Datenbanken

Prof. Dr. Karl Friedrich Gebhardt

Auflage vom 10. Oktober 2019

Prof. Dr. K. F. Gebhardt

Tel: 0711-667345-11(16)(15)(12)
Fax: 0711-667345-10
email: kfg@lehre.dhbw-stuttgart.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Ein wichtiges Entwurfs-Prinzip für moderne Software ist die modulare Trennung der drei Komponenten von Software, nämlich

- Speicherung von Daten (Datenbank-Komponente, *database, DB)*
- Verarbeitung von Daten (Geschäftsprozess-Komponente, *business process, BP)*
- Benutzung der Daten (Benutzer-Schnittstelle, *user interface, UI)*

Die Wichtigkeit von Datenbanksystemen ist offensichtlich.

1.1 Datenbanksystem, DBS

Die Verwaltung und Manipulation von Daten- oder Informationssammlungen spielen in der elektronischen Datenverarbeitung eine zentrale Rolle. Ein **Datenbanksystem (DBS**, ***database system)*** ist Soft- und Hardware, mit der relativ leicht Daten langfristig gespeichert und vielen Benutzern zur Manipulation zur Verfügung gestellt werden können. Die wichtigsten Operationen eines solchen Systems sind das

- **Suchen, Finden** ***(retrieval, find, lookup, fetch)***
- **Einfügen** ***(insert)***
- **Aktualisieren, Modifizieren** ***(update, modify)***
- **Löschen** ***(delete, remove)***

Diese Operationen werden häufig durch die Akronyme CRUD *(create = insert, retrieve, update, delete)* oder – in anderer Reihenfolge – CDUR und RUDI angesprochen.

von Information. Die Informationen oder Daten können alles sein, was zur Ausübung einer Geschäftstätigkeit notwendig ist. Zwischen *Daten* und *Information* machen wir hier keine Unterschiede. In der Literatur findet man manchmal folgende Unterscheidung: Als Daten werden die reinen Werte gesehen und als Information die Bedeutung der Werte.

Nach Shannon bestimmt sich der Wert einer Information durch den Zuwachs der nach Kenntnis der Information beantwortbaren Ja/Nein-Fragen.

Das DBS verarbeitet Informationen über alle unterscheidbaren Objekte, die ein Unternehmen ausmachen, nämlich sogenannte **Entitäten** ***(entity)***. Das sind z.B. Personen, Lagerhäuser, Projekte, Projektteile, Lieferanten, Preise, Geschäftsregeln, Strategien. Mit dem DBS werden Informationen über diese Entitäten gespeichert und manipuliert.

Zwischen den Entitäten gibt es **Beziehungen** ***(relationship)***, die Entitäten miteinander verknüpfen. Z.B. arbeitet eine bestimmte Person an einem bestimmten Projekt. Diese Beziehungen gehören zu den Daten genauso wie die Werte der Daten, und müssen daher durch das DBS repräsentiert werden. Allerdings kann es manchmal sinnvoll sein, eine Beziehung als Entität zu betrachten, über die auch Informationen zu verwalten sind. Daher wird eine Beziehung oft als eine spezielle Art von Entität behandelt. Solche Beziehungs-Entitäten hängen von der Existenz der an der Beziehung beteiligten Entitäten ab.

Entitäten haben **Eigenschaften** ***(property)***. Z.B. haben Personen Namen, Vornamen, Geburtsdatum, Adresse, Größe, Gewicht und ein Elektrokardiogramm. Eigenschaften können sehr einfach sein – Name – oder eine einfache Struktur – Adresse – oder eine komplizierte Struktur haben – Elektrokardiogramm. Gängige Datenbankprodukte haben Schwierigkeiten komplizierte Strukturen zu repräsentieren, d.h. heißt insbesondere Suchen zuzulassen, die sich auf den Inhalt komplizierter Datenstrukturen beziehen. Auch Eigenschaften kann man als Entitäten betrachten, die in einer "Hat-ein"-Beziehung zur durch die Eigenschaft charakterisierten Entität stehen und von dieser Entität existentiell abhängig sind.

Die Komponenten eines DBS sind:

- Datenbank
- Datenbankmanagementsystem
- Benutzer

1.2 Datenbank, DB

Unter **Datenbank** oder **Datenbasis (DB,** ***database)*** versteht man eine Speicherung von Daten im Sekundärspeicher (Platten, Bänder, CD). Auf die Hardware–Struktur gehen wir jetzt nicht näher ein. Die DB kann aus mehreren Datenbanken bestehen, die sich u.U. an verschiedenen Orten befinden. Man spricht dann von einer **verteilten** ***(distributed)*** DB.

Eine DB hat folgende Eigenschaften:

- Die DB repräsentiert einen Aspekt der realen Welt, genannt **Miniwelt** oder ***Universe of Discourse (UoD)***.

- Die DB ist eine logisch zusammenhängende Sammlung von Daten mit inhärenter Bedeutung. Eine zufällige Ansammlung von Daten wird nicht als DB bezeichnet.
  Der Inhalt eines Papierkorbs ist keine DB. Die im Internet gespeicherte Information ist keine DB.

- Die Daten der DB werden für einen bestimmten Zweck gesammelt und bestimmten Benutzern und Anwendungsprogrammen zur Verfügung gestellt. Die DB bezieht ihre Daten aus der realen Welt und reagiert auf reale Ereignisse. Ihr Inhalt ist für – mindestens einige – Leute interessant.
  Daten, die etwa durch die Berechnung einer mathematischen Funktion entstehen, haben zwar einen logischen Zusammenhang und sind für manche Leute interessant, haben aber keinen Bezug zur realen Welt und sind damit keine DB.

Die Daten sollten **integriert** ***(integrated)*** und **von vielen benutzbar** ***(shared)*** sein.

integrated: *Alle* für die Ausübung einer Geschäftstätigkeit notwendigen Daten sind über die DB zugänglich. *Redundanzen* und *Inkonsistenzen* sind weitgehend eliminiert. D.h., wenn ein Datenelement in einer Anwendung der DB verändert wird, dann tritt es auch in allen anderen Anwendungen der DB verändert auf.

shared: Viele verschiedene Benutzer und Anwendungsprogramme haben Zugriff auf dasselbe Datenelement und können es für verschiedene Zwecke verwenden. Der Zugriff kann **gleichzeitig** ***(concurrent access)*** sein. Benutzer haben verschiedene **Sichten** ***(views)*** auf die Daten.

Wesentlich an einer DB ist, dass die Daten dauerhaft oder **persistent** gespeichert werden.

Die Eigenschaft "integriert" unterscheidet eine DB von einem Dateiverwaltungssystem, wo jeder Benutzer die für seine Anwendung benötigten Dateien definiert und mit Daten füllt *(file processing)* ohne Rücksicht darauf, ob diese Daten an anderer Stelle schon existieren oder benötigt werden.

Eine DB ist *selbstbeschreibend*. Sie enthält nicht nur die Daten, sondern auch eine vollständige Beschreibung oder Definition der Daten. Diese sogenannten **Metadaten** ***(metadata)*** sind im **Katalog** ***(catalog)*** gespeichert. Das hat den Vorteil, dass die Verwaltungssoftware (DBMS) unabhängig von der DB geschrieben werden kann. Typisch für traditionelle Datenverwaltungssysteme dagegen ist, dass die Datendefinition Teil des Anwendungsprogramms ist. Bei objektorientierten Systemen werden schließlich auch Anwendungsprogramme (Methoden, Datenoperationen) Teil der Daten.

Die Definition der Daten heißt **DB-Schema** oder **DB-Intension (intensionale Ebene)** oder Metadaten. Die eigentlichen Daten heißen **DB-Zustand** ***(DB state)*** oder **DB-Extension (extensionale Ebene)** oder **Primärdaten** oder **DB-Ausprägung**.

In folgender Tabelle sind diese Begriffe noch einmal zusammengefasst:

Eigentliche Daten	Beschreibung der Daten
Primärdaten	Metadaten
DB-Zustand	DB-Schema
DB-Extension	DB-Intension
DB-Ausprägung	Katalog

Ein **Data-Warehouse (DWH)** ist eine Ansammlung von DBs unterschiedlicher Struktur und bietet Software, die sinnvoll auf den Inhalt dieser DBs zugreift. Diese Software gehört zum **Data-Mining**, das im wesentlichen folgende Verfahren anwendet:

- künstliche neuronale Netze
- Assoziationsanalyse
- Segmentierungsverfahren mit Entscheidungsbäumen
- lineare und logistische Regression

1.3 Datenbankmanagementsystem, DBMS

Das DBMS ***(data base management system)*** ist eine Ansammlung von Programmen, mit denen die DB verwaltet werden kann:

- Definition der Daten in der DB
- Speichern von Daten in der DB
- Manipulation von Daten der DB
- Darstellung von Daten der DB

Das DBMS ist eine *wiedereintrittsfähige (nebenläufig verwendbare) (reentrant, concurrent)* Software, die im Hauptspeicher läuft und von verschiedenen DB-Aktivitäten (Transaktionen) benutzt wird. Das DBMS ist die Schnittstelle zwischen DB und Benutzer und hat dafür zu sorgen, dass

- der Zugriff effizient ist,
- Daten resistent gegen Hard– und Softwarefehler sind,
- Daten **persistent** (dauerhaft) gespeichert werden,
- Daten konsistent oder korrekt bleiben.

Die Benutzer werden durch das DBMS von den Soft- und Hardware-Details der DB abgeschirmt. In Form von Anfragesprachen bietet das DBMS verschiedene Sichten auf die Daten an. Die wichtigste Anfragesprache ist SQL *(Structured Query Language)*.

Das DBMS ist die wichtigste Software-Komponente eines DBS. Es gibt aber auch andere Software-Komponenten: Anwendungsentwicklungswerkzeuge, Berichtsgeneratoren, Dienstprogramme.

1.4 Benutzer

Wir unterscheiden drei Klassen von Benutzern ***(user)***:

- **Anwendungsprogrammierer** *(application programmers, system analysts)*: Sie schreiben in einer gängigen Hochsprache **(Hostsprache, *host language)*** (COBOL, PL/1, C, C++, Pascal, FORTRAN, Smalltalk, Ada, Java, proprietäre 4GL) Programme, die die Datenbank benutzen und dabei Daten suchen, einfügen, aktualisieren und löschen. Diese Operationen laufen alle über das DBMS.
- **Endbenutzer:** Sie verkehren mit der Datenbank über ein Anwendungsprogramm oder direkt über das DBMS. Dazu muss das DBMS eine geeignete Schnittstelle, etwa eine interaktive Anfragesprache zur Verfügung stellen. Das oben genannte SQL ist ein Beispiel für solch eine Anfragesprache. Diese Sprache erfordert gewisse informationstechnische Kenntnisse des Benutzers. Es gibt aber auch menü- oder formulargesteuerte Benutzeroberflächen für den informationstechnisch ungebildeten Benutzer.
- **Daten-Administrator und Datenbank-Administrator:** Der Daten-Administrator (DA, ***data administrator)*** trägt die zentrale Verantwortung für die Daten eines Unternehmens und fällt die strategischen Entscheidungen bezüglich der Daten. Der Datenbank-Administrator (DBA, ***database administrator)*** ist verantwortlich für die Implementation der Entscheidungen des DA.

1.5 Vorteile eines DBS

- Konsistenz der Daten, insbesondere bei mehreren Benutzern
- Datenintegrität kann überwacht werden
- Datenunabhängigkeit (siehe unten)
- Kompaktheit der Daten im Vergleich zu Papier
- Geschwindigkeit des Zugriffs
- Macht weniger Arbeit als Papieraufschriebe
- Daten sind immer aktuell
- Möglichkeit der Einstellung verschiedener Zugriffsniveaus
- Möglichkeit der zentralisierten Kontrolle der Daten

1.6 Nachteile eines DBS

- "Das System muss leben". Man benötigt EDV-Hardware und Software und begibt sich in eine starke Abhängigkeit von der Technik.
- Einstiegskosten *(Overhead)* sind relativ hoch.
- Lohnt sich nicht bei ganz einfachen oder einmaligen Anwendungen.
- Harte Echtzeitbedingungen können von einem DBMS nicht eingehalten werden.

1.7 Datenunabhängigkeit

Moderne DBS sollten und können so strukturiert werden, dass Anwendungsprogramme *ohne* Kenntnis der Datenorganisation und der Datenzugriffstechnik geschrieben werden können *(program – data independence)*.

Verschiedene Anwendungen benötigen unterschiedliche Sichten auf dieselben Daten.

DBA muss die Freiheit haben, die Speicherstruktur oder die Zugriffstechnik jederzeit zu ändern, ohne dass sich das auf das Verhalten – abgesehen von Effizienzgewinn oder -einbußen – von Anwendungsprogrammen auswirkt. Das mag notwendig sein, wenn eine neue Art von Daten hinzukommt oder wenn sich ein Standard ändert.

In objektorientierten Systemen können auch Operationen auf den Daten als Teil der DB-Definition gespeichert werden. Die **Implementation** und die **Signatur** ***(interface, signature)*** einer Operation wird mit den Daten gespeichert. Anwendungsprogramme benutzen *nur* die Signatur, nicht die Implementation der Operationen *(program – operation independence)*.

Moderne Systeme sind i.a. datenunabhängiger als alte Systeme, aber häufig nicht vollständig. Datenunabhängigkeit oder auch **Datenabstraktion** ***(data abstraction)*** sind relative Begriffe.

1.8 Relationale und andere Systeme

Fast alle seit 1980 entwickelten DBS basieren auf dem **relationalen** Modell. Daher werden wir uns zunächst nur mit diesem Modell beschäftigen.

Hier seien kurz die wichtigsten Züge des relationalen Modells genannt.

