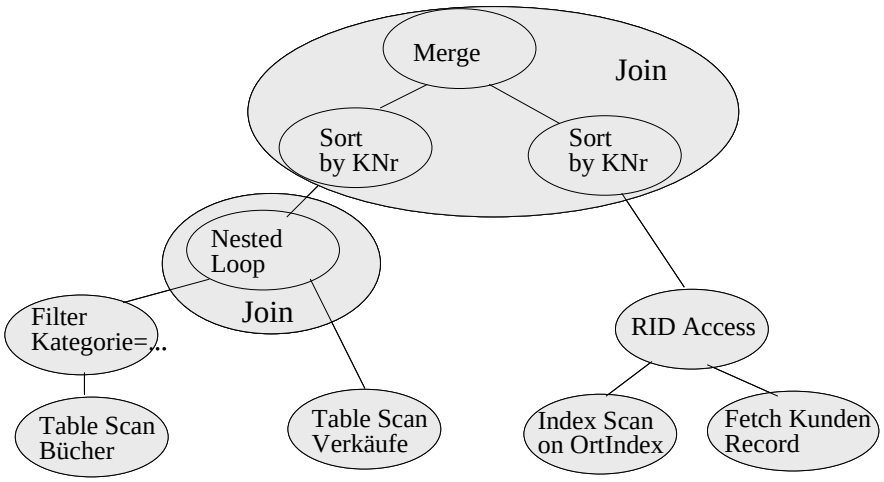
Gruppierung mit Aggregation (GROUP):
Berechnung eines Aggregationsfunktionswerts pro Gruppe auf einer nach dem Gruppierungsattribut sortierten Tupelliste durch sequentielles Lesen aller Tupel.

Hash-Gruppierung mit Aggregation (HASH GROUP):
Berechnung eines Aggegationsfunktionswerts pro Gruppe durch Abbilden aller Tupel auf eine Hashtabelle mit einer Hashfunktion auf dem Gruppierungsattribut und - falls erforderlich - anschließendem SORT und GROUP für jede Hashklasse.

Eine komplexere Query wie

```
Select KNr, Name
From Kunden K, Verkäufe V, Bücher B
Where K.Ort = 'Saarbrücken'
And K.KNr = V.KNr And V.ISBN = B.ISBN
And B.Kategorie = 'Kochbücher'
```

könnte beispielsweise zum folgenden Ausführungsplan führen:



In Oracle8i würde dieser Ausführungsplan folgendermaßen dargestellt:

```
SELECT STATEMENT
        MERGE JOIN
                SORT
                        NESTED LOOP
                                FILTER
                                        TABLE ACCESS    FULL        Bücher
                                TABLE ACCESS            FULL        Verkäufe
                SORT
                        TABLE ACCESS                    BY ROWID    Kunden
                                INDEX RANGE SCAN                    OrtIndex
```

Ein solcher Operatorbaum wird unter Anwendung relationenalgebraischer Äquivalenzregeln erzeugt, beispielsweise der Distributivität von Selektionen (Filter), die sich nur auf eine Tabelle beziehen, über Joins sowie der Assoziativität und Kommutativität von Equi-Joins. Der Anfrageoptimierer generiert eine (u.U. sehr große) Menge alternativer Ausführungspläne. Für jeden Kandidaten werden dann die Zugriffskosten, insbesondere die erwartete Anzahl der Plattenzugriffe, geschätzt, und der beste Ausführungsplan unter den Kandidaten wird schlußendlich gewählt. Diese Optimierung ist ein sehr schwieriges Problem, so daß reale Datenbanksysteme auf Heuristiken zurückgreifen. Bereits die Kostenschätzung für einen einzigen Plan ist schwierig, da die Ausführungskosten in starkem Maße von der Verteilung der Daten selbst abhängen (z.B. von Häufigkeiten von Attributwerten).

Die mengenorientierten Operatoren der Anfrageausführungsschicht eines Datenbanksystems sind sehr gut zur Parallelisierung geeignet, indem man einen Operator "klont" und jedem Klon eine Partition der Eingabedaten übergibt. Dabei kann sowohl I/O- als auch CPU-Parallelität mit sehr gutem Speedup ausgenutzt werden. Die kommerziell führenden Datenbanksysteme laufen alle mit sehr guter Leistung auf Parallelrechnern (mit u.U. 64 und mehr Prozessoren) und parallelen Plattensystemen (sog. Disk-Arrays, mit u.U. Hunderten von Platten).

Der Datenfluß zwischen einem Produzenten- und einem Konsumentenoperator im Operatorbaum (die in einer Kind-Vater-Beziehung im Baum stehen) kann auf zwei Arten realisiert werden. Zum einen kann jeder Operator sein (Teil-)Ergebnis materialisieren, indem er es komplett berechnet und im Hauptspeicher oder in einem temporären Arbeitsbereich auf Magnetplatte(n) ablegt, von wo es der nächste Operator liest. Alternativ dazu kann zwischen bestimmten Operatoren eine Pipeline im Sinne einer Fließbandverarbeitung aufgebaut werden, bei der ein Produzent so schnell wie möglich Tupel seines Output-Stroms, also bevor das gesamt (Teil-)Ergebnis berechnet ist, an den Konsumenten weitergibt. Pipelining vermeidet die Materialisierung von Zwischenergebnissen, ist jedoch deutlich schwieriger zu implementieren, insbesondere in Verbindung mit der Datenparallelität einzelner Operatoren.

Weiterführende Literatur zu Kapitel 14:

- T. Härder, E. Rahm: Datenbanksysteme - Konzepte und Techniken der Implementierung, Springer, 1999
- H. Garcia-Molina, J. Ullman, J. Widom: Database System Implementation, Prentice Hall, 1999
- G. Saake, A. Heuer: Datenbanken - Implementierungstechniken, MITP-Verlag, 1999
- C.T. Yu, W. Meng: Principles of Database Query Processing for Advanced Applications, Morgan Kaufmann, 1998

Kapitel 15: Concurrency Control – Synchronisation von Prozessen und Transaktionen

Wenn mehrere Programme gleichzeitig auf eine Datenbank zugreifen und mindestens ein Programm die Daten ändert, kann es zu Inkonsistenzen kommen, auch wenn jedes einzelne Programm konsistenzbewahrend ist. Die potentiellen Anomalien resultieren aus der parallelen bzw. nebenläufigen Ausführung der Programme. *Nebenläufigkeit (engl.: concurrency)* bedeutet dabei, dass die verschiedenen Schritte mehrerer Programme zeitlich verschränkt ablaufen, also z.B. bei Programm 1 mit den Schritten p11, p12 und p13 und Programm 2 mit den Schritten p21 und p22 in der Reihenfolge p11, p21, p12, p13, p22. Bei einer parallelen Ausführung können sogar Schritte verschiedener Programme echt gleichzeitig ablaufen (z.B. auf einem Multiprozessor-Rechner). Die in diesem Kapitel betrachteten Anomalien treten jedoch bereits bei nebenläufiger Ausführung auf, die wir aus diesem Grund auch als *Quasi-Parallelität* bezeichnen.

Zur Lösung, also garantierten Vermeidung von Anomalien, müssen Programme bzw. deren Instantiierungen zur Laufzeit synchronisiert werden; die Instantiierung bezeichnen wir auch als *Prozesse* oder – im Kontext eines Datenbanksystems – als *Transaktionen*. Die Maßnahmen zur *Synchronisation* bezeichnen wir generell auch als *Concurrency Control* oder Nebenläufigkeitssteuerung. Im folgenden werden zunächst allgemeine Mechanismen zur Synchronisation von Prozessen – auf Betriebssystemebene – und anschließend weitergehende Möglichkeiten zur Synchronisation von Transaktionen – auf Datenbanksystemebene – betrachtet. Die vorgestellten Lösungen für nebenläufige Ausführungen lösen auch die Situationen bei echt paralleler Ausführung und – mit Erweiterungen – auch verteilter Ausführung auf mehreren Rechnern.

15.1 Synchronisation nebenläufiger Prozesse

Unter einem *Prozess* verstehen wir den Ausführungskontext eines beliebigen Programms, also den aktuellen Befehlszähler, die Belegung der Registersätze, die Daten im virtuellen Adressraum usw. Ein Prozess läuft unter der Steuerung des Betriebssystems, das – zusammen mit der Hardware – den Ausführungskontext verwaltet. Synchronisation nebenläufiger Prozesse bedeutet, dass bestimmte Prozesse unter bestimmten Bedingungen auf andere Prozesse bzw. Ereignisse, die von anderen Prozessen ausgelöst werden, warten müssen. Die wichtigste Situation, die Synchronisation erfordert, ist der Zugriff mehrerer Prozesse auf eine gemeinsame Datenstruktur. Um die Datenstruktur konsistent zu halten, darf in vielen dieser Situationen nur maximal ein Prozess zu einer Zeit auf die Datenstruktur zugreifen. Diese Notwendigkeit nennen wir *wechselseitigen Ausschluss (engl.: mutual exclusion)*. Eine andere Situation, die wechselseitigen Ausschluss erfordert, ist die Notwendigkeit, dass von mehreren Prozessen, die dasselbe Programm ausführen, maximal einer in einem bestimmten Abschnitt des Codes sein darf (z.B. der Interrupt-Behandlung des Betriebssystemskerns auf einem Multiprozessor-Rechner). Solche Codeabschnitte nennen wir *kritische Abschnitte (engl.: critical sections)*.

Synchronisation mit Semaphoren

Zur Realisierung von wechselseitigem Ausschluss stellen Betriebssystem Synchronisationsprimitive zur Verfügung, die als *Semaphore* bezeichnet werden (griechisches Wort für Zeichenträger, z.B. für Flaggensignale in der Antike, sinngemäß Ampel). Ein Semaphor ist eine ganzzahlige, nichtnegative Variable mit zwei atomaren, also ununterbrechbaren und unteilbaren, Operationen *P bzw. Wait* und *V bzw. Signal* (von den holländischen Wörtern „passeer = betreten, passieren“ und „verlaat = verlassen“ herrührend – das Semaphorkonzept stammt von Edsger W. Dijkstra). In Unix werden (verallgemeinerte) Semaphore durch die System-Calls semctl, semget und semop unterstützt; Windows hat analoge System-Calls.

Die Semantik der Semaphor-Operationen P und V ist die folgende:

```
Boolean P (Semaphore x) {
        if (x == 1) {x = x-1; return True} else return False;}

void V (Semaphore x) {x = x+1;};
```

Dabei sei x mit dem Wert 1 vorinitialisiert. Der Wert von x kann als die Anzahl verfügbarer Exemplare eines bestimmten Betriebsmittels interpretiert werden. Wenn ein Prozess einen Wert >0 sieht, kann er ein Exemplare dieses Betriebsmittels akquirieren. Durch Initialisierung von x mit einem Wert >1 wird aus dem binären Semaphor ein zählender Semaphor. Essentiell an den Semaphor-Operationen P und V ist, dass beide unteilbar sind. Ein Prozess, der gerade eine P-Operation ausführt, kann also nicht zwischen dem Lesen von x und dem Erniedrigen von x unterbrochen werden. Diese Semantik ist durch das Betriebssystem und die Rechner-Hardware sicherzustellen.

Mit Semaphoren lässt sich wechselseitiger Ausschluss elegant realisieren. Betrachten wir ein Programm p mit Schritten p1, p2, p3 und p4; p2 und p3 sollen zusammen einen kritischen Abschnitt bilden. Wir führen eine Semaphor-Variable mutex ein und erweitern das Programm wie folgt:

```
p1; while (P(mutex)!=True) { }; p2; p3; V(mutex); p4;
```

Bei der nebenläufigen Ausführung zweier Instantiierungen p und p’ dieses erweiterten Programms sind dann beispielsweise die folgenden Reihenfolgen möglich:

p1, p2, p3, p1’, p4, p2’, p3’, p4’
p1, p1’, p2, p3, p2’, p3’, p4, p4’
p1, p1’, p2’, p3’, p2, p3, p4’, p4
usw.,

nicht aber die folgenden verbotenen Reihenfolgen:

p1, p2, p2’, p3, p3’, p4, p4’
p1, p2’, p2, p3’, p3, p4, p4’
p1, p2’, p2, p3, p3’, p4, p4’

Erlaubte und verbotene Reihenfolge der Schritte solcher nebenläufiger Ausführung lassen sich auch durch Ausdrücke sog. *Prozessalgebren* (z.B. CSP - Calculus for Communicating Sequential Processes - von Sir Anthony Hoare) formal charakterisieren.

Implementierung von Semaphoren

Semaphore können auf 3 Arten realisiert werden:

- durch eine *unteilbare Maschineninstruktion* vom Typ *Test-and-Set* (oder einer ähnlichen Instruktion): Test-and-Set <addr> <val> testet den Inhalt der Speicheradresse addr (ein Byte oder ein Wort); falls der Wert gleich 0 ist, wird ein Condition-Code-Register (CC) auf 1 gesetzt und gleichzeitig wird der Inhalt der Speicheradresse auf den Wert val gesetzt; das CC-Register kann von nachfolgenden Sprungbefehlen abgefragt werden
- durch eine *ununterbrechbare Prozedur des Betriebssystemskerns* und einen entsprechenden *System-Call*: Diese Prozedur benötigt auf einem Multiprozessor-Rechner ihrerseits eine unteilbare Maschineninstruktion vom Test-and-Set-Typ, ist also mit Variante 1 implementiert.
- durch eine selbst implementierte Prozedur, die nur dann System-Calls durchführt, wenn ein Prozess warten muss.

Grundsätzlich ist bei allen 3 Varianten zu überlegen, wie man verfährt, wenn ein Prozessor bei einer P-Operation keinen Erfolg hat. Es gibt dafür 2 Möglichkeiten:

- *Busy Wait*: die P-Operation wird so oft wiederholt, bis sie Erfolg hat. Dies macht nur auf einem Multiprozessor-Rechner Sinn; der potentielle Nachteil ist, dass beim Warten CPU-Zeit verbraucht wird. Wenn kritische Abschnitte sehr kurz sind und nur selten auftreten, ist Busy Wait akzeptabel.
- *Lazy Wait*: bei einer nicht erfolgreichen P-Operation legt sich ein Prozess schlafen (deaktiviert sich mittels System-Call selbst). Entweder muss er sich vom Betriebssystem periodisch wecken lassen und dann erneut die P-Operation versuchen, oder die anderen Prozesse müssen bei der V-Operation schlafende Prozesse, die auf die Freigabe des betreffenden Semaphors warten, durch System-Calls wecken (lassen). Der potentielle Nachteil von Lazy Wait ist, dass System-Calls sehr aufwändig sind; ein System-Call benötigt einige Hundert oder sogar einige Tausend Maschineninstruktionen.

Die beiden Varianten busy wait und lazy wait können entweder direkt in die Implementierung der P- und V-Operationen integriert werden, oder die Wahl kann den Prozessen selbst überlassen bleiben.

Die selbst implementierten Prozeduren zur Realisierung der P- und V-Operationen könnten mit Lazy Wait beispielsweise auf der folgenden Datenstruktur aufbauen:

```
struct latch {
     int Status;
     int NumProcesses;
     int Pids[MaxProcesses]; };
```

Diese Datenstruktur implementiert sozusagen einen leichtgewichtigen Semaphore, im Industriejargon auch “Latch” genannt, inklusive einer Warteschlange wartender Prozesse. Der Prozess auf Positions Pids[0] ist der jeweils aktuelle Besitzer des Semaphors. Zur Manipulation der Latch-

Datenstruktur selbst wird die Test-and-Set-Methode auf dem Status-Feld verwendet. Der Code für P und V sieht dann folgendermaßen aus:

```
int P (latch) {
     while ( Test-and-Set(latch.Status, 0) != True ) { };
     latch.Pids[latch.NumProcesses] = getOwnProcessId();
     latch.NumProcesses++;
     if (latch.NumProcesses == 1)
        {latch.Status = 1}
     else {latch.Status = 1; sleep ( )};
     return True; };

int V (latch) {
     while ( Test-and-Set(latch.Status, 0) != True) { };
     for (i=0; i < latch.NumProcesses; i++)
        {latch.Pids[i] = latch.Pids[i+1]};
     latch.NumProcesses--;
     if (latch.NumProcesses == 0)
        {latch.Status = 1}
     else {latch.Status = 1, wakeup (Pids[0])};
     return True;};
```

Datenbanksysteme, Web-Server (z.B. Apache) und Betriebssysteme, die intern mit nebenläufigen Prozessen (häufig leichtgewichtigen Prozessen, sog. Threads) arbeiten, verwenden derartige Low-Level-Routinen zur Synchronisation der Zugriffe auf interne Datenstrukturen.

15.2 Synchronisation nebenläufiger Transaktionen

15.2.1 Transaktionskonzept

Eine *Transaktion* ist eine Folge von (Lese- und Änderungs-) Operationen auf einer oder mehreren Datenbanken, für die die zugrundeliegenden Datenbanksysteme die folgenden Eigenschaften (häufig *ACID-Eigenschaften* genannt) garantieren:

- *Atomarität (engl.: atomicity):*
  Die Änderungen einer Transaktion werden entweder vollständig oder gar nicht durchgeführt (Alles-oder-nichts-Eigenschaft). Im Fehlerfall oder auf "Wunsch" eines Programms werden bereits durchgeführte Änderungen einer Transaktion rückgängig gemacht.

- *Dauerhaftigkeit / Persistenz (engl.: durability / persistence):*
  Die Änderungen einer abgeschlossenen Transaktion gehen auch im Fall von Fehlern der Hardware oder Systemsoftware nicht verloren.

- *Isolation (engl.: isolation):*
  Die Transaktion sieht die Datenbanken so, als gäbe es keine parallelen Transaktionen, und die Änderungen der Transaktion werden bis zum Abschluß der Transaktion vor parallelen Transaktionen verborgen (Äquivalenz zum Einbenutzerbetrieb).

- *Integritätserhaltung (engl.: consistency-preservation, integrity-preservation):*
  Die Transaktion transformiert die Datenbanken von einem - bezüglich der spezifizierten Integritätsbedingungen - konsistenten Zustand in einen anderen konsistenten Zustand.

Die Eigenschaften der Atomarität und Persistenz bedeuten, daß Datenbankanwendungsprogramme größtenteils so geschrieben werden können, als gäbe es keine Fehler der Hardware oder Systemsoftware. Die Transaktionsverwaltung eines DBS stellt mit der **Recovery-Komponente** weitreichende Fehlertoleranzmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht transparent für den Endbenutzer).

Die Eigenschaft der Isolation bedeutet, daß Datenbankanwendungsprogramme so geschrieben werden können, als würde das DBS (ggf. sogar ein verteiltes DBS oder mehrere DBS in einem verteilten System) im Einbenutzerbetrieb arbeiten, obwohl ein DBS in der Regel aus Leistungsgründen effektiv im Mehrbenutzerbetrieb arbeitet. Die Transaktionsverwaltung eines DBS stellt mit der **Concurrency-Control-Komponente** automatische Synchronisationsmaßnahmen zur Verfügung, und zwar transparent für die Anwendungsentwicklung (und natürlich erst recht für den Endbenutzer).

Die Eigenschaft der Integritätserhaltung wird durch die Transaktionsverwaltung nur unterstützt, nicht aber voll erzwungen. Sie kann nur in Verbindung mit entsprechenden Maßnahmen der Anwendungsentwicklung erzwungen werden. Es wird erwartet, daß jedes Anwendungsprogramm bei Beendigung einer Transaktion entsprechende Integritätsprüfungen durchführt oder anstößt (z.B. über Trigger). Falls dann eine Transaktion im fehlerlosen Einbenutzerbetrieb integritätserhaltend ist, garantiert die Transaktionsverwaltung, daß die Integritätsbewahrung auch im Mehrbenutzerbetrieb und bei Auftreten von Hardware- und Systemsoftwarefehlern gilt.

Transaktionen werden dynamisch durch entsprechende DBS-Aufrufe des Anwendungsprogramms erzeugt. In SQL gibt es dafür drei Anweisungen:

- CONNECT ... (BOT = Begin-of-Transaction):
  kennzeichnet den Beginn einer Transaktion.
- COMMIT WORK (EOT = End-of-Transaction):
  kennzeichnet das ("erfolgreiche") Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für gültig.
- ROLLBACK WORK (RBT = Rollback-Transaction, häufig auch Abort genannt):
  kennzeichnet das ("erfolglose") Ende einer Transaktion
  und erklärt alle Änderungen der Transaktion für ungültig.

Implizit kennzeichnen EOT und RBT außerdem den Beginn einer neuen Transaktion,
die bis zum nächsten EOT, RBT oder DISCONNECT dauert.

Beispiele für simple Transaktionsprogramms mit Embedded SQL:

DebitCredit:

```
void main ( ) {
   EXEC SQL BEGIN DECLARE SECTION
      int b /*balance*/, a /*accountid*/, amount;
   EXEC SQL END DECLARE SECTION;
   /* read user input */
   scanf („%d %d“, &a, &amount);
   /* read account balance */
   EXEC SQL Select Balance into :b From Account
      Where Account_Id = :a;
   /* add amount (positive for debit, negative for credit) */
   b = b + amount;
   /* write account balance back into database */
   EXEC SQL Update Account
      Set Balance = :b Where Account_Id = :a;
   EXEC SQL Commit Work;
}
```

FundsTransfer:

```
void main ( ) {
   /* read user input */
   scanf („%d %d %d“, &sourceid, &targetid, &amount);
   /* subtract amount from source account */
   EXEC SQL Update Account
      Set Balance = Balance - :amount Where Account_Id = :sourceid;
   /* add amount to target account */
   EXEC SQL Update Account
      Set Balance = Balance + :amount Where Account_Id = :targetid;
   EXEC SQL Commit Work;
}
```

15.2.2 Probleme bei unkontrollierter (Quasi-) Parallelität

Die (quasi-) parallele Ausführung von Transaktionen ist wünschenswert, um möglichst gute Antwortzeiten zu erzielen. Dazu müssen Plattenzugriffe und CPU-Arbeit für verschiedene Transaktionen parallel erfolgen. Um aber auch korrekte Resultate zu gewährleisten, muß die Parallelität u.U. wieder eingeschränkt werden.