- Daten werden als **Tabellen** und *nur* als Tabellen gesehen.
- Die Datenbankoperationen haben als Ergebnis immer nur Tabellen.

Die Systeme heißen **relational**, weil **Relation** der mathematische Begriff für Tabelle ist. Auf die unterschiedliche Bedeutung von Relation und Tabelle gehen wir später ein.

Benutzer von *nicht*–relationalen Systemen sehen die Daten u.U. anders strukturiert, etwa als Baum bei einem *hierarchischen* System. Entsprechend werden von solchen Systemen Operationen angeboten, mit denen man Bäume durchlaufen und manipulieren kann.

Die folgende Liste enthält die gängigen Datenbank-Kategorien und Beispiele kommerziell erhältlicher Produkte:

File-Systeme: ISAM, VSAM

Invertierte Liste: CA_DATACOMB/DB (Computer Associates International Inc.)

Hierarchisch: IMS (IBM), System 2000
Das hierarchische Modell repräsentiert Daten als hierarchische Baumstrukturen. Jede Hierarchie repräsentiert eine Anzahl in Beziehung stehender Datensätze.

Network (CODASYL): CA-IDMS/DB (Computer Associates International Inc.), IDS, TOTAL, ADABAS
Das Netzwerkmodell repräsentiert Daten als Datensatz-Typen und One-to-Many-Beziehungen als Mengentypen. Diagrammatisch werden Datensatztypen als Rechtecke und die Beziehungen als Pfeile von der One-Entität zur Many-Entität dargestellt. Dieses Modell ist auch bekannt als CODASYL DBTG Model (Computer Data Systems Language Data Base Task Group).

Relational: DB2 (IBM), Rdb/VMS (Digital Equipment), ORACLE (Oracle), INGRES (???), INFORMIX (???), Sybase (Sybase?) und viele andere

Postrelational:

- Deductive DBS:

  Es können Inferenzregeln zur Deduktion oder Inferenz von Information aus den Daten der DB gespeichert und angewendet werden. In einer DB zur Verwaltung von Studenten könnten Regeln gespeichert werden, die bestimmen, wann ein Student durchgefallen ist. Damit ist eine Abfrage nach allen durchgefallenen Studenten möglich. Das könnte natürlich auch ein Stück Programmcode sein. Aber wenn sich die Regeln ändern, dann ist es i.a. schwerer, Programmcode zu ändern, als die Deklaration einer Inferenzregel.
- Expert DBS
- Extendable DBS (POSTGRES)
- Semantic DBS
- Universal relation DBS
- Objekt-relationale DBS
- Objekt-orientierte DBS
- Navigational DBS (Hypercard- oder Hypertext-Systeme)
- XML-DBS
- Multimedia DBS:
  Diese Systeme eignen sich besonders zur Verwaltung großer Bild-, Audio- und Videodaten.
- Geographische DBS:
  Diese Systeme eignen sich besonders zur Verwaltung geographischer Daten.

1.9 Aufbau des Buches

Das Kapitel "Architektur" zeigt die ideale Struktur eines DBS.

Danach folgen gleich drei Kapitel über den Entwurf von DBS mit dem Entity-Relationship-Modell als wichtigste Methode des Datenbank- und modernen Software-Entwurfs.

Es folgen zwei Kapitel über Speicherstrukturen und Zugriffsstrukturen.

Die folgenden Kapitel bis einschließlich Kapitel "Normalformen" beschreiben Theorie und Eigenschaften von relationalen DBS. Dabei werden wir auch auf SQL2 eingehen.

Danach gehen wir auf den "relationalen Entwurf" ein.

Es folgen objekt-orientierte, objekt-relationale DBS und einige vor-relationale DBS.

Die weiteren Kapitel behandeln die Topics Transaktion, Wiederherstellung, Sicherheit, Integrität, View, Optimierung behandelt.

Das DBS POET dient als Beispiel für ein ooDBS.

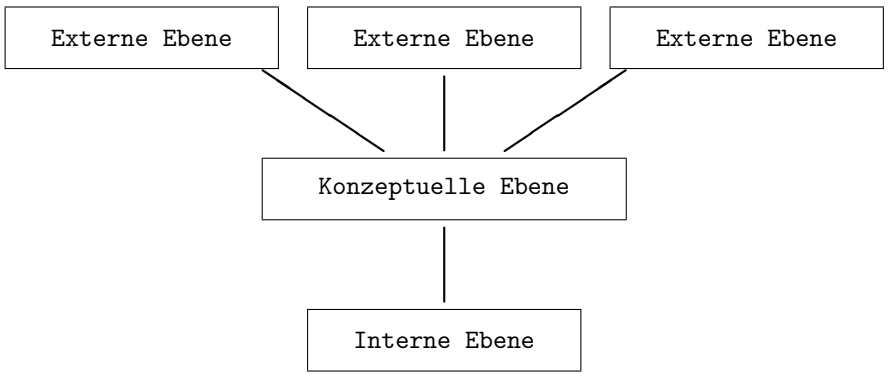
Kapitel 2

Architektur

Die **Architektur** ***(architecture)*** eines DBS ist ein Schema zur Beschreibung von DBS und besteht aus einem oder mehreren Datenmodellen. Ein **Datenmodell** ***(data model)*** ist ein Konzept, mit dem die **Struktur** einer DB beschrieben werden kann. Zur Struktur einer DB gehören die Datentypen, Beziehungen zwischen Daten und Einschränkungen *(constraints)* der Daten. Ferner gehören dazu grundlegende Such- und Aktualisierungs-Operationen, Verhalten von Daten und auch vom Benutzer definierte Operationen.

Wir werden hier ein Schema zur Beschreibung von DBS vorstellen, in das die meisten DBS "hineinpassen" und das weitgehend mit den Empfehlungen von ANSI/SPARC (genauer: ANSI/X3/SPARC, noch genauer: ANSI/X3/SPARC Study Group on Data Base Management Systems, wobei ANSI = "American National Standards Committee on Computers and Information Processing" und SPARC = "Standards Planning and Requirements Committee") übereinstimmt.

Nach dem ANSI/SPARC–Modell sind bei einem DBS drei Ebenen zu unterscheiden:



Interne Ebene ***(internal level):*** Sie beschreibt Datenstrukturen, wie sie physisch gespeichert werden. Definiert Zugriffspfade.

Externe Ebene ***(external level):*** Sie beschäftigt sich mit der Frage, wie einzelne Benutzer die Daten sehen und auf sie zugreifen. Daher gibt es i.a. viele verschiedene externe Ebenen. Eine externe Ebene kann auch auf anderen externen Ebenen aufsetzen.

Konzeptuelle Ebene ***(conceptual level):*** Zeitinvariante, globale Struktur der DB, Schema des Datenmodells ohne Details der Implementierung oder einer Anwendung. Sie ist Schnittstelle zwischen interner und externer Ebene. Die interne Ebene muss die konzeptuelle Ebene implementieren. Die externe Ebene verwendet nur die konzeptuelle, nicht die interne Ebene, um Anwendungen der DB zu schreiben.

Abbildungen ***(mappings):*** Die die Ebenen verbindenden Striche sind Abbildungen, die die Korrespondenz zwischen den DB–Objekten in den verschiedenen Ebenen zeigen.

Beispiel: Angestelltendatenbank

Auf der konzeptuellen Ebene enthält die Datenbank Informationen über Entitäten vom Typ `Angestellter` mit den Eigenschaften `Name`, `Nummer`, `Abteilung` und `Gehalt`. Schematisch stellen wir das folgendermaßen dar:

```
Angestellter
   Name
   Nummer
   Abteilung
   Gehalt
```

Auf der internen Ebene wird ein `Angestellter` repräsentiert durch einen C–String, der als erstes den Namen enthält, dann jeweils durch ein `'\t'` getrennt Nummer, Abteilung und Gehalt (Amerikanische Notation mit Dezimalpunkt, Dezimalstellen nur soviele, wie benötigt werden). Als Zeichensatz wird ISO-8859-1 verwendet. Gespeichert werden die Angestellten in einem ASCII-File zeilenweise, d.h. durch `'\n'` getrennt.

Ein Datenfile sieht also etwa folgendermaßen aus:

```
Depardieux\t3264\tKlassiker\t4725.8\n
Belmondo\t3914\tKomödie\t5207.26\n
Delon\t372\tThriller\t7203\n
Bronson\t11122\tWestern\t6134.32\n
```

Als Beispiel betrachten wir zwei externe Ebenen:

- Anwendung der DB für die Gehaltsabrechnung: Hier werden die Angestellten in einer Tabelle benötigt, die die Angestelltennummer und das Gehalt auf zwei Dezimalstellen (Deutsche Punkt- und Kommanotation) geordnet nach den Angestelltennummern angibt.

Kennung	Lohn
372	7.203,00
3264	4.725,80
3914	5.207,26
11122	6.134,32

- Anwendung für den Direktor: Der Direktor wünscht eine Tabelle mit Name, Abteilung und Jahresgehalt als Kosten auf ganze Tausend DM aufgerundet, geordnet nach Gehalt.

Name	Abteilung	Kosten
Depardieux	Klassiker	57
Belmondo	Komödie	63
Bronson	Western	74
Delon	Thriller	87

Bemerkung: Die Namen für die verschiedenen Eigenschaften sind auf verschiedenen Ebenen durchaus unterschiedlich.

Die wichtigste Leistung der Drei-Ebenen-Architektur ist **logische** und **physische** Datenunabhängigkeit:

- Logische Datenunabhängigkeit gestattet es, die konzeptuelle Ebene zu ändern *ohne* die externen Ebenen ebenfalls ändern zu müssen (Transparenz für die Benutzer der externen Ebenen).
- Physische Datenunabhängigkeit gestattet es, die interne Ebene zu ändern *ohne* Änderung der konzeptuellen und damit auch externen Ebene (Transparenz für die Benutzer der konzeptuellen Ebene, insbesondere für Programmierer von externen Ebenen).
- Die Abbildungen zwischen den Ebenen werden allerdings durch solche Änderungen berührt.

Es folgen begriffliche Einzelheiten zu den drei Ebenen.

2.1 Konzeptuelle Ebene

Die **konzeptuelle Ebene** heißt auch **logische Ebene** und legt fest, welche Daten gespeichert werden sollen. Die **konzeptuelle Sicht** ***(conceptual view)*** ist eine Repräsentation des gesamten Informationsgehalts der DB. Diese Repräsentation ist meistens noch stark abstrahiert von den physisch gespeicherten Daten. Sie ist eine Ansammlung von **konzeptuellen Datensätzen** ***(conceptual record)***. Die konzeptuelle Sicht wird mit Hilfe des **konzeptuellen oder logischen Schemas** ***(conceptual schema)*** definiert, das die Definition der verschiedenen konzeptuellen Datensätze enthält.

Das konzeptuelle Schema wird *datenunabhängig* definiert. D.h. es gibt keinen Bezug auf die Art der Speicherung, Indizierung, Verzeigerung der Daten. Wenn das gut gelingt, dann werden die externen Schemata erst recht datenunabhängig sein.

Das konzeptuelle Schema enthält auch Zugriffsrechte und Integritätsregeln. Manchmal enthält das konzeptuelle Schema auch Informationen darüber, wie die Daten zu verwenden sind, von wo sie wohin fließen, welche Überwachungsmechanismen es gibt.

Das verwendete Datenmodell ist ein sogenanntes ***high-level*** oder konzeptuelles Datenmodell.

Als mögliche Datenmodelle sind hier zu nennen das hierarchische, das relationale, das objektorientierte Datenmodell und das Netzwerkdatenmodell. Objekt-orientierte Datenmodelle sind meistens sehr nahe am Entity-Relationship-Modell, das ein idealer Vertreter eines konzeptuellen Modells wäre. (Das ist in der Praxis allerdings nirgends realisiert.)

2.2 Interne Ebene

Die **interne Ebene** heißt auch **physische Ebene** und legt fest, wie die Daten – i.a. auf einem *Hintergrundspeicher* – gespeichert werden sollen. Die **interne Sicht** ***(internal view)*** ist eine Repräsentation der gesamten DB, die aus **internen Datensätzen** ***(internal record)*** besteht. Interne oder "gespeicherte" Datensätze sind die Datensätze, die tatsächlich gespeichert werden. Die interne Sicht ist insofern noch von der Hardware abgehoben, als ein unendlich großer linearer Adressraum angenommen wird. Einheiten für I/O-Transfer zwischen Hauptspeicher und Sekundärspeicher – Blöcke oder Seiten *(block, page)* – und Spur– oder Plattengrößen werden nicht berücksichtigt.

Die interne Sicht beruht auf dem **internen Schema (Speicherstrukturdefinition)** ***(internal schema, storage structure definition)***, das die gespeicherten Datensatztypen, Indexschemata, Speicherreihenfolgen und dergleichen definiert.

Das Datenmodell der internen Ebene heißt auch **Repräsentations- oder Implementationsmodell** ***(representational or implementation data model)***.

Obwohl diese Modelle nicht alle Einzelheiten der Speicherhierarchie zeigen, können sie doch in einer direkten Weise implementiert werden.

Wenn die interne Ebene zu weit von der Hardware bzw Betriebssystem-Software weg ist, dann wird manchmal noch ein **physisches** Datenmodell ***(physical, low-level data model)*** definiert. Es beschreibt Datensatzformate, Ordnung von Datensätzen und Zugriffspfade.

In der Praxis fällt es einem oft schwer die interne und konzeptuelle Ebene auseinanderzuhalten, was daran liegt, dass die kommerziellen DBS dem DBA die Beschäftigung mit der internen Ebene weitgehend abnehmen. Deutlich sieht man den Unterschied, wenn etwa ein ooDBS zur Speicherung der Daten ein relationales DBS verwendet.