Problemtyp "verlorene Änderung" (lost update):

P1	Time	P2
	/* x = 100 */	
r (x)	1	
	2	r (x)
x := x+100	4	x := x+200
w (x)	5	
	/* x = 200 */	
	6	w (x)
	/* x = 300 */	

Kern des Problems ist die nebenläufige Ausführung von Read- und Write-Schritten:
R1(x) R2(x) W1(x) W2(x)

Problemtyp "inkonsistentes Lesen":

P1	Time	P2
	1	r (x)
	2	x := x – 10
	3	w (x)
sum := 0	4	
r (x)	5	
r (y)	6	
sum := sum +x	7	
sum := sum + y	8	
	9	r (y)
	10	y := y + 10
	11	w (y)

Kern des Problems ist die nebenläufige Ausführung von Read- und Write-Schritten:
R2(x) W2(x) R1(x) R1(y) R2(y) W2(y)

Problemtyp "Dominoeffekt" (dirty read):

P1	Time	P2
r (x)	1	
x := x + 100	2	
w (x)	3	
	4	r (x)
	5	x := x - 100
failure & rollback	6	
	7	w (x)

Kern des Problems ist die nebenläufige Ausführung von Read-, Write- und Abort-Schritten (sowie potentiellen Commit-Schritten):
R1(x) W1(x) R2(x) W2(x) A1 ... C2 (mit potentiellem C2).

Probleme mit Lösungsansätzen mittels Semaphoren

Man könnte versuchen, die erwähnten Synchronisationsprobleme mittels Semaphoren zu lösen. Dabei wären zwei Varianten verfolgenswert:

- Wir führen einen Semaphor mutex für die gesamt Datenbank ein. Eine Transaktion führt vor ihrem ersten Datenbankzugriff eine P-Operation auf mutex durch und nach ihrem letzten Datenbankzugriff eine V-Operation. Dies wäre korrekt, würde aber den gesamten Betrieb zwangsweise sequentialisieren. Durch Einführung von Lese- und Schreibmodi bzw. je einen Semaphor für Lesen und Schreiben könnte man dies graduell verbessern, bei hoher Last von Änderungstransaktionen aber wäre die Methode völlig unakzeptabel.
- Wir führen pro Datenbankobjekt einen Semaphor ein. Vor dem Zugriff auf Objekt x wird P auf dem Semaphor für x aufgerufen und nach dem Zugriff V. Dies alleine aber genügt nicht, um korrekte Ausführung sicherzustellen, wie der folgende Beispielablauf (mit von oben nach unten fortschreitender Zeit) zeigt:

Prozess 1: FundsTransfer von a nach b	*Prozess 2: Prüfen der Summe von a und b*
shared semaphore mutex_a;	
shared semaphore mutex_b;	
P(mutex_a);	
Update Konto Set Stand = Stand – 3000 Where KontoNr = a;	
V(mutex_a);	
	P(mutex_a);
	Select Stand From Konto Where KontoNr = a;
	V(mutex_a);
	P(mutex_b); ...; V(mutex_b);
P(mutex_b); ...; V(mutex_b);	

Eine korrekte Lösung wäre dagegen, wenn beide Prozesse alle ihre V-Operationen erst nach allen Datenbankzugriffen (also z.B. mit dem Commit Work) aufrufen.
Insgesamt zeigt diese Überlegung, dass Semaphore nur Mechanismen sind, und dass man zusätzliche Überlegungen – insbesondere auch Theoriebildung – für korrekte Synchronisationsstrategien braucht. Die andiskutierten Semaphor-basierten Lösungen haben folgende Nachteile:

- Es ist – ohne weitere Theorie – unklar, wann jeweils ein Semaphor frühestens freigegeben werden darf. Anwendungsprogrammierer wären hinsichtlich des korrekten Umgangs mit Semaphoren überfordert, da man Wechselwirkungen des eigenen Programms mit allen anderen, potentiell nebenläufig ausgeführten Programmen beachten muss (selbst, wenn man den Code dieser Programme gar nicht kennt!).
- Es ist unklar, wie man mit Insert- und Delete-Operationen auf der Datenbank oder mit prädikatorientierten Operationen umgeht.
- Es ist nicht einfach ersichtlich, ob es durch die Verwendung mehrerer Semaphore womöglich zu Deadlocks (Verklemmungen, zyklischen Wartesituationen) kommen kann. Falls es dazu kommt, ist unklar, wie man solche Situationen erkennt und behebt.
- Die Verwaltung sehr vieler Semaphore (z.B. als Unix-Semaphore) wäre extrem aufwändig.

Konsequenz: Das DBS sollte die notwendigen Semaphore und die P- und V-Operationen darauf selbst automatisch generieren!

15.2.3 Korrektheitskriterium Serialisierbarkeit

Intuitives Kriterium:

Es dürfen nur (quasi-) parallele Abläufe von Transaktionen zugelassen werden, die zu einer sequentiellen Ausführung der mit Commit beendeten Transaktionen äquivalent sind. Dadurch werden alle Anomalien wie "verlorene Änderungen", "inkonsistentes Lesen", usw. vermieden.

Formalisierung des intutitiven Korrektheitskriteriums:

- Modellbildung für (quasi-) parallele Abläufe von Transaktionen
- Definition der Äquivalenz von Abläufen
- Entwicklung (beweisbar) korrekter Concurrency-Control-Verfahren
  zur automatischen Synchronisation durch das DBS

Begriffsbildung und Definitionen:

Einschränkung für den Rahmen der Vorlesung:
Es wird angenommen, daß stets alle Transaktionen mit Commit beendet werden.
(Für den allgemeinen Fall siehe Theorie der Rücksetzbarkeit und verwandte Theorien.)

Ein **Schedule** s ist ein Tripel (T, A, <) mit:
- T ist eine Menge von Transaktionen (eigentlich nur Transaktionsnummern).
- A ist eine Menge von (Datenbankzugriffs-) Aktionen der Form Ri (x) oder Wi (x), wobei
  - R und W für Lesezugriffe (Reads) und Schreibzugriffe (Writes) stehen,
  - x ein Objekt der Datenbank ist, auf das sich der Zugriff bezieht (z.B. eine Seite) und
  - jede Aktion zu genau einer Transaktion Ti gehört.
- < ist eine partielle Ordnung auf der Aktionenmenge A, wobei für jedes Aktionenpaar
  $\langle a, b \rangle \in A \times A$ gelten soll:
  Wenn a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist,
  dann müssen a und b bezüglich < geordnet sein; es muß also $(a < b) \vee (b < a)$ gelten.

Zwei Aktionen $a, b \in A$ in einem Schedule s = (T, A, <) sind in **Konflikt**, wenn
- a und b auf dasselbe Datenbankobjekt zugreifen und a oder b ein Schreibzugriff ist und
- a und b zu verschiedenen Transaktionen gehören.

Die beiden Aktionen a und b werden auch als "Konfliktpaar" bezeichnet.

Bemerkungen:
- Es ist wichtig, daß die Aktionen eines Konfliktpaars geordnet sind, damit überhaupt
  eine Aussage über deren Effekt möglich ist.
- Aktionen, die nicht in Konflikt sind, können innerhalb eines Schedules in beliebiger
  Reihenfolge ausgeführt werden, ohne daß dadurch unterschiedliche Effekte entstehen.
  Mit anderen Worten:
  Um zu entscheiden, ob zwei Schedules bezüglich ihrer Effekte äquivalent sind,
  ist nur die Reihenfolge der Aktionen in den Konfliktpaaren von Bedeutung.

Ein Schedule s = (T, A, <) heißt **seriell**, wenn für je zwei Transaktionen Ti, Tj $\in$ T gilt, daß
entweder alle Aktionen von Ti bezüglich < vor allen Aktionen von Tj liegen
oder alle Aktionen von Tj vor allen Aktionen von Ti.
In einem seriellen Schedule gibt es somit keine (quasi-) parallelen Transaktionen.

Beispiele:
(Es wird in den Beispielen nur A angegeben; T steckt implizit in den Indizes der Aktionen,
und < ist implizit durch die Anordnung der Aktionen von links nach rechts gegeben.
In den Beispielen ist < immer eine totale Ordnung.)

Schedule s1: R1 (a) W1 (a) R2 (a) R2 (b) R1 (b) W1 (b)

Schedule s2: R1 (a) W1 (a) R2 (a) R3 (b) R2 (b) W2 (b) R3 (c) W3 (c) R1 (c)

Schedule s2': R3 (b) R3 (c) W3 (c) R1 (a) W1 (a) R1 (c) R2 (a) R2 (b) W2 (b)

Konfliktpaare für s1: <W1(a), R2(a)>, <R2(b), W1(b)>

Konfliktpaare für s2: <W1(a), R2(a)>, <R3(b), W2(b)>, <W3(c), R1(c)>

Konfliktpaare für s2': <W3(c), R1(c)>, <R3(b), W2(b)>, <W1(a), R2(a)>

Definition der Äquivalenz von Schedules:

Zwei Schedules s und s' sind **(konflikt-) äquivalent**, wenn sie dieselben Transaktionsmengen und Aktionsmengen haben und in beiden Schedules dieselben Konfliktpaare (jeweils mit derselben Aktionsreihenfolge) vorliegen.

Definition der Korrektheit von nichtseriellen Schedules:

Ein Schedule s ist **(konflikt-) serialisierbar**, wenn er zu einem seriellen Schedule äquivalent ist.

Bemerkungen:
- Serialisierbarkeit ist die Formalisierung des intuitiven Kriteriums der Äquivalenz zum Einbenutzerbetrieb.
- Damit Serialisierbarkeit als Korrektheitskriterium Sinn macht, sind folgende (nicht beweisbare) Annahmen implizit zugrundegelegt:
  - Ein serieller Schedule ist korrekt (unter der wiederum impliziten Annahme, daß das Transaktionsprogramm an sich korrekt, insbesondere integritätserhaltend ist).
  - Es wird angenommen, daß alle Transaktionen von verschiedenen, unabhängigen Benutzern stammen und daß für diese Benutzer jede beliebige sequentielle Ausführung der Transaktionen akzeptabel ist.

Algorithmische Überprüfung der Serialisierbarkeit eines Schedules:

Sei s = (T, A, <) ein Schedule. Der **Abhängigkeitsgraph** für diesen Schedule ist ein gerichteter Graph mit

- Transaktionen als Knoten und
- einer Kante zwischen Ti und Tj genau dann, wenn es ein Konfliktpaar <a, b> in s gibt, so daß a zu Ti und b zu Tj gehört und a < b gilt.

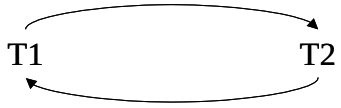
Satz:
Ein Schedule ist serialisierbar genau dann, wenn sein Abhängigkeitsgraph azyklisch ist, d.h. keine Zyklen enthält.

Man erhält einen zum Schedule äquivalenten seriellen Schedule - eine **Serialisierung** - durch topologische Sortierung des Abhängigkeitsgraphen (eine totale Ordnung der Transaktionen, die die durch die Kanten des Graphen gegebene partielle Ordnung beinhaltet):

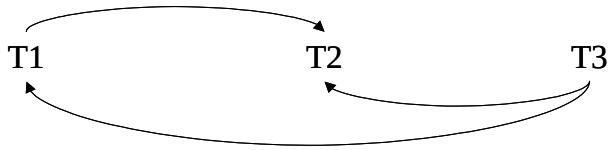
- Man wähle eine beliebige Quelle des Graphen (d.h. einen Knoten ohne eingehende Kanten) und entferne diese sowie alle davon ausgehenden Kanten.
- Falls der Graph noch Knoten hat, wiederholt man diese Prozedur für den verbleibenden Graphen.

Beispiele:

Abhängigkeitsgraph für s1:



Abhängigkeitsgraph für s2:



15.2.4 Das strikte Zweiphasen-Sperrprotokoll

Das DBS fordert für eine Transaktion vor dem Zugriff auf ein Objekt eine Sperre auf dem Objekt an. Sperren sind im Prinzip Semaphore, die durch das DBS selbst implementiert werden. Dazu stellt die Concurrency-Control-Komponente die folgenden zwei Operationen für Sperren zur Verfügung:

- Lock (Transaktion Ti, Objekt x, Modus m) und
- Unlock (Transaktion Ti, Objekt x).

Das DBS kann nie zwei Transaktionen gleichzeitig eine Sperre auf demselben Objekt (in unverträglichen Sperrmodi) geben. Im Sperrkonfliktfall muß die "spätere" Transaktion bis zur Sperrfreigabe warten. Die Concurrency-Control-Komponente legt in diesem Fall die Transaktion schlafen und weckt sie bei Sperrfreigabe automatisch wieder auf. Zu diesem Zweck verwaltet die Concurrency-Control verschiedene Warteschlangen.

Beispiel:
(mit der Notation Li (x) für Lock (Ti, x, Exclusive) und Ui (x) für Unlock (Ti, x)**:**

L1 (a) W1 (a) L1 (c) W1 (c) U1 (a) U1 (c)
L2 (b) W2 (b) L2 (c) - - - - - - - - - - - - W2 (c) U2 (b) U2 (c)

Zweiphasen-Sperrprotokoll (Two-Phase Locking, 2PL):

Regel 1: Bevor eine Transaktion auf ein Objekt zugreift, muß das Objekt für die Transaktion gesperrt werden ("Wachstumsphase" der Transaktion).

Regel 2: Nachdem eine Transaktion eine Sperre freigegeben hat, darf sie keine weiteren Sperren mehr erwerben ("Schrumpfungsphase" der Transaktion).

Satz:
Das Zweiphasen-Sperrprotokoll garantiert, daß alle entstehenden Schedules serialisierbar sind.

Beweis:

Sei s ein Schedule, der unter dem 2PL entstanden ist.
Annahme: s ist nicht serialisierbar.
$\Rightarrow$ es gibt einen Zyklus im Abhängigkeitsgraphen
$T1 \rightarrow T2 \rightarrow T3 \rightarrow ... \rightarrow Tk \rightarrow T1$,
also Konfliktpaare
<a1 (x1), a2 (x1)>, <a2 (x2), a3 (x2)>, ..., <ak (xk), a1 (xk)>
Da Objekte vor dem Zugriff gesperrt werden und andere Transaktionen erst nach der entsprechenden Sperrfreigabe auf das Objekt zugreifen können, gilt bezüglich der Reihenfolge der Lock- und Unlock-Operationen:
U1 (x1) < L2 (x1), U2 (x2) < L3 (x2), ..., Uk (xk) < L1 (xk)
Wegen Regel 2 des 2PL gilt ferner:
L2 (x1) < U2 (x2), L3 (x2) < U3 (x3), ...
$\Rightarrow$ U1 (x1) < L2 (x1) < U2 (x2) < L3 (x2) < U3 (x3) < ... < Uk (xk) < L1 (xk)
Dann aber wäre wegen U1 (x1) < L1 (xk) die Regel 2 für Transaktion T1 verletzt.
Widerspruch ! $\Rightarrow$ Behauptung

Beobachtungen:

- Das 2PL ist nicht ausreichend, um Domino-Effekte zu verhindern (siehe Beispiel für Domino-Effekte).
- Um das 2PL korrekt anwenden zu können, muß man für jede Transaktion wissen, ab welchem Zeitpunkt im Laufe ihrer Ausführung keine weiteren Objekte mehr benötigt werden, die nicht bereits für die Transaktion gesperrt sind. Ab diesem Zeitpunkt können Sperren freigegeben werden.
  Dieses A-priori-Wissen ist unrealistisch, da die von der Transaktion benötigten Objekte von der Ausführungsdynamik des zugrundeliegenden Transaktionsprogramms abhängen.

Striktes Zweiphasen-Sperrprotokoll (Strict Two-Phase Locking, Strict 2PL):

Regel 1: wie beim allgemeinen 2PL

Regel 2: Halte alle jemals für eine Transaktion erworbenen Sperren bis zum Ende der Transaktion (Abschluß des Commit oder Rollback) und gebe sie dann auf einmal frei. (Die Schrumpfungsphase fällt also zu einem einzigen Zeitpunkt zusammen.)

Bemerkungen:

- Das strikte 2PL gewährleistet die Äquivalenz (in einem weitergehenden Sinne) eines parallelen Ablaufs von Transaktionen (die mit Commit oder Rollback beendet sind) zu einem sequentiellen Ablauf derjenigen Transaktionen, die mit Commit beendet sind (ohne Beweis). Das strikte 2PL verhindert also auch Domino-Effekte.
- Das strikte 2PL ist als Concurrency-Control-Protokoll in praktisch allen kommerziellen DBS implementiert.
- Das 2PL kann auf verschiedenen Sperrgranulaten implementiert werden:
  - Seitensperren (in den meisten DBS)
  - Relationssperren oder Tablespace-Sperren (sehr grob und damit restriktiv)
  - Tupelsperren und Sperren auf Indexeinträgen
    (sehr feines Granulat und daher bezüglich der Mehrbenutzerleistung am besten, aber auch sehr komplex und daher nur in den besseren DBS implementiert)
  - Kombinationen (die meisten Transaktionen sperren z.B. Seiten, einige lange Transaktionen aber sperren ganze Relationen), sog. Mehrgranulats-Sperrverfahren
- Das 2PL kann mit verschiedenen Sperrmodi realisiert werden. In der Regel sind die Sperrmodi
  Exclusive - für Writes - und *Shared* - für Reads -
  vorgesehen.
  Der Sperrmodus für einen Zugriff wird jeweils vom DBS automatisch gewählt.
  Für Sperrmodi muß eine *Kompatibilitätsmatrix (Verträglichkeitsmatrix)* spezifiziert sein, die festlegt, ob eine Sperranforderung im Modus m gewährt werden darf, wenn das zu sperrende Objekt bereits für eine andere Transaktion im Modus k gesperrt ist.

Beispiel einer Kompatibilitätsmatrix:

		Transaktion fordert eine Sperre an im Modus	
		Shared	*Exclusive*
Objekt ist gesperrt im Modus	*Shared*	+	-
	Exclusive	-	-

Deadlocks (Verklemmungen):

Das 2PL kann zu Deadlocks führen, bei denen zwei oder mehr Transaktionen in einem Zyklus wechselseitig auf die Freigabe von Sperren warten. Die Concurrency-Control-Komponente erkennt solche Deadlocks, indem sie über alle Wartebeziehungen in Form eines Wartegraphen Buch führt und - periodisch oder bei jedem Sperrkonflikt - einen Zyklustest für den Wartegraphen durchführt. Falls ein Zyklus gefunden wird, wird eine der am Zyklus beteiligten Transaktionen als Deadlock-Opfer ausgewählt und zurückgesetzt (von der Recovery-Komponente des DBS).

Das Anwendungsprogramm erhält in diesem Fall einen speziellen SQLCODE für die SQL-Anweisung, bei der der Deadlock entstand. Das Programm sollte in einem solchen Fall selbst den erneuten Start der Transaktion veranlassen (ggf. nach erneuten Initialisierungen), damit der Deadlock und seine Behandlung gegenüber dem Endbenutzer transparent ist.

Beispiele:
(mit der Notation Xi (x) für Lock (Ti, x, Exclusive) und Si (x) für Lock (Ti, x, Shared))

Beispiel1:

```
X1 (a)  W1 (a)                          X1 (b) - - - - - - - -
                X2 (b)  W2 (b) X2 (a) - - - - - - - - - - - -
```

Beispiel 2:

```
S1 (a)  R1 (a)                  X1 (a)  - - - - - - - - - - - -
                S2 (a)  R2 (a)          X2 (a) - - - - - - - -
```

Der zweite Fall - Deadlock aufgrund von Sperrkonversionen (Verschärfung einer Shared-Sperre in eine Exclusive-Sperre) - tritt in der Praxis potentiell häufig auf. Er kann durch zusätzliche Sperrmodi weitgehend vermieden werden. Diese erfordern aber u.U. spezielle Hinweise durch das Programm (die der Programmierer vorsehen muß). Aus diesem Grund hat SQL spezielle Anweisungszusätze wie z.B. DECLARE C CURSOR FOR SELECT ... *FOR UPDATE* ...

Weiterführende Literatur zu Kapitel 15:

- G. Weikum, G. Vossen: Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001
- J. Gray, A. Reuter: Transaction Processing – Concepts and Techniques, Morgan Kaufman, 1993

Kapitel 16: Transaktionsorientierte Daten-Recovery

16.1 Crash-Recovery

Fehlerkategorien (die die Atomarität oder Persistenz von Transaktionen gefährden)**:**

1) Fehler im Anwendungsprogramm
2) Fehler in der Systemsoftware (BS oder DBS),
   die zum "Systemabsturz" (engl.: system failure, (soft) crash) führen
   - "Bohrbugs": deterministisch und reproduzierbar;
     sind in gut ausgetester Systemsoftware so gut wie eliminiert
   - "Heisenbugs": schwer einzugrenzen und praktisch nicht reproduzierbar;
     treten vor allem in Überlastsituationen im Mehrbenutzerbetrieb auf
3) Stromausfall
4) Transienter Hardwarefehler (CPU, Speicher)
5) Fehlbedienung (Systemadministrator, Operateur, Techniker)
6) Plattenfehler (Media-Fehler),
   der zum Verlust von permanent gespeicherten Daten führt
   (engl.: disk failure, hard crash)
7) Katastrophe (Feuer, Erdbeben, etc.)

Fehlerbehandlung durch das DBS:

1) Unvollendete Transaktionen von fehlerhaft beendeten Programmen werden zurückgesetzt.
2) - 5)
   Das DBS wird im *Warmstart* neu gestartet und führt dabei eine ***Crash-Recovery*** durch.
   Dabei werden alle vor dem Absturz unvollendeten Transaktionen zurückgesetzt
   (***Undo*** von Transaktionen).
   Änderungen von beendeten Transaktionen werden nötigenfalls DBS-intern wiederholt
   (***Redo*** von Transaktionen).
   Bemerkungen:
   - Transaktionen, für die die Recovery ein Undo durchgeführt hat, müssen vom Anwendungsprogramm oder sogar vom Endbenutzer neu gestartet werden.
   - Das Prinzip einer "Backward Recovery" (Undo mit anschließender Fortsetzung der Verarbeitung) hat sich gegenüber einer "Forward Recovery" (z.B. instruktionssynchrone Verarbeitung auf einem zweiten Rechner, der im Fehlerfall die Verarbeitung fortsetzt) vor allem im Hinblick auf "Heisenbugs" sehr bewährt.
   - Die Fehlerkategorie 3 kann durch ununterbrechbare Stromversorgung eliminiert werden.
6) Wiederherstellung der verlorenen Daten durch Aufsetzen auf einer Archivkopie der Datenbank und Wiederholung der Änderungen abgeschlossener Transaktionen (***Media-Recovery***).
7) Änderungen abgeschlossener Transaktionen müssen auf einem zweiten, genügend weit entfernten Rechner doppelt erfaßt werden; im Katastrophenfall übernimmt der zweite Rechner die Verarbeitung.