2.3 Externe Ebene

Die **externe Ebene** oder auch **Sicht** ist die Ebene des einzelnen Benutzers. Es gibt hier den Begriff des **externen Schemas** ***(external schema)***, auf dem die **externe Sicht** ***(external view)*** beruht.

Eine externe Sicht ist ein für einen Endbenutzer speziell aufbereiteter Ausschnitt der DB. Die externe Sicht besteht häufig aus **externen Datensätzen** ***(external record)***, die i.a. nicht den gespeicherten Datensätzen gleichen. Man spricht manchmal auch von **logischen Datensätzen** ***(logical record)***, was wir aber vermeiden werden.

Externe Schemata können hierarchisch aufgebaut sein. D.h. ein externes Schema kann sich von einem oder mehreren externen Schemata ableiten.

2.4 Sprachen

Jedem Benutzer steht seine Sprache zur Verfügung.

Ein Teil dieser Sprachen betrifft DB-Objekte und DB-Operationen und heißt **DSL** ***(data sub-language)***. Er ist in die Hostsprache **eingebettet** ***(embedded)***. Ein DBS kann mehrere Hostsprachen und DSLs unterstützen. Die von den meisten Systemen unterstützte DSL ist SQL.

Jenachdem wie gut die DSL in die Hostsprache eingebettet ist, spricht man von einer **engen** ***(tightly coupled)*** oder **losen** ***(loosely coupled)*** **Kopplung** von DSL und Hostsprache.

Die DSL hat gewöhnlich zwei Komponenten:

1. **Datendefinitionssprache** ***(data definition language, DDL)***, mit der DB-Objekte definiert oder deklariert werden können. Mit der DDL werden die Schemata geschrieben, auf denen die Sichten beruhen. Außerdem werden mit der DDL Integritätsregeln oder Einschränkungen für die Daten definiert. (Manchmal wird der letztere Teil der DDL als eigene Komponente CDL *(constraint definition language)* gesehen.)

2. **Datenmanipulationssprache** ***(data manipulation language, DML)***, mit der die DB-Objekte durch den Anwendungsprogrammierer oder Endbenutzer manipuliert werden können. Diese Manipulationen transferieren Information von und zur DB.

Im Prinzip ist auf jeder Ebene eine andere Sprache möglich. Die meisten DBMS bieten aber häufig nur eine Sprache, nämlich SQL an. Das ooDBMS POET hat als DDL C++ und Java.

Bezogen auf die interne Ebene heißt die DDL auch SDL ***(storage definition language)***; bezogen auf die externe Ebene auch VDL ***(view definition language)***; DDL bleibt dann für die konzeptuelle Ebene reserviert. SQL kann DDL, SDL und VDL sein, wobei der Trend dahin geht, SDL-Anteile aus der Sprachdefinition zu eliminieren.

Bei den DMLs kann man zwei Typen unterscheiden:

1. ***high-level*** oder **deklarative** oder
   nicht-prozedurale oder ***set-at-a-time*** oder
   Mengen-orientierte Sprachen, die es erlauben DB-Operationen in knapper Form interaktiv oder embedded auszudrücken. Neben SQL gehören dazu auch moderne bequeme Oberflächen, wie sie etwa MS-Access bietet.

2. ***low-level*** oder **prozedurale** oder
   record-at-a-time oder **Datensatz-orientierte** Sprachen, die in einer Hostsprache eingebettet sein müssen. Hier werden einzelne Datensätze ermittelt und bearbeitet.

2.5 Benutzerfreundliche Oberflächen

Benutzerfreundliche Oberflächen gehören auch zu den Sprachen. Trotzdem widmen wir ihnen einen eigenen Abschnitt. Man unterscheidet:

Menübasierte Oberfläche: Der Benutzer muss sich keine Kommandos merken. Ein Angebot von Optionen (Menü) führt den Benutzer bei der Formulierung einer Anfrage. Beliebt sind Zeilen- und Pulldown-Menüs und die Verwendung von Funktionstasten für naive Benutzer.

Graphische Oberfläche: Teile des DB-Schemas werden als Diagramm dargestellt. Der Benutzer kann Anfragen formulieren, indem er das Diagramm manipuliert. Kombiniert wird das i.a. mit Menüs.

Formularbasierte Oberfläche: Dem Benutzer wird ein Formular angeboten, das er auszufüllen hat, um entweder diese Daten zu speichern oder zu suchen. Formular-Spezifikations-Sprachen ermöglichen die leichte Entwicklung solcher formular-basierter Oberflächen für den naiven Benutzer zur Durchführung von Standard-Transaktionen *(canned transaction)*.

Natürlichsprachliche Oberfläche: Diese Oberflächen akzeptieren Anfragen, die in Englisch oder anderen Sprachen formuliert werden können. Dialoge helfen bei der Klärung von Missverständnissen.

In der Praxis sind alle möglichen Kombinationen dieser Einteilung zu finden.

2.6 Datenbankadministrator — DBA

Daten-Administrator: In jedem Unternehmen, das eine Datenbank verwendet, sollte es eine Person geben, die die zentrale Verantwortung für die Daten hat. Das ist der Daten-Administrator (DA). Er kennt die Bedeutung der Daten für das Unternehmen. Er entscheidet, welche Daten zu speichern sind, und legt Regeln und Grundsätze fest, nach denen die Daten behandelt werden sollen. Insbesondere entscheidet er, wer welche Operationen auf der Datenbank ausführen darf. Der DA ist typischerweise ein Manager in gehobener Stellung, der nicht unbedingt mit den technischen Details eines DBS vertraut sein muss.

Datenbank-Administrator: Der Datenbank-Administrator (DBA) ist eine informationstechnisch ausgebildete Person, die dafür verantwortlich ist, dass die Entscheidungen des DA implementiert werden. Er richtet die Datenbank ein, vergibt die Zugriffsrechte nach den Richtlinien des DA. Er ist auch für die Leistungsfähigkeit des Systems verantwortlich. Dazu mag es erforderlich sein, eine Gruppe von Systemprogrammierern zu leiten.

Die Aufgaben des DBA sind im einzelnen:

- Definition des konzeptuellen Schemas: Der DA bestimmt, welche Daten gespeichert werden. D.h. er benennt die für das Unternehmen relevanten Entitäten, deren Eigenschaften und Beziehungen. Dieser Prozess heißt *logisches* oder *konzeptuelles* Datenbankdesign *(logical or conceptual database design)* (Beispiel E/R-Diagramm). Auf Grund dieses Designs wird der DBA unter Verwendung der konzeptuellen DDL das konzeptuelle Schema erstellen. Das DBMS benutzt dieses Schema, um auf DB-Anfragen zu reagieren.
- Definition des internen Schemas: Der DBA entscheidet, wie die Daten in der gespeicherten DB repräsentiert werden sollen. Dieser Prozess heißt *physisches* Datenbankdesign *(physical database design)*. Auf Grund dieses Designs erstellt der DBA das interne Schema unter Verwendung der internen DDL. Ferner muss er die Abbildung konzeptuelles↔internes Schema definieren, wozu ihm entweder die konzeptuelle oder die interne DDL Möglichkeiten bietet. Die beiden Aufgaben – Erstellung von Schema und Abbildung – wollen wir hier klar unterscheiden, obwohl sie in der Praxis häufig ineinanderlaufen, (weil etwa die gleiche DDL verwendet wird,) was aber zu Wartungsproblemen führen kann.

- Betreuung der Benutzer: Der DBA sorgt dafür, dass einem Benutzer die Daten zur Verfügung stehen, die er benötigt. Dazu schreibt er oder hilft er schreiben die externen Schemata unter Verwendung der zur Verfügung stehenden externen DDLs. Wieder muss auch die Abbildung externes↔konzeptuelles Schema definiert werden, ein Prozess, der aus Wartungsgründen getrennt von der Schemaerstellung zu sehen ist.
- Definition von Zugriffsrechten und Integritätsregeln: Obwohl das schon zum konzeptuellen Design gehört, wird dieser Punkt wegen seiner Wichtigkeit extra erwähnt.
- Definition von Datensicherungsmechanismen *(backup* und *recovery)*: Eine DB kann durch menschliches Versagen (insbesondere des DBA), durch Software- und Hardwarefehler beschädigt werden. Durch periodisches Entladen oder Dumpen *(unloading* oder *dumping)* der DB kann der Schaden in Grenzen gehalten werden. Dazu müssen Strategien definiert und implementiert werden.
- Überwachung der Systemleistung: DBA hat dafür zu sorgen, dass z.B. die Antwortzeiten akzeptabel bleiben. Das kann entweder Hardware- oder Software-Updates erfordern oder die Reorganisation der gespeicherten DB notwendig machen. D.h. während sich das konzeptuelle Schema nicht ändert, können sich durchaus das interne Schema und die damit verbundenen Abbildungen ändern.
- Reaktion auf sich ändernde Anforderungen an die DB: Hier kann es auch mal zu einem Eingriff in das konzeptuelle Schema kommen. Der DBA muss dafür sorgen, dass alle Abbildungen, die damit zusammenhängen, nachgezogen werden.

2.7 Datenbankmanagementsystem — DBMS

Alle Zugriffe auf die DB gehen über das DBMS. Dabei sind folgende Schritte typisch:

1. Ein Benutzer formuliert eine Anforderung an die DB unter Verwendung einer DSL, typischerweise SQL.
2. Das DBMS fängt die Anforderung ab und analysiert sie. Dabei verwendet das DBMS das externe Schema des betreffenden Benutzers, die entsprechende Abbildung extern↔konzeptuell, das konzeptuelle Schema, die Abbildung konzeptuell↔intern und das interne Schema.
3. Das DBMS führt die notwendigen Operationen an der gespeicherten DB durch.

Bei einer Suchoperation z.B. muss das DBMS aus den zugehörigen gespeicherten Datensätzen die konzeptuellen Datensätze konstruieren und aus diesen die externen Datensätze, wobei auf jeder Stufe u.U. Datenkonvertierungen notwendig sind.

Im allgemeinen ist solch ein Prozess *interpretativ* mit geringer Performanz. Allerdings bieten manche DBS die *Compilation* von Zugriffen an.

Das DBMS muss folgende Funktionen unterstützen:

Datendefinition: Das DBMS muss in der Lage sein, Datendefinitionen, d.h. externe, konzeptuelle, interne Schemata im Quellcode zu akzeptieren und in einen geeigneten Objektcode zu übersetzen. Das DBMS hat Compiler für die verschiedenen DDLs.

Datenmanipulation: Zum Verarbeiten von DMLs muss das DBMS entsprechende Komponenten haben. DML-Anforderungen können *geplant* oder *ungeplant* sein.

geplant: Die Anforderung ist lange vorher bekannt, so dass der DBA die Speicherstruktur daran anpassen kann (operative oder Produktionsanwendungen, *compiled* oder *canned transaction).*

ungeplant: Das ist eine ad-hoc-Anforderung (Anwendungen der Entscheidungsfindung, *decision support).* Aufgabe des DBMS ist es, auch solche Anforderungen mit hoher Performanz zu erfüllen.

Datensicherheit und –integrität: Alle Anforderungen, die die Sicherheit, Zugriffsrechte und Integrität der Daten verletzen, werden vom DBMS zurückgewiesen.

Transaktionsmanager *(transaction manager)***:** Das DBMS muss parallele Zugriffe auf die DB so synchronisieren, dass die DB nicht inkonsistent wird.

Datenlexikon *(data dictionary)***:** Mit dem DBMS muss ein Datenlexikon oder **Datenkatalog** angelegt werden können, das selbst eine DB ist und Daten "über" die Daten enthält (Metadaten). Diese MetaDB enthält Informationen über die Schemata und Abbildungen der DB und sich selbst. Andere Bezeichnungen für die MetaDB sind ***directory, catalog, data repository, data encyclopedia***.

Man kann zusammenfassen, dass das DBMS die **Benutzeroberfläche** *(user interface)* der DB liefern muss.

Das DBMS ist kein Filemanager. Das Filemanagementsystem ist Teil des ganzen DBS und unterscheidet sich wesentlich vom DBMS:

- Es kreiert und löscht Files.
- Der Filemanager weiß nichts über die interne Datensatzstruktur und kann daher keine Anforderungen bearbeiten, die eine Kenntnis der Datensatzstruktur voraussetzen.
- Zugriffsrechte und Integritätsregeln werden typischerweise nicht unterstützt.
- Parallele Zugriffe werden typischerweise nicht unterstützt.

2.8 Datenkommunikationsmanager

DB-Anforderungen eines Endbenutzers werden häufig von dessen Workstation zum DBMS übertragen, das auf einer anderen Maschine läuft. Die Antworten des DBMS werden zurückübertragen. Die Übertragung geschieht in Form von **Botschaften** ***(communication messages)***. Der **Datenkommunikationsmanager** ***(data communications manager, DC manager)*** ist die dafür verantwortliche Software-Komponente und fungiert als autonomes System neben dem DBMS. Manchmal werden diese beiden Systeme als *eine* Software-Komponente, das DB/DC–System ***(database/data-communications system)*** gesehen.

Da Kommunikation ein eigenes Gebiet ist, werden wir hier nicht weiter darauf eingehen.

2.9 Client/Server–Architektur

Bisher hatten wir die ANSI/SPARC–Architektur eines DBS besprochen. Nun wollen wir den etwas übergeordneten Standpunkt einer Client/Server–Architektur einnehmen. Aus dieser Sicht hat ein DBS eine einfache zweiteilige Struktur, die aus dem *Server* (oder auch *Backend)* und mehreren *Clients* (oder auch *Frontends)* besteht:

- Der Server ist das DBMS selbst.
- Clients sind die verschiedenen Anwendungen, die das DBMS benutzen.