Arbeitsprinzipien der Recovery-Komponente:

- Um im Fehlerfall Undo und Redo durchführen zu können, müssen Änderungen auf einem hinreichend stabilen Speichermedium (für die Crash-Recovery auf Platte, für die Media-Recovery auf mehreren Platten und/oder Band) protokolliert werden. Für jede Änderung gibt es eine *Undo-Information*, um die Änderung rückgängig machen zu können, und eine *Redo-Information*, um die Änderung wiederholen zu können.

- ***Atomaritätsprinzip:***
  Vor einer Veränderung der permanenten Datenbank (durch Zurückschreiben geänderter Seiten aus dem Puffer auf die Platte) muß die entsprechende Undo-Information auf stabilen Speicher geschrieben werden.

- ***Persistenzprinzip:***
  Vor dem Commit einer Transaktion muß die entsprechende Redo-Information auf stabilen Speicher geschrieben werden.

- ***Idempotenzprinzip:***
  Da es während eines Warmstarts zu einem erneuten Ausfall kommen kann, müssen Recovery-Maßnahmen beliebig oft wiederholbar sein.
  Dabei muß sichergestellt sein, daß
  - zweimaliges Undo derselben Änderung denselben Effekt hat wie einmaliges Undo,
  - zweimaliges Redo derselben Änderung denselben Effekt hat wie einmaliges Redo,
  - Undo einer Änderung, die durch den Systemabsturz bereits verloren gegangen ist, keinen Effekt hat und
  - Redo einer Änderung, die durch den Systemabsturz gar nicht verloren gegangen ist, keinen Effekt hat.

Generelle Zielsetzungen der Recovery-Komponente:

- Hohe Verfügbarkeit = MTTF / (MTTF + MTTR)

  (Eine Verfügbarkeit von 99 % impliziert beispielsweise eine "Auszeit" von ca. 5000 Minuten pro Jahr, während der das System nicht verfügbar ist;
  eine Verfügbarkeit von 99.9 % impliziert eine "Auszeit" von ca. 500 Minuten pro Jahr.)

  mit MTTF = Mean Time To Failure

  MTTR = Mean Time to Repair

- Schnelle Recovery beim Warmstart (kurze MTTR)
- Geringer Overhead im laufenden Normalbetrieb

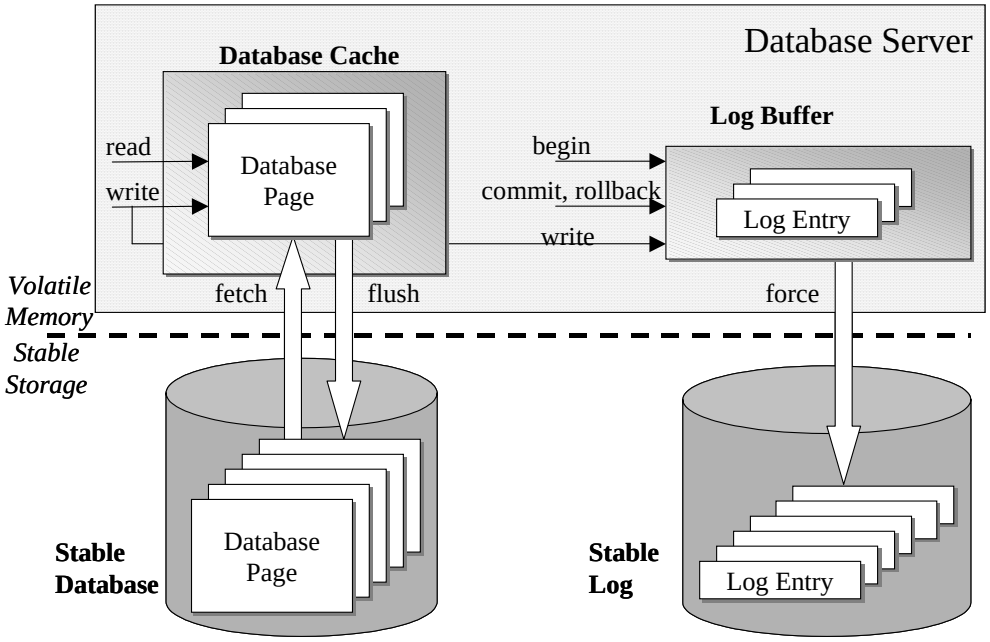
Implementierung der Crash-Recovery mittels Logging:

- Änderungen werden in Form von ***Logsätzen*** in einer Logdatei protokolliert.
  Um nicht für alle Änderung einen Plattenzugriff durchführen zu müssen, werden Logsätze zunächst in einen ***Logpuffer*** im Hauptspeicher geschrieben und nur bei Bedarf (s.u.) sequentiell in die ***Logdatei*** auf Platte geschrieben.

  Bemerkung:
  Um sicherzustellen, daß für die Logdatei tatsächlich immer sequentielle I/Os stattfinden, sollte die Logdatei auf einer separaten Platte liegen.
- Ein Logsatz enthält in der Regel sowohl die ***Undo-Information*** als auch die ***Redo-Information*** einer Änderung.
  Im einfachsten Fall bezieht sich ein Logsatz bzw. die darin protokollierte Änderung auf genau eine Seite der Datenbank. Der Logsatz beschreibt entweder den Zustand der Seite bzw. der geänderten Bytes vor und nach der Änderung (physisches Logging, zustandsorientiertes Logging) oder eine Operation auf der Byte-Ebene (physiologisches Logging, operationsorientiertes Logging auf der Byte-Ebene).
  Ein Spezialfall des zustandsorientierten Logging ist das Protokollieren ganzer Seiten jeweils vor und nach einer Änderung ("Before-Images" und "After-Images").

  Bemerkung:
  Operationsorientiertes Logging ist auch für komplexere Operationen möglich, die mehrere Seiten ändern (logisches Logging). Diese Variante ist aber schwieriger zu realisieren und daher in kommerziellen Systemen selten zu finden.
- Zusätzlich werden Logsätze für die Beendigung von Transaktionen (EOT und RBT) geschrieben, sowie optional auch für den Beginn von Transaktionen (BOT).
- Logsätze sind - über Logpuffer und Logdatei hinweg - transaktionsweise rückwärts verkettet. Dadurch können einzelne Transaktionen auch im laufenden Normalbetrieb einfach zurückgesetzt werden.
- Logsätze sind fortlaufend numeriert. Die Nummer eines Logsatzes - genannt ***"Log Sequence Number" (LSN, LogSeqNo)*** - gibt den relativen Zeitpunkt einer Änderung an. Sie kann auch - je nach Implementierung - zur Adressierung der Logsätze in der Logdatei dienen.
  Die LSN der jeweils jüngsten Änderung einer Seite wird in einem speziellen Feld ***"SeqNo" im Seiten-Header*** gespeichert.
- Der Logpuffer muß in den folgenden Fällen in die Logdatei auf Platte geschrieben werden:
  - ***Undo-Regel:*** vor dem Zurückschreiben einer geänderten Seite aus dem Puffer in die Datenbank auf der Platte, sofern die Seite Änderungen von noch nicht abgeschlossenen Transaktionen enthält (*Write Ahead Logging, WAL-Prinzip*)
  - ***Redo-Regel:*** unmittelbar vor dem Commit einer Transaktion, die Änderungen durchgeführt hat.

Architektur für das Logging im laufenden Normalbetrieb:



Vorgehen beim Warmstart nach einem Crash:

1) ***Analysephase:***
   Die Logdatei wird sequentiell gelesen (in aufsteigender LSN-Reihenfolge), und dabei wird bestimmt, welche Transaktionen vor dem Crash beendet wurden ("Gewinner") und welche unvollendet waren ("Verlierer").

2) ***Redo-Phase:***
   Die Logdatei wird nochmals sequentiell gelesen (in aufsteigender LSN-Reihenfolge), und dabei werden die Änderungen der Gewinner – oder beim sog. *Redo-History-Algorithmus* – alle Änderungen wiederholt, sofern die Änderungen noch nicht in der Datenbank enthalten sind. Dazu wird getestet, ob die LSN des Logsatzes größer (d.h. jünger) ist als die LSN im Header der betreffenden Seite; und nur wenn dies der Fall ist, wird die Änderung wiederholt, wobei die LSN im Header der Seite entsprechend erhöht wird. Dieser Test stellt die Idempotenz der Redo-Phase sicher.

3) ***Undo-Phase:***
   Die Logdatei wird nochmals sequentiell gelesen (in absteigender LSN-Reihenfolge), und dabei werden die Änderungen der Verlierer rückgängig gemacht, sofern die Änderungen in der Datenbank enthalten sind. Dazu wird getestet, ob die LSN des Logsatzes kleiner oder gleich der LSN im Header der betreffenden Seite ist (d.h. nicht jünger); und nur wenn dies der Fall ist, wird die Änderung rückgängig gemacht und die LSN im Header der Seite erniedrigt.
   Dieser Test stellt die Idempotenz der Undo-Phase sicher.

Verbesserung der Effizienz:

- *im laufenden Normalbetrieb:*
  Wegen des Persistenzprinzips muß eigentlich bei jedem Commit einer Änderungstransaktion der Logpuffer in die Logdatei auf Platte geschrieben werden. Man kann die Anzahl der Log-I/Os reduzieren, indem man das Schreiben mehrerer EOT-Logsätze zusammenfaßt. Dazu muß das Commit von Transaktionen (genauer: das Melden des SQLCODE 0 für die COMMIT-WORK-Anweisung des Anwendungsprogramms) ggf. verzögert werden, bis der Logpuffer voll ist, der Logpuffer wegen des WAL-Prinzips auf Platte geschrieben werden muß oder ein Commit bereits sehr lange verzögert wurde. Diese Technik nennt man *Group-Commit*.

- *beim Warmstart:*
  Die Analysephase und die Redo-Phase der Recovery müssen im Prinzip die gesamte Logdatei lesen, obwohl vermutlich nur der jüngere Teil der Logdatei relevant ist.
  Dies kann verbessert werden, indem man im laufenden Normalbetrieb periodisch *Sicherungspunkte (Checkpoints)* erzeugt, die dazu beitragen, den Analyse- und Redo-Aufwand beim Warmstart zu begrenzen. Da Sicherungspunkte zusätzlichen Overhead im laufenden Normalbetrieb verursachen, kann der Systemadministrator bei den meisten kommerziellen DBS die Frequenz der Sicherungspunkterzeugung einstellen (typische Werte liegen in der Größenordnung von 10 Minuten).

  Variante 1: Synchrone Sicherungspunkte
  Beim Sicherungspunkt werden alle geänderten Seiten aus dem Datenbankpuffer in die Datenbank auf Platte zurückgeschrieben. Zusätzlich wird die Tatsache, daß ein Sicherungspunkt erzeugt wurde in der Logdatei durch einen Checkpoint-Logsatz vermerkt. Die Plattenadresse (und/oder LSN) des jeweils jüngsten Checkpoint-Logsatzes wird in einem speziellen "Bootstrap-Block" mit einer festen Plattenadresse festgehalten.
  Bei einem eventuellen späteren Warmstart muß die Redo-Phase die Logdatei dann nur ab dem jüngsten Checkpoint-Logsatz lesen.