Wegen dieser klaren Zweiteilung in Client und Server liegt es nahe, diese beiden Komponenten auf verschiedenen Maschinen laufen zu lassen *(verteilte Abarbeitung*, *distributed processing)*. Das ist so attraktiv, dass der Begriff Client/Server eng mit verteilten Systemen verknüpft wird, was aber nicht korrekt ist.

2.10 Anwendungsprogramme

An dieser Stelle wollen wir die Gelegenheit nehmen, kurz die Anwendungen zu nennen.

Wir unterscheiden

- Anwendungen, die vom Benutzer geschrieben werden, wobei eine geeignete DSL verwendet wird.
- Anwendungen, die der DBS-Lieferant zur Verfügung stellt, sogenannte **Werkzeuge** oder **Tools**. Tools sind dazu da, die Erstellung und Benutzung anderer Anwendungen zu erleichtern. D.h. sie sind spezialisierte Anwendungen, die meist nur für eine Aufgabe besonders gut geeignet sind. Einige Beispiele:
  - Interpreter/Compiler für Zugriffssprachen *(query language processors)*
  - Interpreter/Compiler für natürliche Sprachen *(natural language processors)*
  - Berichtsgeneratoren *(report writers)*
  - Graphische Teilsysteme *(graphics subsystems)*
  - Tabellenkalkulation *(spreadsheets)*
  - Statistische Pakete *(statistical packages)*
  - Anwendungsgeneratoren *(application generators, 4GL processors)*
  - CASE–Produkte

2.11 Dienstprogramme

Dienstprogramme *(utilities)* sind Programme, die dem DBA bei seinen Verwaltungsaufgaben helfen. Diese Programme können auf der externen oder internen Ebene arbeiten. Einige Beispiele:

- Ladeprogramme *(load, reload)*, die Teile der DB (auch ganze DB) aus einem backup-Medium rekonstruieren oder eine Anfangsversion der DB aus nicht-DB-Dateien kreieren.
- Entladeprogramme *(unload, dump)*, die Teile der DB (auch ganze DB) für backup-Zwecke auslagern oder auf nicht-DB-Dateien schreiben.
- Reorganisationsprogramme *(reorganization)*, mit denen interne Datenstrukturen der DB verändert, insbesondere an Leistungsanforderungen angepasst werden können.
- Statistische Programme, die statistische Kenngrößen der DB (Filegrößen, Zugriffshäufigkeiten) ermitteln und analysieren.

2.12 Verteilte Abarbeitung

Verteilte Abarbeitung *(distributed processing)* bedeutet, dass verschiedene Maschinen über ein Netzwerk so gekoppelt werden können, dass sie eine Datenverarbeitungsaufgabe gemeinsam durchführen. Für DBS heißt das im einfachsten Fall, dass das DBMS(-backend) als Server auf einer Maschine läuft, und die Anwendungen oder Frontends auf verschiedenen Maschinen als Clients laufen.

Welche Vorteile hat solch ein System:

- Parallele Verarbeitung: Mehrere Prozessoren arbeiten an einer Aufgabe, so dass Durchsatz und Antwortzeiten verbessert werden.
- Die Server-Maschine kann speziell für DBMS-Zwecke ausgerüstet werden *(database machine)*.
- Die Client-Maschinen können auf die jeweiligen Anwendungen zugeschnitten werden (etwa mit besonderen Graphikprozessoren oder Druckern ausgerüstet werden).
- Die Anwendungen sind den betrieblichen Gegebenheiten entsprechend räumlich trennbar. Sie können gleichzeitig auf die DB zugreifen. Das DBMS auf der Server-Maschine sorgt dafür, dass nichts schief geht.

Es ist auch möglich, dass die DB auf verschiedene Server-Maschinen verteilt ist. Für den Client gibt es nun zwei Möglichkeiten:

- Ein Client kann auf eine beliebige Anzahl von Servern zugreifen, aber nur auf einen zur Zeit. Der Benutzer muss wissen, auf welchem Server sich die von ihm gewünschten Daten befinden. Er kann mit *einem* Zugriff nicht Daten von verschiedenen Servern kombinieren.
- Ein Client kann gleichzeitig auf eine beliebige Anzahl von Servern zugreifen. *Ein* Zugriff kann Daten von verschiedenen Servern kombinieren. Der Benutzer muss nicht wissen, wo sich die Daten befinden (*transparenter* Zugriff). Für ihn verhalten sich die vielen Server wie *ein* Server. Man spricht jetzt von einem **verteilten DBS (*distributed DBS*)**.

Es gibt natürlich DBSs, die für spezifische Zwecke entwickelt werden. Beispiel sind die *on-line transaction processing* Systeme (OLTP), die eine große Anzahl parallel laufender Transaktionen ohne große Verzögerung verarbeiten müssen.

Kapitel 3

Entwicklung von Datenbanksystemen

Die Entwicklung von Datenbanksystemen folgt einer im Software-Engineering oder generell im Systems-Engineering bewährten Methode oder Methodologie:

- **Definition einer Daten-Norm (Daten-Standard)** ***(data standard)***
- **Problembeschreibung** oder Anforderungsanalyse ***(problem statement*** oder *requirements specification, collection of facts)*
- **Analyse** ***(analysis)***: Identifikation von Entitäten, ihren Eigenschaften und ihren Beziehungen. (Erstellung eines Entity-Relationship-Modells (E/R) kann hier schon begonnen werden, ist aber nicht Ziel der Analyse.)
- **Entwurf** ***(design, logical or conceptual database design*** oder ***data model mapping)***: Erstellung eines Entity-Relationship-Modells (ER) oder Objekt-Modells (UML). Dann entweder
  1. Relationale DB
     (a) Umsetzung des Modells in Relationen
     (b) Normalisierung der Relationen

     oder andere Möglichkeiten:
  2. Objekt-relationale DB: Umsetzung des Modells in Klassen
  3. Objekt-orientierte DB: Umsetzung des Modells in Klassen
  4. Netzwerk-DB: Umsetzung des Modells in ein Bachmann-Diagramm
  5. Hierarchische DB: Umsetzung des Modells in Hierarchien
  6. ...
- **Implementierung** des Entwurfs mit Statements einer DDL
  1. Relationale DB: z.B. SQL2
  2. Objekt-relationale DB: z.B. SQL3

3. Objekt-orientierte DB: z.B. OQL und/oder C++/Java Klassen
4. Netzwerk-DBS:
5. Hierarchisches DBS:
6. ...

- **Erstellung von Anwendungen** Hier empfiehlt sich, für jede Anwendung einen eigenen Entwicklungsprozess zu initiieren.

Anstatt der hier vorgestellten vereinfachten Form eines Entwicklungsprozesses sollte man sich für große Projekte an die im RUP (Rational Unified Process) definierte Vorgehensweise halten.

Unterschiedliche Begriffe werden in der Datenbank- und objekt-orientierten Welt verwendet. Folgende Tabelle vergleicht die Begriffe:

E/R-Modell	objektorientiert	andere Bezeichnungen
Entität als Typ	Klasse	Typ
Entität als Instanz	Objekt	Instanz, Ausprägung, Exemplar, identifizierbares Etwas, Variable
Beziehung	Assoziation	Verknüpfung "Hat-ein"–Beziehung (`has-a`) Aggregation, Komposition "Benutzt-ein"–Beziehung (`uses-a`) Ähnlichkeit (`is-like-a`)
Eigenschaft (abgeleitet)	Attribut Methode	 Operation, Botschaft, Verhalten
Obertyp	Basisklasse	Oberklasse, Superklasse, Supertyp, Eltertyp, Obermenge "Ist-ein"–Beziehung (`is-a`), Erweiterung, Vererbung, Substitution, Generalisierung / Spezialisierung
Untertyp	Abgeleitete Klasse	Unterklasse, Subklasse, Subtyp, Kindtyp, Teilmenge

Auf die Daten-Norm, Problembeschreibung und Analyse wird in den nächsten Abschnitten eingegangen. Den anderen Punkten sind eigene Kapitel gewidmet.

3.1 Definition einer Daten-Norm

Unter einer Daten-Norm eines Datenbanksystems verstehen wir Regeln, nach denen das System entwickelt wird. Diese Regeln heißen auch **Daten-Standard oder Daten-Architektur** ***(enterprise data standard, data architecture)*** [37].

Oft kann man die Daten-Norm von anderen Systemen übernehmen. Sie umfasst alle Aspekte des Datenbanksystems. Ein Beispiel solcher Regeln soll verdeutlichen, was gemeint ist:

Daten Zugriff *(data access)***:**

1. Clients dürfen Daten nur über gespeicherte Datenbankprozeduren *(stored procedures)* aktualisieren.
2. Benutze Views um gespeicherte Daten auszugeben.
3. Der Client sollte eine minimale Anzahl von Zeilen erhalten.

Change Management Standard:

1. Alle Datenbankobjekte werden mit SQL-Skripten erzeugt.
2. Die SQL-Skripte werden in einer eigenen Datenbank gespeichert.

Versions-Kontrolle *(version control)*:

1. Check in / Check out, Delta Management

Allgemeine Namens-Konventionen *(general naming conventions)*:

1. Alle Variablennamen enthalten nur Kleinbuchstaben.
2. Variablennamen sind mnemonisch und repräsentieren den Inhalt der Variablen.
3. Variablennamen haben höchsten 40 Zeichen.
4. Variablennamen beginnen mit einem Buchstaben (a – z). Der Rest kann eine Kombination von Buchstaben, Zahlen und _ sein. Mit _ werden Teile des Namens getrennt.
5. Variablennamen enthalten keinen White Space.

Tabellennamen *(table names)*:

1. Die maximale Länge ist 20 Zeichen.
2. Benutze Singular anstatt des Plurals (”kunde” anstatt ”kunden”).
3. Vermeide Abkürzungen.
4. Verwende keine Schlüsselworte.

Spaltennamen *(column names)*:

1. Der Spaltenname beginnt mit den zwei ersten Zeichen des Tabellennamens.

Indexnamen *(index names)*:

1. Indexnamen enthalten die ersten fünf Zeichen des Tabellennamens und werden durchnumeriert beginnend mit `_1`.

Datentypen *(data types)*:

1. Alle Tabellen müssen ein Zeitstempel-Feld haben.
2. Benütze nur SQL2-Datentypen.

3. Minimiere den Speicherverbrauch durch Verwendung des kleinsmöglichen Datentyps (`TINYINT` anstatt von `INT`).
4. Vermeide Zeichenketten variabler Länge.
5. Vermeide NULLs.

Die Definition einer Daten-Norm lohnt sich, wenn nicht-triviale Anwendungen erstellt oder gewartet werden. Sie sollte so einfach wie möglich sein, und es muss dafür gesorgt werden, dass die Daten-Norm tatsächlich auch benutzt wird. Es gibt verschiedene Organisationen, die Daten-Normen definiert haben (ISO, OSF DCE, POSIX, X/OPEN, COSE, CORBA, IEEE, SQL Access Group, ISOC).

3.2 Problembeschreibung

In einer für den Laien verständlichen Sprache wird die vorliegende Datenbankanwendung beschrieben. Es sollen alle relevanten Daten und ihre Bedeutung möglichst genau beschrieben werden **(Datenanforderungen**, ***data requirements)***.

Ferner sind auch die Anforderungen an die Operationen auf den Daten zu beschreiben **(Operationenanforderungen**, ***operation requirements)***.

3.3 Analyse

Nackte Daten werden zu Information, wenn sie eine Interpretation innerhalb eines Unternehmens-Kontextes haben.

Jedes Datenbanksystem basiert auf atomaren Datenelementen. Bei der Definition von Datenelementen sollte man unbedingt nach der ***one-concept-equals-one-data-element*** Regel [25] vorgehen. Z.B. sollte man unbedingt unterscheiden zwischen dem Namen einer Firma und einer Referenz auf diese Firma. Falsch wäre auch, den Namen und die Adresse einer Firma in einem Datenelement zu verwalten. Ein Konzept *(concept)* ist hier zu verstehen als die kleinste Einheit, die noch Sinn macht, d.h. Bedeutung enthält.

Oft muss man zwischen Aufwand und Flexibilität abwägen, inwieweit man diese Regel konsequent verfolgt. Wenn wir z.B. den Namen einer Person als ein Datenelement betrachten, dann ist das zunächst wesentlich einfacher, als wenn wir den Namen bestehend aus drei Datenelementen (Vorname, Mittelname, Nachname) zusammensetzen. Allerdings vergeben wir uns bei der einfachen Lösung Flexibilität. Wir können zum Beispiel nicht ohne weiteres nach dem Nachnamen oder Vornamen sortieren. Für eine Anrede in einem Brief müssen wir den Namen kompliziert auseinandernehmen.

Kapitel 4

Entity-Relationship-Modell (E/R)

Datenbanksysteme "verstehen" ihre Daten i.a. sehr wenig, sie wissen nicht welche Bedeutung (Semantik) die Daten haben, was zur Folge hat, dass Benutzeroberflächen ziemlich primitiv sind. Semantik von Daten ist graduell zu verstehen: Denn Datenbankstruktur, Wertebereiche, Schlüssel und Integritätsregeln sind auch schon semantischer Natur. Z.B. würde ein Wertebereichskonzept verhindern, dass zwei Tabellen auf Grund gemeinsamer Werte von Teilegewichten und Liefermengen gejoined werden.

In diesem Kapitel beschäftigen wir uns mit Möglichkeiten (**semantische Modellierung**), mehr Semantik einem DBS zu vermitteln, indem wir als den prominentesten Vertreter der **erweiterten *(extended)*** oder **semantischen *(semantic)*** Modelle das Entity-Relationship-Modell **(E/R)** behandeln. "Semantisches Modell" heißt nicht, dass es gelungen ist, die ganze Semantik der Daten zu erfassen, sondern bedeutet nur *einen* weiteren Schritt im Prozess der semantischen Modellierung.

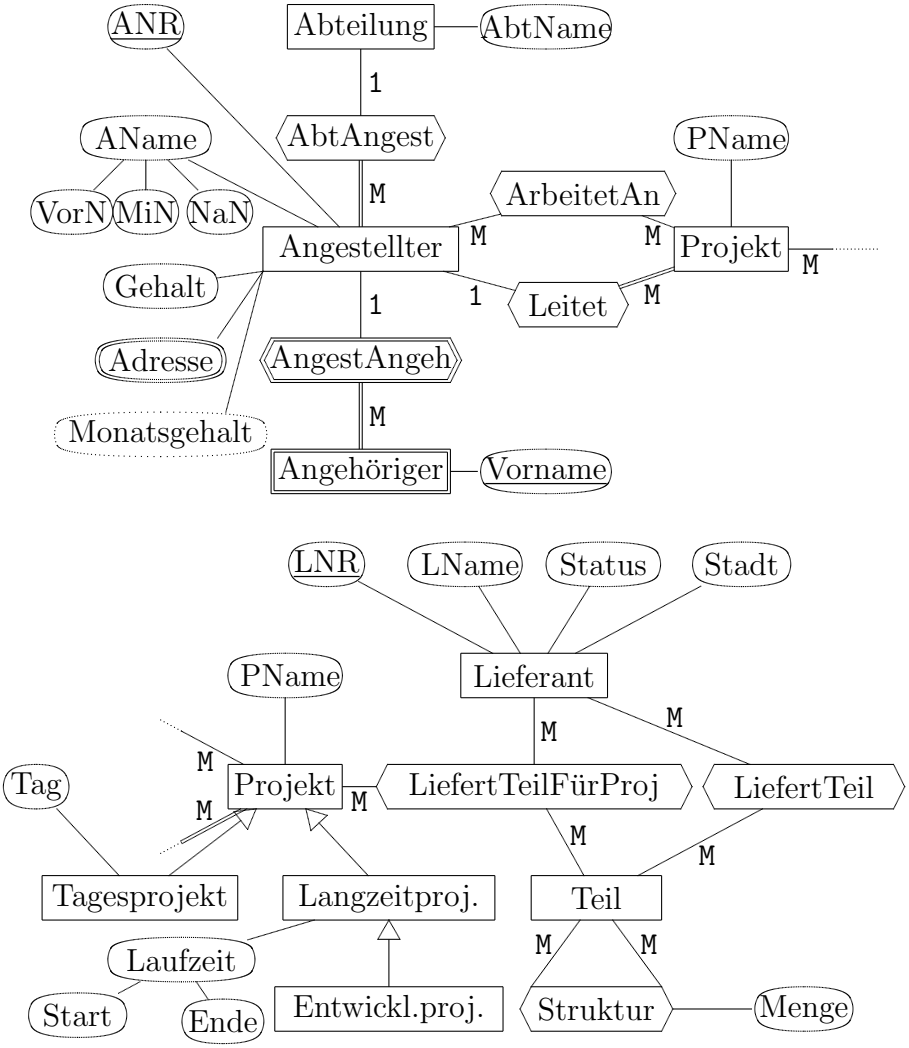
4.1 Elemente von E/R

1976 hat Peter Pin-Shan Chen [8] das E/R zusammen mit einer Diagramm-Technik eingeführt. Im folgenden werden die Grundbegriffe vorgestellt, die wegen ihrer axiomatischen Natur eigentlich nicht definiert, allenfalls erklärt werden können.

Der zentrale Begriff ist die **Entität *(entity)***. Die Anzahl der Entitäten eines Systems ist i.a. wesentlich kleiner als die Anzahl der Datenelemente. Dadurch werden die Problembeschreibung bzw Anforderungsanalyse und die konzeptuelle Entwurfsphase stark vereinfacht.

Folgende Abbildung zeigt ein Beispieldiagramm für eine kleine Firma. Dieses Beispiel wird im folgenden näher erklärt.

Beispieldiagramm:



4.1.1 Entität *(entity)*

Eine Entität definiert Chen als "a thing which can be distinctly identified". Eine Entität ist ein "Etwas" oder Objekt, das eindeutig identifiziert werden kann.

Entitäten können eine physikalische oder auch begriffliche, strategische Existenz haben, z.B. Person, Firma, Rezeptur.

Den Begriff Entität verwenden wir sowohl für Typen oder Klassen von Objekten – dann eventuell genauer **Entitätstyp** ***(intension of entity)*** – alsauch für die Objekte selbst – **Instanzen, Exemplare, Ausprägungen** ***(extension of entity)*** eines Entitätstyps.

Es werden **reguläre (starke,** ***regular, strong)*** und **schwache** ***(weak)*** Entitäten unterschieden. Die Existenz einer schwachen Entität beruht auf der Existenz einer anderen Entität. Der Angehörige eines Angestellten kann in der Datenbank nur vorkommen, wenn auch der entsprechende Angestellte vorkommt, und ist daher eine schwache Entität.

Entitäten werden im Diagramm durch Rechtecke dargestellt, die den Namen der Entität enthalten. Bei schwachen Entitäten wird das Rechteck verdoppelt.

Schwache Entitäten stehen immer in Beziehung zu einer oder mehreren Entitäten **(idenitifizierende** Beziehung).

4.1.2 Beziehung *(relationship)*

Eine Beziehung oder auch Assoziation ist "an association among entities". Die Beziehung "AbtAngest" repräsentiert die Tatsache, dass ein bestimmter Angestellter zu einer bestimmten Abteilung gehört. Die an einer Beziehung beteiligten Entitäten heißen **Teilnehmer *(participants)*** einer Beziehung. Die Anzahl der Teilnehmer ist der **Grad *(degree)*** der Beziehung. Je nach Grad kann die Beziehung **binär, ternär** oder ***n*-wertig** sein.

Wenn jede Instanz einer teilnehmenden Entität an mindestens einer Instanz der Beziehung teilnimmt, d.h. mindestens einmal die Beziehung eingeht, dann ist diese Art der Teilnahme **zwingend *(total, mandatory, existence dependency)***, sonst **freigestellt, partiell *(optional, partial)***. Jeder Angestellte muss in einer Abteilung sein, daher nimmt der Angestellte an der Beziehung "AbtAngest" zwingend teil. Eine Abteilung muss keinen Angestellten haben. Abteilung nimmt daher an dieser Beziehung partiell teil.

Eine Beziehung kann **eineindeutig *(one-to-one, 1:1)***, **einseitig eindeutig *(one-to-many, 1:M)*** oder **komplex *(many-to-many, M:N)*** sein. Die Zahlen 1, N, M sind *Multiplizitäten*. Die Bezeichnung *Kardinalität* bzw *Kardinalitätsverhältnis (cardinality ratio)* ist veraltet und falsch; der Begriff Kardinalität ist reserviert für die Anzahl der Elemente von Mengen.

Die durch Multiplizitäten und Art der Teilnahme dargestellten Einschränkungen heißen **strukturelle Einschränkungen *(structural constraints)***.

Eine Beziehung kann auch eine Dreier- oder höherwertige Beziehung (Beziehung zwischen drei oder noch mehr Entitäten) mit den entsprechenden Kombinationen der Multiplizitäten sein. Von der Einführung solcher Beziehungen ist abzuraten. Es sollte versucht werden, diese Beziehungen durch mehrere Zweier-Beziehungen darzustellen. Dadurch werden von vornherein Redundanzen vermieden. Eine n-wertige Beziehung kann immer durch eine schwache Entität dargestellt werden, die binäre Beziehungen zu den n beteiligten Entitäten unterhält.

Beziehungen werden im Diagramm durch Rhomben dargestellt, die den Namen der Beziehung enthalten. Wenn es eine Beziehung zu einer schwachen Entität ist, dann wird der Rhombus verdoppelt. Zwischen Entität und Beziehung wird bei partieller Teilnahme eine Kante, bei zwingender Teilnahme eine Doppelkante gezogen. Bei "eins-zu-" wird eine "1", bei "viele-zu-" ein "M" an die Kante geschrieben.

Bei Beziehungen zwischen Instanzen desselben Entitätstyps **(Eigenbeziehung, *involuted relationship)*** kann die Art der Beziehung durch **Rollennamen** geklärt werden, die eventuell in Klammern an die Beziehungskanten geschrieben werden. Beispiel ist die Beziehung "Struktur".

4.1.3 Eigenschaft *(property)*

Entitäten *und* Beziehungen haben Eigenschaften, mit denen sie näher beschrieben werden. Die Eigenschaften – genauer die *Werte* der Eigenschaften – können sein:

- **eindeutig** ***(unique, key)***
  Die Entität wird durch die Eigenschaft eindeutig bestimmt. "ANR" ist eine eindeutige Eigenschaft.
- **einfach** ***(simple, atomic)*** oder **zusammengesetzt** ***(composite)***
  "Gehalt" ist eine einfache Eigenschaft, "Name" ist eine zusammengesetzte Eigenschaft.
- **ein- oder mehrwertig** ***(single- or multi-valued)***
  "Adresse" ist eine mehrwertige Eigenschaft. Ein Angestellter kann mehrere Adressen haben.
- **abhängig** bzw **abgeleitet** ***(derived)*** oder **nicht abhängig** ***(base, stored)***
  "Monatsgehalt" ist eine abhängige oder abgeleitete Eigenschaft.

Künstliche Schlüsseleigenschaften sollten hier nur erscheinen, wenn der Auftraggeber ausdrücklich die Verwaltung eines Schlüssels wünscht. Ansonsten gehört die Einführung von Schlüsseln zum konkreten Datenbank-Entwurf.

Im Diagramm werden Eigenschaften durch Ellipsen mit dem Namen der Eigenschaft dargestellt. Eindeutige Eigenschaften werden unterstrichen. Bei mehrwertigen Eigenschaften wird die Ellipse verdoppelt. Abhängige Eigenschaften haben eine punktierte Ellipse. Da die Einzeichnung von Eigenschaften das Diagramm unübersichtlich macht, sollten Eigenschaften separat aufgeführt werden.

Bei schwachen Entitäten gibt es häufig einen **partiellen Schlüssel** ***(partial key)***, der die Entität nur bezüglich der sie bedingenden Entität eindeutig bestimmt. Der partielle Schlüssel wird auch unterstrichen.

4.1.4 Aggregation *(aggregation)* und Komposition *(composition)*

Aggregation oder Komposition sind ein Abstraktionskonzept für den Bau von aus Komponenten zusammengesetzten Entitäten (Komponentengruppen).

Wenn die Komponenten eher Eigenschaften sind, dann wird man die Aggregation im E/R-Modell mit zusammengesetzten mehrwertigen Eigenschaften modellieren.

Wenn die Komponenten eigenständige Entitäten sind, dann empfiehlt sich eine `has-a`-Beziehung bzw `is-part-of`-Beziehung.

4.1.5 Untertyp *(subtype)*

Die Instanz einer Entität ist mindestens vom Typ *eines* Entitätstyps. Sie kann aber auch Instanz *mehrerer* Entitätstypen sein, wenn eine "Ist-ein"-Typenhierachie vorliegt. Ein Systemprogrammierer *ist* ein Programmierer (ist Untertyp von Programmierer), ein Programmierer *ist* ein Angestellter (ist Untertyp von Angestellter). Eigenschaften und Beziehungen werden von den Untertypen geerbt. Alles, was für den **Obertyp (Vatertyp, Obermenge,** ***parent type, supertype, superclass)*** gilt, gilt auch für den **Untertyp (Sohntyp, Teilmenge,** ***subtype, subset, subclass)***.

Der Untertyp ist eine **Spezialisierung** des Obertyps. Der Obertyp ist eine **Generalisierung** des Untertyps oder verschiedener Untertypen. Generalisierung und Spezialisierung sind *inverse* Prozeduren der Datenmodellierung.

Betrachten wir Untertypen als Teilmengen, dann stellt sich die Frage, ob die Teilmengen **disjunkt** ***(disjoined)*** sind. Man spricht von nicht **überlappenden** ***(non overlapping)*** Teilmengen oder von der **Einschränkung durch Disjunktheit** ***(disjointness constraint)***.

Eine weitere Einschränkung der Spezialisierung/Generalisierung ist die der **Vollständigkeit** ***(completeness constraint)***, die **total** oder **partiell** sein kann. Bei der totalen Spezialisierung muss jede Entität des Obertyps auch Entität eines Untertyps sein. Eine Generalisierung ist üblicherweise total, da der Obertyp aus den Gemeinsamkeiten der Untertypen konstruiert wird.

Es lassen sich folgende Einfüge- und Löschregeln ableiten:

- Löschen einer Entität im Obertyp hat Löschung dieser Entität in allen Untertypen zur Folge.
- Einfügen einer Entität in den Obertyp muss zur Folge haben, dass die Entität in alle zu ihr passenden Untertypen eingefügt wird. Bei totaler Spezialisierung muss die Entität mindestens in einem Untertypen eingefügt werden.

Da jeder Untertyp auch seinerseits Untertypen haben kann, führt das zu **Spezialisierungshierarchien** und zu **Spezialisierungsnetzen** ***(specialization lattice)*** bei Untertypen, die mehr als einen Obertyp haben **(Mehrfachvererbung,** ***multiple inheritance, shared subclass)***.