  Die Analysephase kann ebenfalls optimiert werden, indem man bei einem Sicherungspunkt in den Checkpoint-Logsatz eine Liste der zu diesem Zeitpunkt aktiven Transaktionen aufnimmt sowie für jede aktive Transaktion die LSN des jeweils letzten Logsatzes der Transaktion vor dem Sicherungspunkt. Beim Warmstart braucht die Analysephase dann ebenfalls die Logdatei nur ab dem jüngsten Checkpoint-Logsatz lesen.

  Variante 2: Asynchrone Sicherungspunkte
  Synchrone Sicherungspunkte belasten den laufenden Normalbetrieb vor allem dadurch, daß sie sehr viele Pufferseiten "impulsartig" innerhalb kurzer Zeit in die Datenbank schreiben. Bei asynchrone Sicherungspunkten werden zum Zeitpunkt des Sicherungspunktes gar keine Seiten in die Datenbank zurückgeschrieben. Stattdessen läßt man einen ständig mitlaufenden Hintergrundprozeß veränderte Pufferseiten immer dann in Datenbank zurückschreiben, wenn die entsprechende Platte gerade unbelegt ist.
  Um trotzdem den Analyse- und Redo-Aufwand beim Warmstart gering zu halten, werden bei einem asynchronen Sicherungspunkt Informationen über den aktuellen Pufferinhalt in

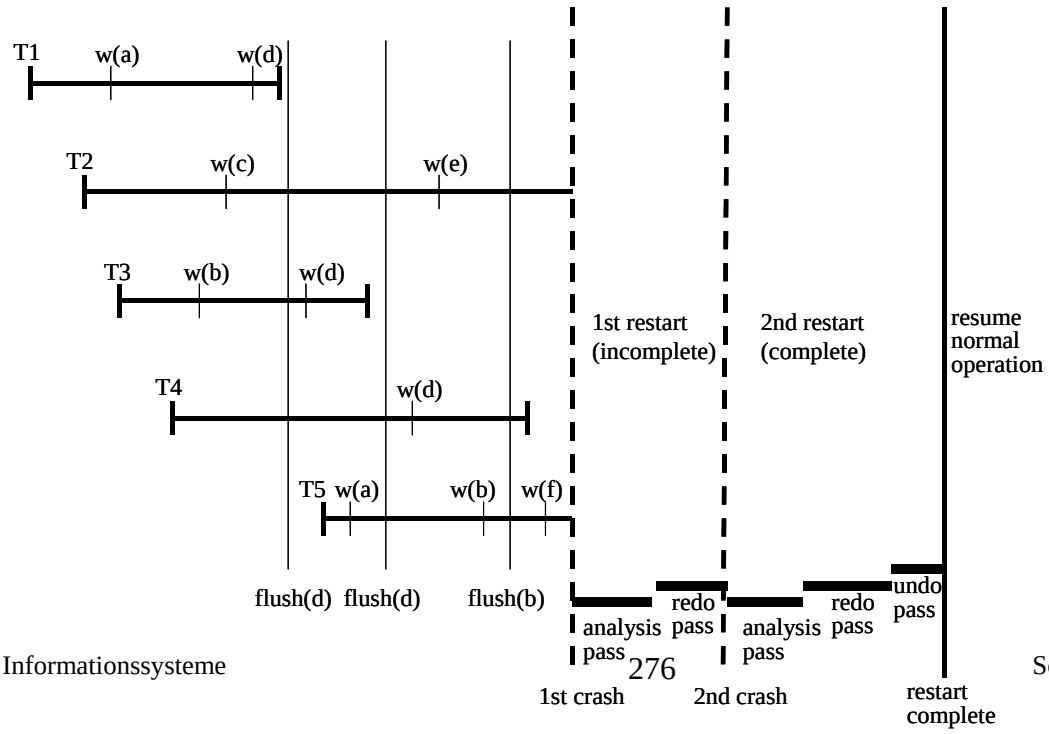
den Checkpoint-Logsatz geschrieben (zusätzlich zur Information über die aktiven Transaktionen): Für jede veränderte Pufferseite wird die LSN des ältesten Logsatzes dieser Seite seit dem letzten Zurückschreiben der Seite im Checkpoint-Logsatz eingetragen.
Beim Warmstart braucht die Redo-Phase die Logdatei dann nur ab der ältesten LSN aller entsprechenden Seiten-Einträge im jüngsten Checkpoint-Logsatz lesen.

Die Analyse-Phase kann außerdem Information über die zum Crash-Zeitpunkt „schmutzigen“ und damit potentiell Redo-bedürftigen Seiten im Cache sowie deren jeweils ältester Redo-bedürftiger Änderung rekonstruieren. Dies Information wird in der sog. Dirty-Pages-Table im Hauptspeicher zusammengestellt; sie zielt auf die Minimierung der Random I/Os auf der Datenbank während der Redo-Phase und erlaubt verschiedene Formen von I/O-Optimierungen (Batching und Prefetching und damit einhergehende Optimierungen der Plattenlatenzzeiten).
Diese Buchführungsinformation ist konservativ, enthält also auch potentiell schmutzige Seiten, die vor dem Crash vom Cache-Manager in die Datenbank zurückgeschrieben wurden. Durch zusätzliches Protokollieren dieser Flush-Aktionen des Cache-Managers kann die DirtyPagesTable-Information noch weiter verbessert werden.

Behandlung subtiler Sonderfälle:

Um das Undo für Transaktionen, die bereits vor dem Crash Verlierer waren korrekt und gleichzeitig möglichst einfach behandeln zu können, werden auch für alle Undo-Schritte selbst Logsätze geschrieben, sog. Compensation Log Entries (CLEs). Beim Redo werden diese Schritte wiederholt ; beim Undo werden sie zusammen mit den Logsätzen der dazugehörigen regulären Vorwärtsoperationen übersprungen.
Diese Redo-History-Methode deckt zum einen den Fall ab, dass Transaktionen im laufenden Betrieb zurückgesetzt wurden (z.B. als Deadlockopfer), bevor es später zu einem Systemabsturz kommt, und zum anderen den Fall, dass das System während der Crash-Recovery selbst nochmals ausfällt. Ohne CLEs wäre die korrekte Behandlung solcher Sonderfälle sehr kompliziert und potentiell ineffizient.

Beispielablauf (ohne Sicherungspunkte):



Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(T1)			1: begin(T1)	
2: begin(T2)			2: begin(T2)	
3: write(a,T1)	a: 3		3: write(a,T1)	
4: begin(T3)			4: begin(T3)	
5: begin(T4)			5: begin(T4)	
6: write(b,T3)	b: 6		6: write(b,T3)	
7: write (c,T2)	c: 7		7: write(c,T2)	
8: write(d,T1)	d: 8		8: write(d,T1)	
9: commit(T1)			9: commit(T1)	1,2,3,4,5,6,7,8,9
10: flush(d)		d:8		
11: write(d,T3)	d: 11		11: write(d,T3)	
12: begin(T5)			12: begin(T5)	
13: write(a,T5)	a: 13		13: write(a,T5)	
14: commit(T3)			14: commit(T3)	11,12,13,14
15: flush(d)		d: 11		
16: write(d,T4)	d: 16		16: write(d,T4)	
17: write(e,T2)	e: 17		17: write(e,T2)	
18: write(b,T5)	b: 18		18: write(b,T5)	
19: flush(b)		b: 18		16,17,18
20: commit(T4)			20: commit(T4)	20
21: write(f,T5)	f: 21		21: write(f,T5)	
system crash				

restart				
analysis pass: losers = {T2,T5}				
redo(3)	a: 3			
consider-redo(6)	b: 18			
flush (a)		a: 3		
consider-redo(8)	d: 11			
consider-redo(11)	d: 11			
second system crash				

second restart				
analysis pass: losers = {T2,T5}				
consider-redo(3)	a:3			
consider-redo(6)	b: 18			
consider-redo(8)	d: 11			
consider-redo(11)	d: 11			
redo(16)	d: 16			
undo(18)	b: 17			
consider-undo(17)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			
second restart complete: resume normal operation				

Ablauf für dasselbe Beispiel mit Optimierungen (asynchrone Sicherungspunkte, Flush-Logsätze):

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(T1)			1: begin(T1)	
2: begin(T2)			2: begin(T2)	
3: write(a,T1)	a: 3		3: write(a,T1)	
4: begin(T3)			4: begin(T3)	
5: begin(T4)			5: begin(T4)	
6: write(b,T3)	b: 6		6: write(b,T3)	
7: write (c,T2)	c: 7		7: write(c,T2)	
8: write(d,T1)	d: 8		8: write(d,T1)	
9: commit(T1)			9: commit(T1)	1,2,3,4,5,6,7,8,9
10: flush(d)		d:8	**10: flush(d)**	
11: write(d,T3)	d: 11		11: write(d,T3)	
12: begin(T5)			12: begin(T5)	
13: write(a,T5)	a: 13		13: write(a,T5)	
14: checkpoint			14: CP DirtyPages: {a,b,c,d} RedoLSNs: a:3, b:6, c:7, d:11 ActiveTrans: {T2,T3,T4,T5}	10,11,12,13,14
15: commit(T3)			15: commit(T3)	15
16: flush(d)		d: 11	**16: flush(d)**	
17: write(d,T4)	d: 17		17: write(d,T4)	
18: write(e,T2)	e: 18		18: write(e,T2)	
19: write(b,T5)	b: 19		19: write(b,T5)	
20: flush(b)		b: 19	**20: flush(b)**	16,17,18,19
21: commit(T4)			21: commit(T4)	20,21
22: write(f,T5)	f: 22		22: write(f,T5)	
system crash				

restart				
analysis pass: losers = {T2,T5} **DirtyPages = {a,c,d,e,f}** **RedoLSNs: a:3, c:7, d:17, e:18**				
redo(3)	a:3			
consider-redo(6)	b: 19			
skip-redo(8)				
skip-redo(11)				
redo(17)	d:17			
undo(19)	b: 18			
consider-undo(18)	e: 0			
consider-undo(13)	a: 3			
consider-undo(7)	c: 0			
restart complete: resume normal operation				