Generalisierung und Spezialisierung sind nur verschiedene Sichten auf dieselbe Beziehung. Als Entwurfs-Prozesse sind das allerdings unterschiedliche Methoden:

- Spezialisierung: Wir beginnen mit einem Entitätstyp und definieren dann Untertypen durch sukzessive Spezialisierung. Man spricht auch von einem ***top-down conceptual refinement***.
- Generalisierung: Der umgekehrte Weg beginnt bei sehr speziellen Untertypen, aus denen durch sukzessive Generalisierung eine Obertypenhierarchie bzw ein Obertypennetz gebildet wird. Man spricht von ***bottom-up conceptual synthesis***.

In der Praxis wird man irgendwo in der Mitte beginnen und beide Methoden anwenden.

Die Typ-Untertyp-Beziehung wird im Diagramm durch einen Pfeil vom Untertyp zum Obertyp dargestellt. Manchmal erscheint auch das Teilmengensymbol, weil die Instanzen des Subtyps Teilmengen der Instanzen des Obertyps sind.

Beziehungen können auch eine Untertyp-Obertyp-Struktur haben.

4.1.6 Kategorie *(category)*

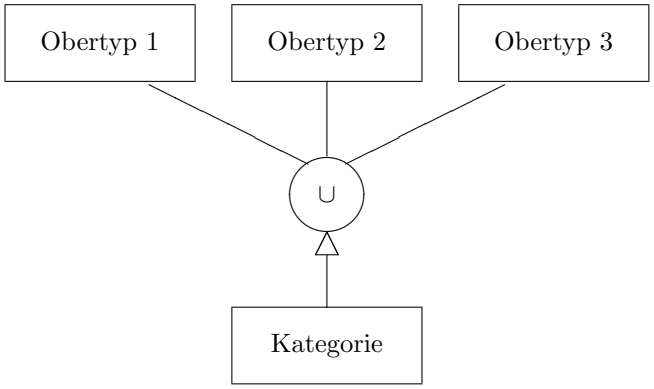
Ein Fahrzeughalter kann eine Person oder eine Firma sein. Da nicht jede Person und nicht jede Firma ein Fahrzeughalter ist, kann Fahrzeughalter kein Obertyp sein.

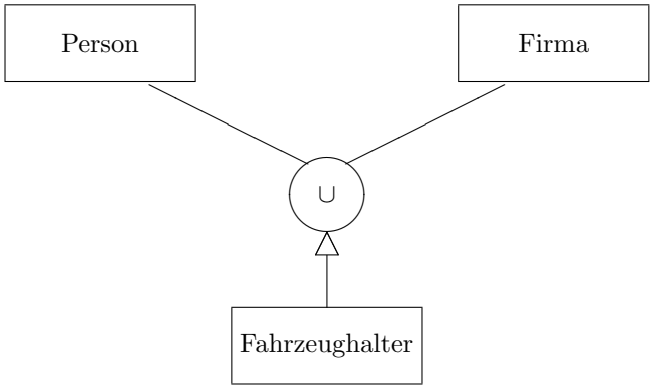
Fahrzeughalter kann auch kein Untertyp sein. Denn von wem sollte Fahrzeughalter ein Untertyp sein – Person oder Firma?

Fahrzeughalter ist ein Beispiel für eine **Kategorie**, die von verschiedenen Obertypen **exklusiv** erben kann. Mit "exklusiv" ist gemeint, dass die Entität Fahrzeughalter nur von genau

einem Obertyp erben kann, dass sie nicht die Kombination mehrerer Obertypen ist, wie das bei Mehrfachvererbung der Fall ist. Eine Kategorie erbt *einen* Obertyp aus einer Vereinigung von Obertypen.

Die graphische Darstellung sieht folgendermaßen aus, wobei der Kreis ein Vereinigungszeichen enthält.

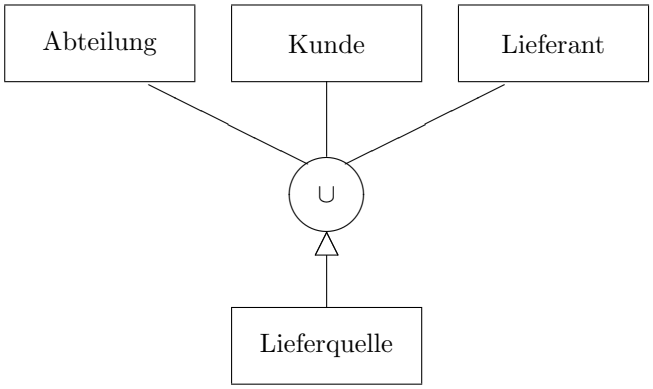




Eine Kategorie kann **partiell** oder **total** sein. Totalität bedeutet, dass jeder Obertyp der Kategorie zur Kategorie gehören muss. In diesem Fall ist eine Darstellung als Obertyp-Untertyp-Beziehung auch möglich, wobei die Kategorie der Obertyp ist, und meistens vorzuziehen, insbesondere wenn die Entitäten viele Attribute teilen.

Kategorien kommen häufig vor. Sie sind eine Art von Rolle, die eine Entität spielen kann. Insbesondere Personen sind in den verschiedensten Rollen zu finden.

Beispiel:



4.2 Verwandte Modelle

In den vorhergehenden Abschnitten wurde *eine* Form des E/R-Modells vorgestellt. Wir nennen sie E/R-Modell schlechthin. Diese Form scheint uns am nützlichsten zu sein, unterscheidet sich aber von insbesondere historischen Vorläufern, von denen im folgenden einige kurz charakterisiert werden.

4.2.1 EER-Modell

Das EER-Modell *(Erweitertes (Extended) Entity-Relationship-Modell)* wurde ursprünglich 1976 von Chen vorgeschlagen. Informationen werden mit Hilfe von drei Begriffen dargestellt:

1. *Entitäten (entities)* stehen für die zu modellierenden Objekte.
2. *Attribute (attributes)* beschreiben die Eigenschaften von Objekten.
3. *Beziehungen (relationships)* zwischen Entitäten.

Das EER-Modell entspricht dem E/R-Modell ohne Vererbung.

4.2.2 RM/T-Modell

Im 1979 von Codd vorgestellten RM/T-Modell beschreiben Entitäten Objekte *und* Beziehungen. Sie haben Eigenschaften und können durch die Operationen *ErzeugeEntität, AktualisiereEntität* und *LöscheEntität* verändert werden. Das Modell umfasst Untertyp/Obertyp-Hierarchie. Die Entitäten werden klassifiziert in

***asssoziativ (associative)* :** Entität repräsentiert eine mögliche N:M-Beziehung zwischen voneinander unabhängigen Entitäten.

charakteristisch (characteristic) **:** Entität beschreibt – ähnlich einer schwachen Entität – einen Aspekt einer übergeordneten Entität.

grundlegend (kernel) **:** Entität ist weder assoziativ noch charakteristisch.

bestimmend (designative) **:** Entität repräsentiert eine mögliche 1:N-Beziehung zwischen voneinander unabhängigen Entitäten. Wenn ein Mitarbeiter nur zu einer Abteilung gehört, dann sagt man, er *bestimme* die Abteilung.

Ein wichtiges Konzept innerhalb des RM/T-Modells ist, dass jede Entität mit Hilfe eines systemerzeugten Bezeichners, des sogenannten *Surrogats*, identifiziert werden kann. Dadurch wird die Definition von Primärschlüsseln auf Modellebene überflüssig. Aber wir haben auch schon beim E/R-Modell von Schlüsseldefinitionen abgeraten, wenn diese sich nicht zwingend aus der Anforderungsanalyse ergeben.

4.3 Andere Datenmodelle

4.3.1 Funktionale Datenmodelle (FDM)

FDMs nutzen das Konzept der mathematischen Funktion als den fundamentalen Modellierungskonstrukt. Jede Anfrage wird als ein Funktionsaufruf mit bestimmten Argumenten behandelt. Das bekannteste Beispiel für eine Anfragesprache ist DAPLEX.

4.3.2 Verschachteltes relationales Datenmodell

Das verschachteltes relationale Datenmodell *(nested relational data model)* gibt die Einschränkung der ersten Normalform auf und ist daher auch bekannt als Non-1NF oder N1NF relationales Modell. Es werden zusammengesetzte und mehrwertige Attribute erlaubt, was zu komplexen Tupeln mit einer hierarchischen Struktur führt.

Damit können Objekte repräsentiert werden, die natürlicherweise hierarchisch strukturiert sind. Diese Datenmodell lässt sich gut mit einer hierarchischen DB implementieren. 1:M-Beziehungen sind leicht darzustellen, M:N-Beziehungen dagegen äußerst schwierig.

4.3.3 Strukturelles Datenmodell

Das strukturelle Datenmodell ist eine Erweiterung des relationalen Modells, die vor Einführung von Fremdschlüsseln in das relationale Modell gemacht wurde und daher als wesentlichen Bestandteil **Beziehungen *(connections)*** zwischen Relationen als strukturelles Element hat.

4.3.4 Semantisches Datenmodell (SDM)

SDM führt die Konzepte von Klasse und Subklasse ein. Es werden verschiedene Klassentypen unterschieden: Klassen von konkreten Objekten, von abstrakten Objekten, von Aggregaten und von Ereignissen. SDM-Konzepte flossen in das objekt-orientierte E/R-Modell ein.

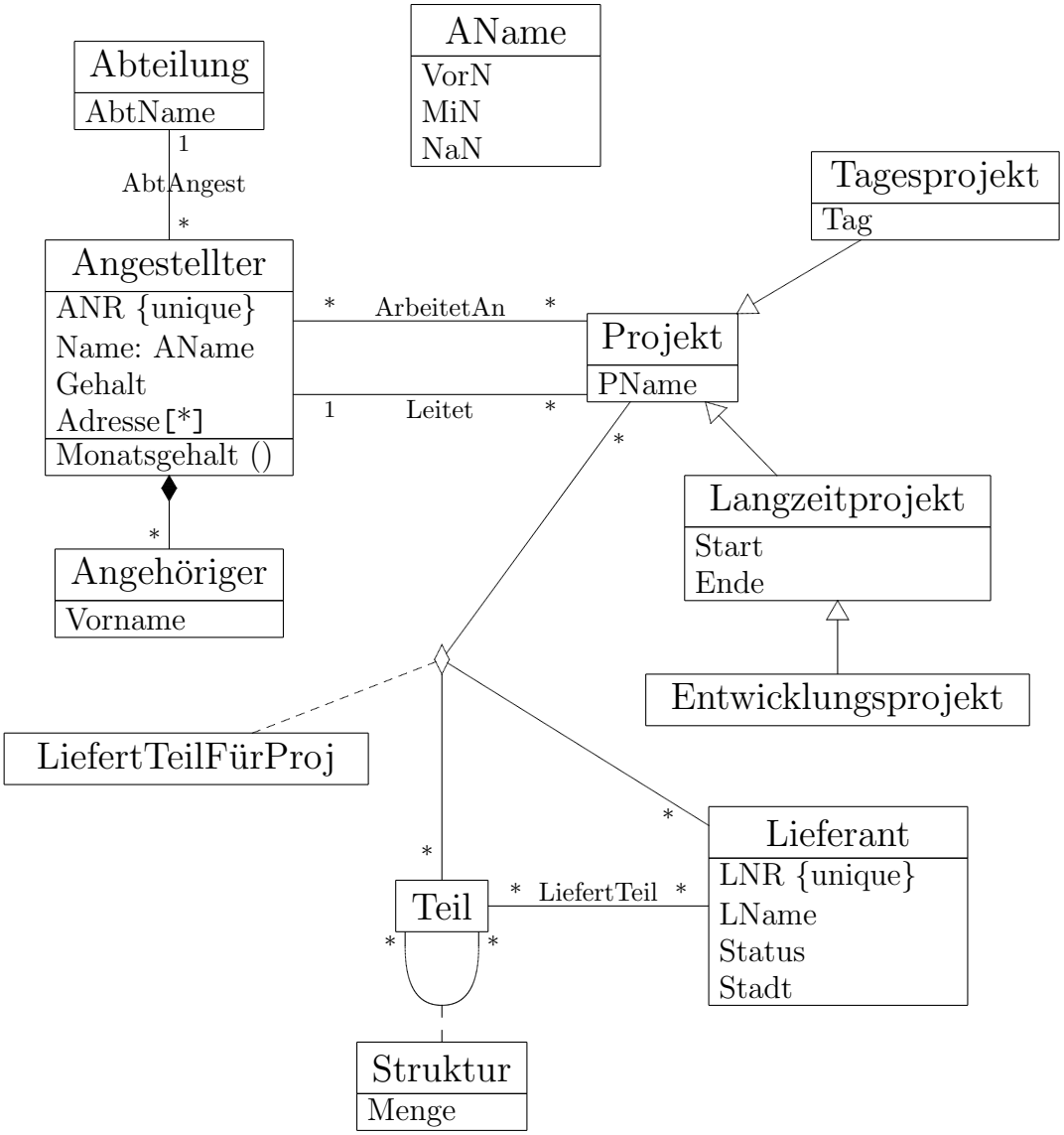
4.3.5 Object Role Modeling (ORM)

Bei ORM *(object role modeling, fact-oriented modeling* oder auch NIAM *Natural language Information Analysis Method* oder auch FCO-IM *Fully-Communication Oriented Information Modeling)* werden Attribute vermieden. Attribute werden als sogenannte ***value types*** betrachtet, die Beziehungen *(fact)* zu sogenannten ***entity types*** eingehen. Dadurch werden die Diagramme "umgangssprachlicher" und für nicht Fachleute leichter verständlich. Allerdings werden die Diagramme auch sehr überladen. Es gibt sehr viel Literatur dazu[21].

Kapitel 5

Entity-Relationship-Modell in UML-Notation

Das Einführende Beispiel des letzten Kapitels sieht in UML-Notation folgendermaßen aus:



Die wichtigste Komponente objekt-orientierter Softwareentwicklung ist das Klassen- oder Objekt-Modell **(statische Sicht, strukturelle Sicht, *static view, structural view)***, das graphisch als Klassendiagramm ***(class diagram)*** dargestellt wird. In diesem Diagramm werden die Klassen und ihre Beziehungen zueinander dargestellt. Beim Datenbankdesign werden wesentliche Teile dieser Technik mit großem Erfolg schon seit 1976 (Peter Pin-Shan Chen) als Entity-Relationship-Diagramm verwendet.

Das Verhalten eines Objekts wird beschrieben, indem Operationen benannt werden, ohne dass Details über das dynamische Verhalten gegeben werden.

5.1 Objekt *(object)*, Klasse *(class)*, Entität *(entity)*

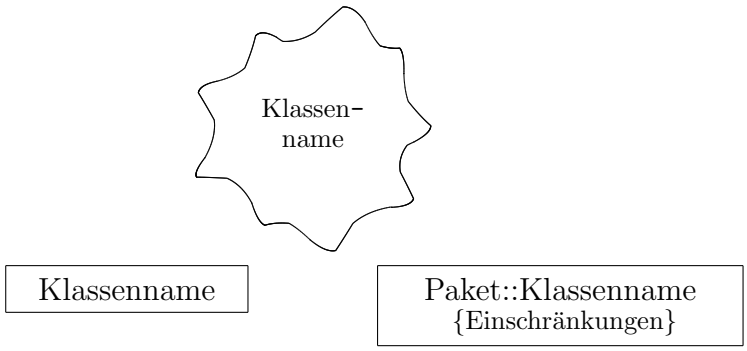
Chen definiert eine Entität als "a thing which can be distinctly identified". Eine Entität ist ein "Etwas" oder Objekt, das eindeutig identifiziert werden kann.

Entitäten können eine physikalische oder auch begriffliche, strategische Existenz haben, z.B. "Person", "Firma", "Rezeptur", "Vorgehensweise" sind Entitäten.

Den Begriff Entität verwenden wir sowohl für Typen oder **Klassen** von Objekten – dann eventuell genauer **Entitätstyp *(intension of entity)*** – alsauch für die **Objekte** selbst – **Instanzen, Exemplare, Ausprägungen *(extension of entity)*** eines Entitätstyps.

Entitäten werden im Diagramm durch Rechtecke dargestellt, die den Namen der Entität enthalten. Normalerweise werden in den Diagrammen immer Klassen verwendet. Objekte werden **unterstrichen**.

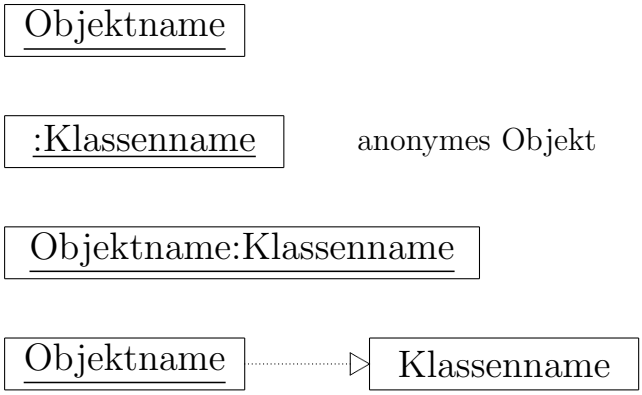
5.1.1 Klasse



Das ist die einfachste Notation für eine Klasse. Dem Namen der Klasse kann man einen Paketnamen und Einschränkungen mitgeben. Wie man Attribute und Operationen einer Klasse notiert, folgt in einem späteren Abschnitt.

5.1.2 Objekt

Objekte werden in Klassendiagrammen relativ selten verwendet. Trotzdem gibt es ziemlich viele Notationsmöglichkeiten.



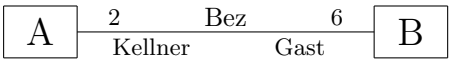
Instanziierung eines Objekts

5.2 Multiplizitäten

Bevor wir auf die Notation der verschiedenen Beziehungen zwischen Klassen eingehen, müssen wir zeigen, wie die **Multiplizität** ***(multiplicity)*** oder **Ordinalität** ***(ordinality)*** zu notieren ist. Mit der Multiplizität wird die Anzahl der an einer Beziehung beteiligten Objekte angegeben. (Nach UML darf dafür nicht mehr der Begriff **Kardinalität** ***(cardinality)*** verwendet werden, der für die Anzahl der Objekte einer Klasse reserviert ist.)

Multiplizität	Bedeutung
1	genau ein
*****	viele, kein oder mehr, optional
1..*	ein oder mehr
0..1	kein oder ein, optional
m..n	m bis n
m..*	m bis unendlich
m	genau m
m,l,k..n	m oder l oder k bis n

Auf welcher Seite einer Beziehung müssen Multiplizitäten stehen? Da das in UML leider nicht sehr vernünftig definiert wurde und daher immer wieder zu Missverständnissen führt, soll das hier an einem Beispiel verdeutlicht werden. Folgende Abbildung zeigt eine Beziehung `Bez` zwischen zwei Klassen `A` und `B`. Die Objekte der Klassen treten dabei auch noch in **Rollen** ***(role)*** auf:



Das bedeutet:

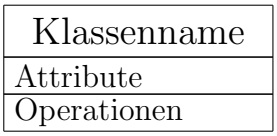
> *Ein* `A`*-Objekt steht (in der Rolle* `Kellner`*) zu genau 6* `B`*-Objekten in der Beziehung* `Bez` *(, wobei die* `B`*-Objekte dabei in der Rolle* `Gast` *auftreten).*
> *Ein* `B`*-Objekt steht (in der Rolle* `Gast`*) zu genau 2* `A`*-Objekten in der Beziehung* `Bez` *(, wobei die* `A`*-Objekte dabei in der Rolle* `Kellner` *auftreten).*

Bemerkung: Der Rollenname bedeutet oft nur, dass in unserem Beispiel die Klasse `A` eine Referenz (oder Pointer) mit Name `Gast` hat, und `B` eine Referenz mit Name `Kellner` hat.

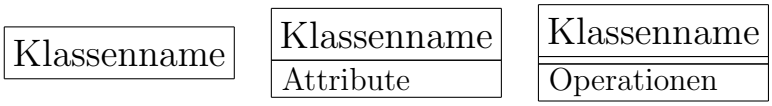
Jetzt können wir auf die verschiedenen Beziehungen einer Klasse zu anderen Klassen eingehen. Die engste Beziehung haben dabei die Eigenschaften einer Klasse.

5.3 Eigenschaften *(properties)*

Zu den Eigenschaften einer Klasse gehören die **Attribute** ***(attribute)*** und die **Operationen** ***(operation)*** oder Methoden.



Die Angabe von Attributen oder Operationen ist optional. Sie erscheinen meistens nur in *einem* Diagramm. (Eine Klasse kann in vielen Diagrammen vorkommen.)



Bemerkung: Das die Klasse repräsentierende Rechteck wird in ***compartments*** eingeteilt. Eine Klasse kann beliebig viele Compartments haben, wenn das Design es erfordert.

5.3.1 Attribute

Die Darstellung eines Attributs hat folgende Form:

$$\begin{aligned}\text{Sichtbarkeit}_{\text{opt}}/_{\text{opt}}\text{Name}_{\text{opt}} &: \text{Typ}_{\text{opt}}[\text{Multiplizität}]_{\text{opt}} \\ &= \text{Anfangswert}_{\text{opt}}\{\text{Eigenschaften}\}_{\text{opt}}\end{aligned}$$

Alle Angaben sind optional (opt). Wenn das Attribut aber überhaupt erscheinen soll dann muss wenigsten entweder der Name oder der :Typ angegeben werden. (Ein Attribut kann ganz weggelassen werden.) Die Sichtbarkeit kann die Werte

"`+`" ***(public)***,
"`#`" ***(protected)***,
"`-`" ***(private)*** und
"`~`" ***(package)***

annehmen.

"`/`" bezeichnet ein abgeleitetes Attribut, ein Attribut, das berechnet werden kann.

Die "Eigenschaften" sind meistens Einschränkungen, wie z.B.:

- Ordnung:**ordered** oder **unodered**
- Eindeutigkeit: **unique** oder **not unique**
- Es können hier auch die Collection-Typen verwendet werden: **bag, orderedSet, set, sequence**
- Schreib-Lese-Eigenschaften, z.B.: **readOnly**
- Einschränkungen beliebiger Art

Im allgemeinen sind Attribute Objekte von Klassen. (Wenn sie primitive Datentypen sind, dann kann man sie als Objekte einer Klasse des primitiven Datentyps auffassen.)

Statische Attribute (Klassenattribute) werden unterstrichen.

5.3.2 Operationen

Die Darstellung einer Operation hat folgende Form:

$$\text{Sichtbarkeit}_{\text{opt}}\text{Name (Parameterliste)} : \text{Rückgabetyp}_{\text{opt}} \quad \{\text{Eigenschaften}\}_{\text{opt}}$$

Mit Ausnahme des Namens und der Parameterliste der Operation sind alle Angaben optional (opt). Die Sichtbarkeit kann ebenfalls die Werte

"`+`" ***(public)***,
"`#`" ***(protected)***,
"`-`" ***(private)*** und
"`~`" ***(package)***

annehmen.

Statische Operationen (Klassenmethoden) werden unterstrichen.

Beispiel:

Konto
- anzKonten :int
- kontoStand :double
- transaktion :Transaktion [*]
+ getAnzKonten () :int
+ getKontoStand () :double
+ letzteTransaktion () :Transaktion
+ transaktion (datum:Date) :Transaktion [*]

Als "Eigenschaften" bieten sich bei Operationen irgendwelche Vor-, Nachbedingungen und Invarianten an ***(preconditions, postconditions, invariants)***. Exceptions und sogenannte Bodyconditions (Bedingungen, die durch Subklassen überschreibbar sind) gehören auch dazu. Oft ist es vernünftiger diese Eigenschaften in einer Notiz unterzubringen.

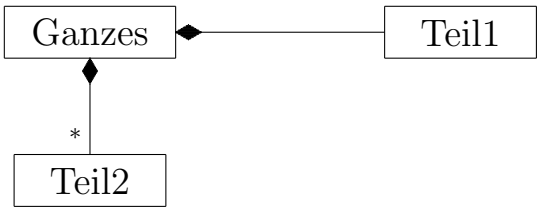
5.4 Teil-Ganzes-Beziehung

Die **Teil-Ganzes-** oder **Hat-ein-**Beziehung ***(whole-part, has-a)*** (auch Komponente / Komponentengruppe) hat große Ähnlichkeit mit Attributen. D.h. viele Attribute können auch als eine Hat-ein-Beziehung modelliert werden. In den objektorientierten Programmiersprachen gibt es da oft keine Implementierungsunterschiede.

Bei der Hat-ein-Beziehung unterscheidet man noch **Komposition** ***(composition)*** und **Aggregation** ***(aggregation)***.

5.4.1 Komposition

Die Komposition wird mit einer gefüllten Raute auf der Seite des Ganzen notiert (mit impliziter Multiplizität 1). Auf der Seite des Teils kann eine Multiplizität angegeben werden.



Bei der Komposition besteht zwischen Teil und Ganzem eine sogenannte **Existenzabhängigkeit**. Teil und Ganzes sind nicht ohne einander "lebensfähig". Ein Teil kann nur zu *einem* Ganzen gehören.

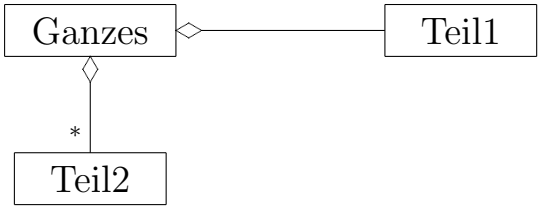
Komposition und Attribut sind kaum gegeneinander abzugrenzen. Im Hinblick auf Datenbanken haben die Teile der Komposition wie Attribute meistens keine eigene Persistenz. D.h. wenn das Ganze aus der Datenbank gelöscht wird, werden auch alle seine Teile gelöscht. Wenn ein Ganzes neu angelegt wird, werden auch alle seine Teile neu erzeugt. Ein Attribut beschreibt den Zustand eines Objekts und kann verwendet werden, um ein Objekt etwa in einer Datenbank zu suchen. Das Teil einer Komposition wird in dem Zusammenhang eher nicht verwendet.

Attribute und Teile (Komposition) können zwar Beziehungen zu Objekten außerhalb des Ganzen haben, wobei aber nur in der Richtung nach außen navigierbar ist. D.h. das Teil kann von außerhalb des Ganzen nicht direkt referenziert werden.

Die Komposition wird manchmal auch so interpretiert, dass die Teile nur einen einzigen Besitzer haben.

5.4.2 Aggregation

Die Aggregation wird mit einer leeren Raute auf der Seite des Ganzen notiert (mit impliziter Multiplizität 0..1). Auf der Seite des Teils kann eine Multiplizität angegeben werden.



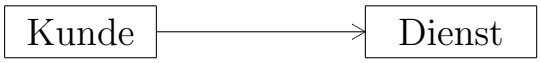
Die Teile können – zeitlich – vor und nach dem Ganzen existieren. Ein Teil kann durch ein anderes Teil desselben Typs ausgetauscht werden. Ein Teil kann von einem Ganzen zu einem anderen Ganzen transferiert werden. Aber ein Teil gehört zu einer Zeit höchstens zu *einem* Ganzen.

Im Hinblick auf Datenbanken haben hier die Teile meistens eine eigene Persistenz. Sie werden nicht automatisch mit dem Ganzen gelöscht oder angelegt oder aus der Datenbank geladen.