Pseudocode für optimierte Crash-Recovery:

```
/* Data structures for the page model recovery algorithm of choice: */

type Page: record of
        PageNo: identifier;
        PageSeqNo: identifier;
        Status: (clean, dirty);
        Contents: array [PageSize] of char;
     end;
persistent var StableDatabase:
        set of Page indexed by PageNo;
var DatabaseCache:
        set of Page indexed by PageNo;
type LogEntry: record of
        LogSeqNo: identifier;
        TransId: identifier;
        PageNo: identifier;
        ActionType: (write, full-write, begin, commit, rollback,
             compensate, checkpoint, flush);
        ActiveTrans: set of TransInfo;
             /* present only in log entries of type checkpoint */
        DirtyPages: set of DirtyPageInfo;
             /* present only in log entries of type checkpoint */
        UndoInfo: array of char;
        RedoInfo: array of char;
        PreviousSeqNo: identifier;
        NextUndoSeqNo: identifier;
     end;
persistent var StableLog:
        ordered set of LogEntry indexed by LogSeqNo;
var LogBuffer:
        ordered set of LogEntry indexed by LogSeqNo;
persistent var MasterRecord: record of
        StartPointer: identifier;
        LastCP: identifier;
     end;
type TransInfo: record of
        TransId: identifier;
        LastSeqNo: identifier;
     end;
var ActiveTrans:
        set of TransInfo indexed by TransId;
typeDirtyPageInfo: record of
        PageNo: identifier;
        RedoSeqNo: identifier;
     end;
var DirtyPages:
        set of DirtyPageInfo indexed by PageNo;
```

```
/* actions during normal operation */

write or full-write (pageno, transid, s):
   DatabaseCache[pageno].Contents := modified contents;
   DatabaseCache[pageno].PageSeqNo := s;
   DatabaseCache[pageno].Status := dirty;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := write or full-write;
   newlogentry.TransId := transid;
   newlogentry.PageNo := pageno;
   newlogentry.UndoInfo := information to undo update (before image for full-write);
   newlogentry.RedoInfo := information to redo update (after image for full-write);
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   ActiveTrans[transid].LastSeqNo := s;
   LogBuffer += newlogentry;
   if pageno not in DirtyPages then
      DirtyPages += pageno;
      DirtyPages[pageno].RedoSeqNo := s;
   end /*if*/;

fetch (pageno):
   DatabaseCache += pageno;
   DatabaseCache[pageno].Contents := StableDatabase[pageno].Contents;
   DatabaseCache[pageno].PageSeqNo := StableDatabase[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;

flush (pageno):
   if there is logentry in LogBuffer with logentry.PageNo = pageno
   then
        force ( );
   end /*if*/;
   StableDatabase[pageno].Contents := DatabaseCache[pageno].Contents;
   StableDatabase[pageno].PageSeqNo := DatabaseCache[pageno].PageSeqNo;
   DatabaseCache[pageno].Status := clean;
   newlogentry.LogSeqNo := next sequence number to be generated;
   newlogentry.ActionType := flush;
   newlogentry.PageNo := pageno;
   LogBuffer += newlogentry;
   DirtyPages -= pageno;

force ( ):
     StableLog += LogBuffer;
     LogBuffer := empty;

begin (transid, s):
   ActiveTrans += transid;
   ActiveTrans[transid].LastSeqNo := s;
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := begin;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := nil;
   LogBuffer += newlogentry;

commit (transid, s):
   newlogentry.LogSeqNo := s;
   newlogentry.ActionType := commit;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   LogBuffer += newlogentry;
   ActiveTrans -= transid;
   force ( );
```

```
abort (transid):
   logentry :=
         ActiveTrans[transid].LastSeqNo;
   while logentry is not nil and
         logentry.ActionType = write or full-write
   do
         newlogentry.LogSeqNo := new sequence number;
         newlogentry.ActionType := compensation;
         newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
         newlogentry.RedoInfo := inverse action of the action in logentry;
         newlogentry.NextUndoSeqNo := logentry.PreviousSeqNo;
         ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
         LogBuffer += newlogentry;
         write (logentry.PageNo) according to logentry.UndoInfo;
         logentry := logentry.PreviousSeqNo;
   end /*while*/
   newlogentry.LogSeqNo := new sequence number;
   newlogentry.ActionType := rollback;
   newlogentry.TransId := transid;
   newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
   newlogentry.NextUndoSeqNo := nil;
   LogBuffer += newlogentry;
   ActiveTrans -= transid;
   force ( );


log truncation ( ):
    OldestUndoLSN :=
         min {i | StableLog[i].TransId is in ActiveTrans};
    SystemRedoLSN := min {DirtyPages[p].RedoSeqNo};
    OldestRedoPage := page p such that
         DirtyPages[p].RedoSeqNo = SystemRedoLSN;
    NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    OldStartPointer := MasterRecord.StartPointer;
    while OldStartPointer - NewStartPointer is not sufficiently large
          and SystemRedoLSN < OldestUndoLSN
    do
        flush (OldestRedoPage);
        SystemRedoLSN := min{DatabaseCache[p].RedoLSN};
        OldestRedoPage := page p such that
              DatabaseCache[p].RedoLSN = SystemRedoLSN;
        NewStartPointer := min{OldestUndoLSN, SystemRedoLSN};
    end /*while*/;
    MasterRecord.StartPointer := NewStartPointer;

checkpoint ( ):
    logentry.ActionType := checkpoint;
    logentry.ActiveTrans := ActiveTrans (as maintained in memory);
    logentry.DirtyPages := DirtyPages (as maintained in memory);
    logentry.LogSeqNo := next sequence number to be generated;
    LogBuffer += logentry;
    force ( );
    MasterRecord.LastCP := logentry.LogSeqNo;
```

```
/* recovery algorithms */

restart ( ):
   analysis pass ( ) returns losers, DirtyPages;
   redo pass ( );
   undo pass ( );

analysis pass ( ) returns losers, DirtyPages:
   var losers: set of record
            TransId: identifier;
            LastSeqNo: identifier;
         end indexed by TransId;
   cp := MasterRecord.LastCP;
   losers := StableLog[cp].ActiveTrans;
   DirtyPages := StableLog[cp].DirtyPages;
   max := LogSeqNo of most recent log entry in StableLog;
   for i := cp to max do
        case StableLog[i].ActionType:
            begin:
                  losers += StableLog[i].TransId;
                  losers[StableLog[i].TransId].LastSeqNo := nil;
            commit:
                  losers -= StableLog[i].TransId;
            full-write:
                  losers[StableLog[i].TransId].LastSeqNo := i;
        end /*case*/;

        if StableLog[i].ActionType = write or full-write or compensate
              and StableLog[i].PageNo not in DirtyPages
        then
              DirtyPages += StableLog[i].PageNo;
              DirtyPages[StableLog[i].PageNo].RedoSeqNo := i;
        end /*if*/;
        if StableLog[i].ActionType = flush
        then
               DirtyPages -= StableLog[i].PageNo;
        end /*if*/;
   end /*for*/;

redo pass ( ):
   SystemRedoLSN := min {DirtyPages[p].RedoSeqNo};
   max := LogSeqNo of most recent log entry in StableLog;
   for i := SystemRedoLSN to max do
          if StableLog[i].ActionType = write or full-write or compensate
          then
             pageno = StableLog[i].PageNo;
             if pageno in DirtyPages and
                DirtyPages[pageno].RedoSeqNo < i
             then
                fetch (pageno);
                if DatabaseCache[pageno].PageSeqNo < i
                then
                   read and write (pageno)
                        according to StableLog[i].RedoInfo;
                   DatabaseCache[pageno].PageSeqNo := i;
                end /*if*/;
             end /*if*/;
          end /*if*/;
   end /*for*/;
```

```
undo pass ( ):
   ActiveTrans := empty;
   for each t in losers
   do
         ActiveTrans += t;
         ActiveTrans[t].LastSeqNo := losers[t].LastSeqNo;
   end /*for*/;
   while there exists t in losers
         such that losers[t].LastSeqNo <> nil
   do
        nexttrans := TransNo in losers
             such that losers[nexttrans].LastSeqNo =
             max {losers[x].LastSeqNo | x in losers};
       nextentry := losers[nexttrans].LastSeqNo;

       if StableLog[nextentry].ActionType = compensation
       then
          losers[nexttrans].LastSeqNo := StableLog[nextentry].NextUndoSeqNo;
       end /*if*/;

       if StableLog[nextentry].ActionType = write or full-write
       then
          pageno = StableLog[nextentry].PageNo;
          fetch (pageno);
          if DatabaseCache[pageno].PageSeqNo >= nextentry.LogSeqNo
          then
             newlogentry.LogSeqNo := new sequence number;
             newlogentry.ActionType := compensation;
             newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
             newlogentry.NextUndoSeqNo := nextentry.PreviousSeqNo;
             newlogentry.RedoInfo :=
                  inverse action of the action in nextentry;
             ActiveTrans[transid].LastSeqNo := newlogentry.LogSeqNo;
             LogBuffer += newlogentry;
             read and write (StableLog[nextentry].PageNo)
                  according to StableLog[nextentry].UndoInfo;
             DatabaseCache[pageno].PageSeqNo := newlogentry.LogSeqNo;
         end /*if*/;
         losers[nexttrans].LastSeqNo =
               StableLog[nextentry].PreviousSeqNo;
       end /*if*/;

       if StableLog[nextentry].ActionType = begin
       then
           newlogentry.LogSeqNo := new sequence number;
           newlogentry.ActionType := rollback;
           newlogentry.TransId := StableLog[nextentry].TransId;
           newlogentry.PreviousSeqNo := ActiveTrans[transid].LastSeqNo;
           LogBuffer += newlogentry;
           ActiveTrans -= transid;
           losers -= transid;
       end /*if*/;

   end /*while*/;
   force ( );
```

16.2 Commit-Protokolle für verteilte Transaktionen

Problem der Atomarität in einem verteilten System

Transaktionen können Daten in mehreren Systemen ändern (mehrere Datenbanken desselben DBS oder mehrere DBS). Es ist erheblich schwieriger, die Atomarität einer solchen verteilten Transaktion zu gewährleisten. Zusätzlich zu den Logging-Maßnahmen der beteiligten Systeme ist ein verteiltes Protokoll notwendig, um das Commit oder Rollback der Transaktion in allen Systemen einheitlich durchzuführen. Es darf nicht passieren, dass ein System einen Commit-Logsatz für die Transaktion schreibt, während ein anderes System seinen Teil der Transaktion zurücksetzt, beispielsweise weil dieses System abstürzt. Zusätzlich müssen verschiedene Arten von Nachrichtenfehlern (verlorene Nachrichten, duplizierte Nachrichten) verkraftet werden.

2-Phasen-Commit-Protokoll (2PC)

Eines der beteiligten Systeme wird zum *Koordinator* für den Ablauf des Commit einer Transaktion bestimmt. Es kann z.B. immer das System gewählt werden, von dem aus die Transaktion initiiert wurde. Alle Systeme, in denen die Transaktion Änderungen durchgeführt hat, werden als *Agenten* (oder Teilnehmer) bezeichnet. Das Protokoll läuft dann in zwei Phasen ab:

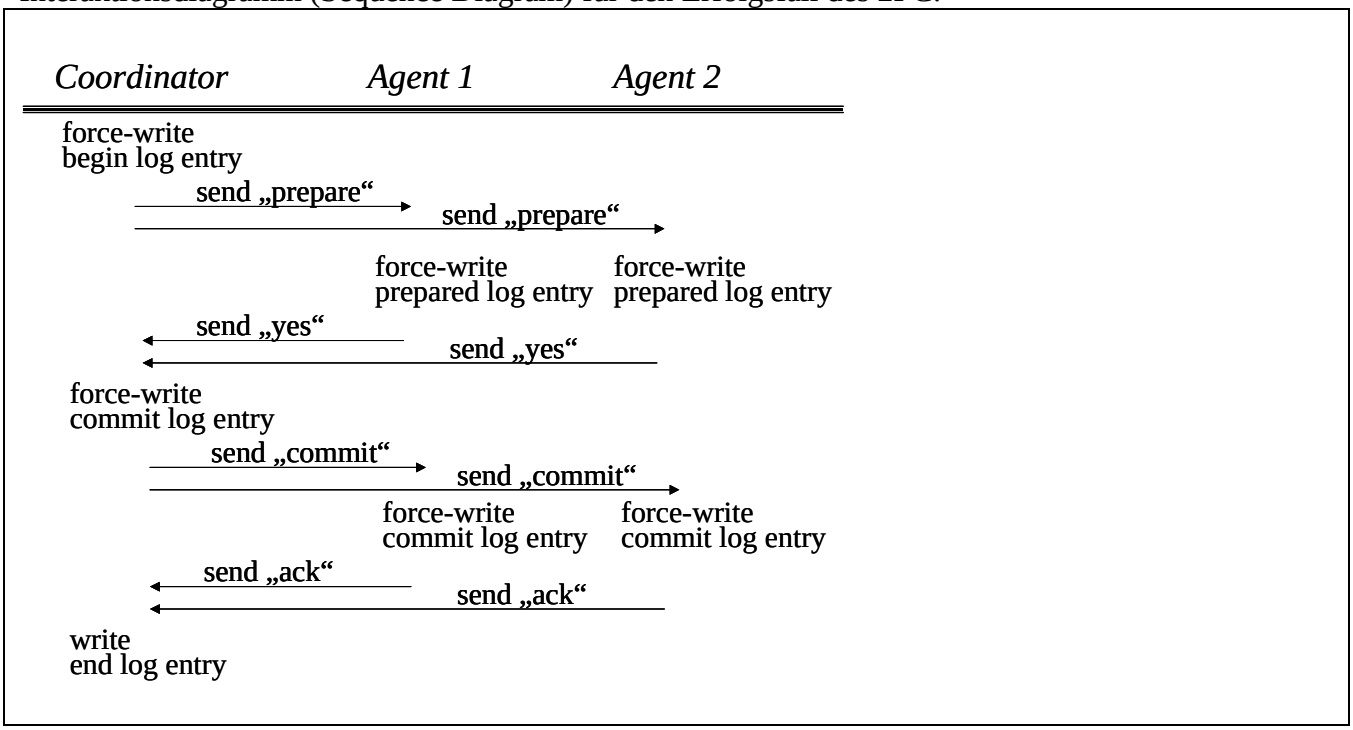
- *Phase 1:* Der Koordinator fragt alle Agenten, ob sie in der Lage sind, die Transaktion mit Commit zu beenden, und bittet die Agenten gleichzeitig, sich sowohl auf ein Commit als auch ein Rollback vorzubereiten („Prepare“-Nachricht). Im positiven Fall gehen dann alle Agenten in den „Prepared“-Zustand über.
- *Phase 2:* Aufgrund des „Umfrageergebnisses“ der ersten Phase trifft der Koordinator dann eine globale Entscheidung für die Transaktion und teilt diese allen Agenten mit. Nur wenn alle Agenten positiv geantwortet haben, kann der Koordinator eine Commit-Entscheidung treffen, andernfalls muss seine Entscheidung Rollback lauten.

In der Zeit zwischen dem Antworten eines Agenten und dem Erhalt der Koordinatorentscheidung kann ein Agent keine eigenmächtige Entscheidung treffen. Insbesondere muss der Agent weiterhin alle Sperren für die Transaktion halten, da auch ein Rollback noch möglich ist. Da der Koordinator ausfallen kann oder die Nachricht mit seiner Entscheidung verloren gehen kann, können Agenten auf diese Weise in eine gewisse Blockierungssituation geraten, die sich sehr negativ auf die Leistung des betroffenen Systems auswirken kann. Man bezeichnet aus diesem Grund das 2PC auch als ein „blockierendes“ Commit-Protokoll. Man kann beweisen, dass es – unter Berücksichtigung aller möglichen Fehlerfälle – kein nichtblockierendes verteiltes Commit-Protokoll geben kann.

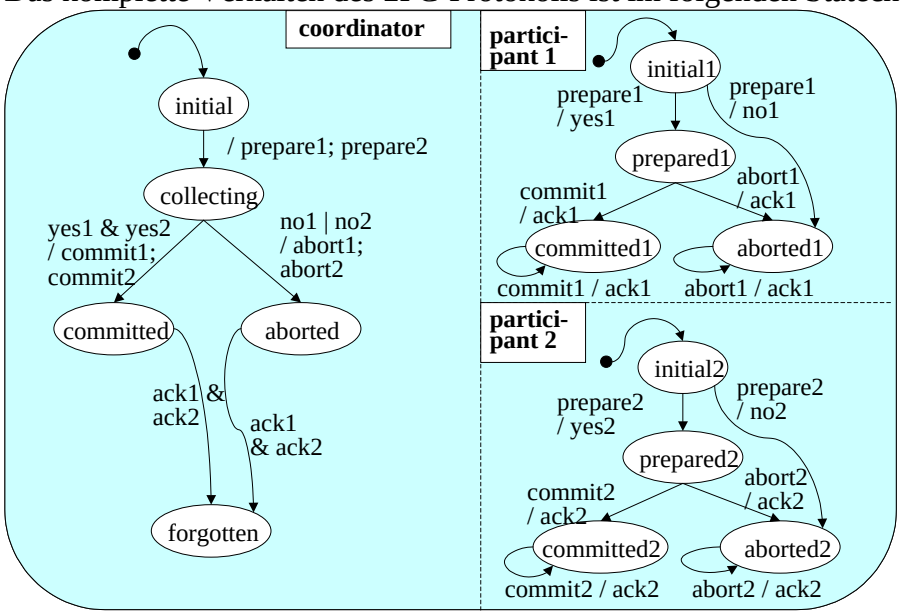
Zusatzbemerkungen:

- Es gibt optimierte Varianten des 2PC, die Nachrichten und/oder Log-I/Os in bestimmten Fällen vermeiden.
- Das 2PC ist standardisiert. Es wird von praktisch allen kommerziellen DBS unterstützt sowie von vielen TP-Monitoren (Transaction Processing Monitors) und Object-Request-Brokers (ORBs) unterstützt. TP-Monitor und ORBs sind Systemsoftware-Komponenten, die eine transaktionsorientierte Kommunikation zwischen Clients und Servers steuern; sie übernehmen dabei typischerweise die Koordinatorrolle im 2PC. ORBs koordinieren Aufrufe von Methoden auf Business-Objekten, die z.B. als sog. Enterprise Java Beans (EJB) gekapselt werden; die Unterstützung von 2PC ist dabei zur Konsistenzerhaltung essentiell.

Interaktionsdiagramm (Sequence Diagram) für den Erfolgsfall des 2PC:



Das komplette Verhalten des 2PC-Protokolls ist im folgenden Statechart spezifiziert (vgl. Kapitel 13):



Behandlung von Nachrichtenverlusten:

Falls bei einem Prozess eine – aufgrund des Protokolls – erwartete Nachricht nicht innerhalb einer bestimmten „Timeout“-Frist eintrifft, wird angenommen, dass der Sender ausgefallen ist, und der Empfänger verhält sich wie einem Knotenausfall.

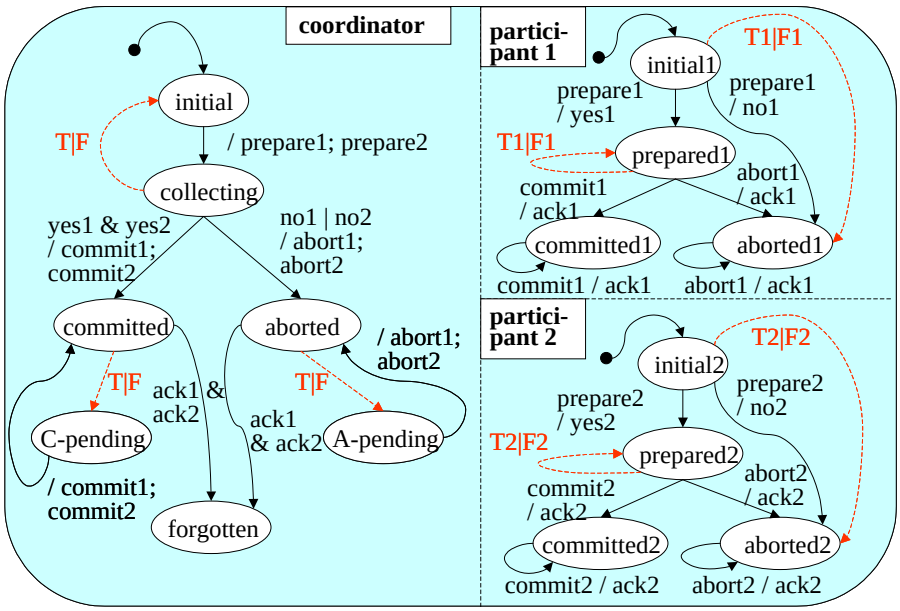
Behandlung von Knotenausfällen:

Ausfallender Prozess	*Reaktion Koordinator*	*Reaktion Agent*
Koordinator in Phase 1 (vor dem Schreiben des Commit-Logsatzes)	Nach dem Restart: Wiederholung des Sendens der Prepare-Nachricht	- Falls noch nicht prepared: Warten oder einseitiges Abort - Falls bereits prepared: Warten (Blockierung) oder ggf. andere Agenten fragen
Koordinator in Phase 2 (vor dem Schreiben des End-Logsatzes)	Nach dem Restart: Wiederholen des Sendens der Commit- oder Abort-Nachricht	- Vor dem Schreiben des Commit- oder Rollback-Logsatzes: Warten (Blockierung) oder ggf. andere Agenten fragen - Nach dem Schreiben des Commit- oder Rollback-Logsatzes: Wiederholen der Ack-Nachricht bei wiederholtem Empfang der Commit- oder Abort-Nachricht
Agent in Phase 1 (vor dem Schreiben des Prepared-Logsatzes)	Bei ausbleibendem Votum: Wiederholen des Sendens der Prepare-Nachricht	Nach dem Restart: Einseitiges Abort
Agent in Phase 2 (vor dem Schreiben des Commit- oder Rollback-Logsatzes)	Bei ausbleibendem Ack: Wiederholen des Sendens der Commit- oder Abort-Nachricht	Nach dem Restart: Warten auf Nachricht des Koordinators (wiederholte Prepare-Nachricht oder (wiederholte) Commit- oder Abort-Nachricht) oder Nachfragen beim Koordinator oder anderen Agenten

Timeouts und eigene Ausfälle können durch ein sog. *Terminationsprotokoll* als Erweiterung des 2PC systematisch in das Protokoll eingebaut werden. Dazu führt man im Statechart zwei zusätzliche Typen von Transitionen ein:

- *Timeout-Transitionen (T)* feuern, wenn ein Prozess zu lange auf eine Nachricht wartet, und
- *Failure-Transitionen (F)* feuern, wenn ein Prozess nach einem Absturz einen Warmstart durchführt und den jüngsten, durch einen Logsatz dokumentierten Zustand vor dem Crash wiederherstellt.

Das vollständige 2PC mitsamt Terminationsprotokoll ist im folgenden Statechart spezifiziert:



Weiterführende Literatur zu Kapitel 16:

- G. Weikum, G. Vossen: Transactional Information Systems – Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001
- J. Gray, A. Reuter: Transaction Processing – Concepts and Techniques, Morgan Kaufman, 1993