Oft wird die Aggregation auch so interpretiert, dass die Teile mehr als einen Besitzer haben, wobei die Besitzer aber unterschiedlichen Typs sein müssen.

5.5 Benutzung

Die **Benutzung** oder **Navigierbarkeit** oder **Delegation** ***(uses-a)*** ist eine Beziehung zwischen Benutzer ***(user)*** und Benutztem ***(used)***, bei der der Benutzte meistens von vielen benutzt wird und nichts von seinen Nutzern weiß.



Das kann man folgendermaßen lesen:

- Benutzung: Ein Kunde *benutzt* ein Dienst-Objekt.
- Delegation: Ein Kunde *delegiert an* ein Dienst-Objekt.
- Navigierbarkeit: Ein Kunde *hat eine Referenz* auf ein Dienst-Objekt. Ein Kunde kennt ein Dienst-Objekt.

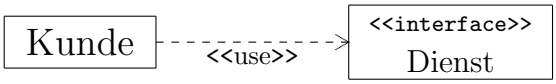
Auf der Seite des Benutzers hat man implizit die Multiplizität $*$.

Auf der Dienst-Seite ist auch eine Multiplizität möglich, wenn der Kunde etwa mehr als ein Dienstobjekt benutzt.

Der Dienst hat immer eine eigene Persistenz. Sein Lebenszyklus ist unabhängig von dem der Kunden. Er kennt typischerweise seine Kunden nicht, d.h. es gibt keine Navigationsmöglichkeit vom Dienst zum Kunden.

Die Benutzungs-Beziehung impliziert folgende Multiplizitäten:

Wenn eine Schnittstelle benutzt wird, dann kann auch folgende Notation verwendet werden:



5.6 Erweiterung, Vererbung

Die Instanz einer Entität ist mindestens vom Typ *eines* Entitätstyps. Sie kann aber auch Instanz *mehrerer* Entitätstypen sein, wenn eine "Ist-ein"-Typenhierachie vorliegt. Z.B. müssen folgende Fragen mit "ja" beantwortbar sein:

- Ist ein Systemprogrammierer ein Programmierer?
- Ist ein Systemprogrammierer eine **Erweiterung *(extension)*** von Programmierer?
- Ist ein Systemprogrammierer eine **Subklasse *(subclass)*** von Programmierer?
- Ist ein Systemprogrammierer ein **Untertyp *(subtype)*** von Programmierer?
- Sind alle Elemente der Menge Systemprogrammierer Elemente der Menge Programmierer?

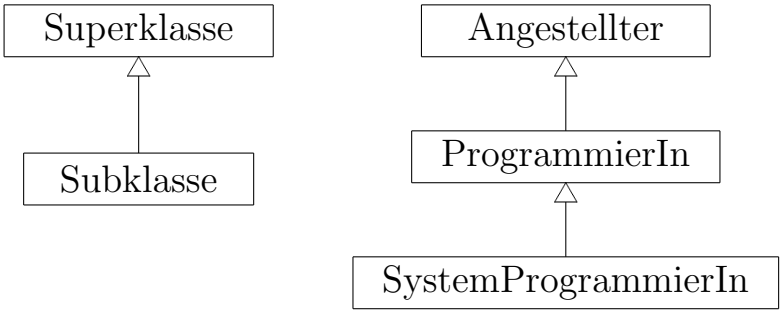
Ferner *ist* ein Programmierer ein Angestellter (ist Untertyp von Angestellter). Eigenschaften und Beziehungen werden von den Untertypen geerbt. Alles, was für die **Superklasse, (Basisklasse, Eltertyp, Obertyp, Obermenge, *superclass, baseclass, parent type, supertype, superset)*** gilt, gilt auch für die **Subklasse (Kindtyp, Subtyp, Teilmenge, *subclass, child type, subtype, subset)***.

Die wichtigsten Tests, ob wirklich eine Erweiterung vorliegt sind:

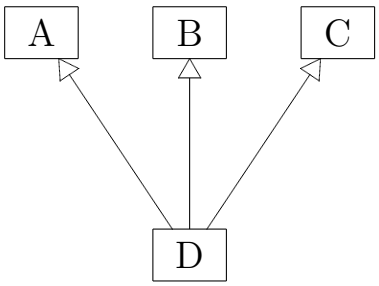
- Ist-ein-Beziehung
- Substitutionsprinzip von Liskov
- Teilmengen-Beziehung

Der Untertyp ist eine **Spezialisierung** des Obertyps. Der Obertyp ist eine **Generalisierung** des Untertyps oder verschiedener Untertypen. Generalisierung und Spezialisierung sind *inverse* Prozeduren der Datenmodellierung.

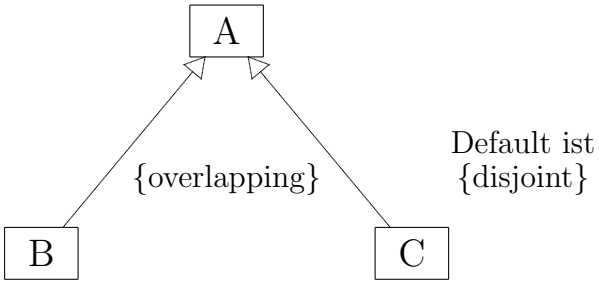
Die Erweiterung wird durch einen Pfeil mit dreieckiger Pfeilspitze dargestellt.



Mehrfachvererbung ist diagrammatisch leicht darstellbar. Auf die damit zusammenhängenden Probleme sei hier nur hingewiesen.

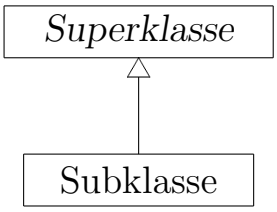


Betrachten wir Erweiterungen als Teilmengen, dann stellt sich die Frage, ob die Teilmengen **disjunkt** ***(disjoint)*** oder **überlappend** ***(overlapping)*** sind. Man spricht von der **Einschränkung durch Disjunktheit** ***(disjointness constraint)***. In der Darstellung ist *disjoint* Default.



Eine weitere Einschränkung der Erweiterung bzw Spezialisierung/Generalisierung ist die der **Vollständigkeit** ***(completeness constraint)***, die **total** oder **partiell** sein kann. Bei der totalen Erweiterung muss jede Entität der Superklasse auch Entität einer Subklasse sein. Die Superklasse ist dann eine **abstrakte** ***(abstract)*** Klasse, d.h. von ihr können keine Objekte angelegt werden. Eine Generalisierung ist üblicherweise total, da die Superklasse aus den Gemeinsamkeiten der Subklassen konstruiert wird.

Eine abstrakte Klasse wird mit **kursivem** Klassennamen dargestellt:



Die abtrakten (nicht implementierten) Methoden einer abtrakten Klasse werden auch kursiv dargestellt.

Da jeder Untertyp auch seinerseits Untertypen haben kann, führt das zu **Spezialisierungshierarchien** und zu **Spezialisierungsnetzen** ***(specialization lattice)*** bei Untertypen, die mehr als einen Obertyp haben **(Mehrfachvererbung,** ***multiple inheritance, shared subclass)***.

Generalisierung und Spezialisierung sind nur verschiedene Sichten auf dieselbe Beziehung. Als Entwurfs-Prozesse sind das allerdings unterschiedliche Methoden:

- Spezialisierung bzw Erweiterung: Wir beginnen mit einem Entitätstyp und definieren dann Untertypen durch sukzessive Spezialisierung bzw Erweiterung. Man spricht auch von einem ***top-down conceptual refinement***.

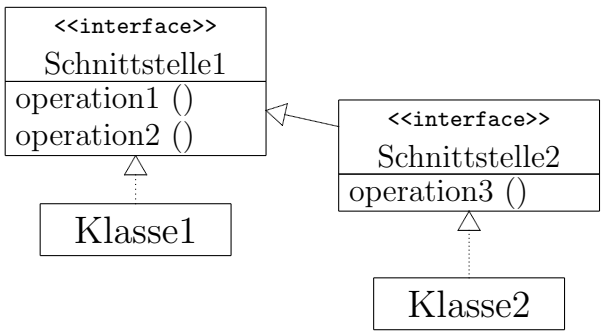
- Generalisierung: Der umgekehrte Weg beginnt bei sehr speziellen Untertypen, aus denen durch sukzessive Generalisierung eine Obertypenhierarchie bzw ein Obertypennetz gebildet wird. Man spricht von ***bottom-up conceptual synthesis***.

In der Praxis wird man irgendwo in der Mitte beginnen und beide Methoden anwenden.

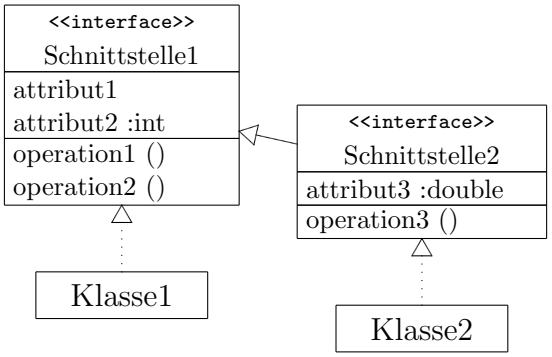
5.7 Realisierung

Die Beziehung **Realisierung** ***(realization)*** ist die Beziehung zwischen einer **Spezifikation** ***(specification)*** oder einem **Spezifikator** ***(specifier)*** und einer **Implementierung** ***(implementation)*** oder **Implementor** ***(implementor)***. Meistens ist das die Beziehung zwischen einer Schnittstelle und einer sie realisierende Klasse. Die Realisierung wird durch einen gestrichelten Erweiterungspfeil dargestellt.

Eine **Schnittstelle** ***(interface)*** ist eine total abstrakte Klasse. Alle ihre Operationen sind abstrakt. Eine Schnittstelle kann andere Schnittstellen erweitern.



Seit UML 2 kann eine Schnittstelle auch Attribute haben:

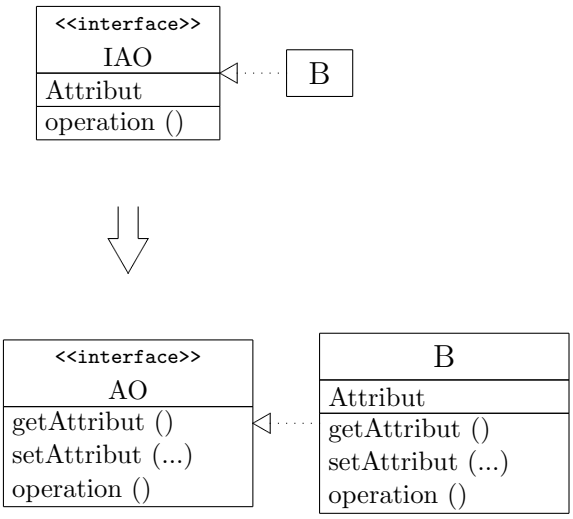


Das bedeutet, dass eine Schnittstelle auch aktiv Beziehungen zu anderen Schnittstellen und Klassen aufnehmen kann. Damit kann man dann den Klassenentwurf sehr abstrakt halten.

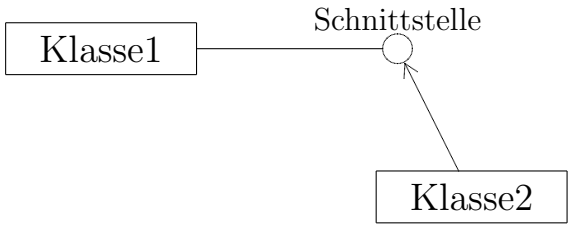
Eine Klasse kann mehr als eine Schnittstelle mit Attributen realisieren. Man hat dann natürlich die Probleme der Mehrfachvererbung.

In C++ kann eine Schnittstelle mit Attributen einfach als abstrakte Klasse realisiert werden.

In Java müsste man die Attribute durch die entsprechenden set/get-Methoden repräsentieren. Die eigentlichen Attribute müssten in der realisierenden Klasse Aufgenommen werden.



Klassen können Schnittstellen anbieten, die dann von anderen Klassen benutzt werden. Die Benutzung einer Schnittstelle wird auch durch eine Socket-Notation dargestellt.



5.8 Assoziation

Wenn eine Beziehung zwischen Klassen nicht eine der in den vorhergehenden Abschnitten diskutierten Beziehungen ist (Attribut, Komposition, Aggregation, Benutzung, Erweiterung, Realisierung), dann kann sie als Assoziation ***(association)*** modelliert werden. Die Assoziation ist die allgemeinste Form einer Beziehung, die bestimmte Objekte semantisch eingehen.

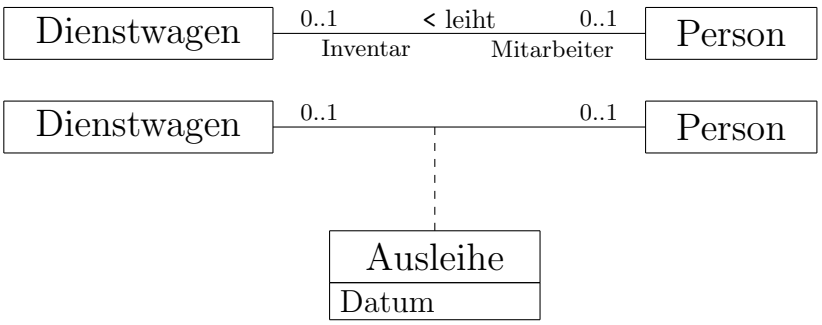
Die an einer Beziehung beteiligten Entitäten heißen **Teilnehmer** ***(participants)*** einer Beziehung. Die Anzahl der Teilnehmer ist der **Grad** ***(degree)*** der Beziehung. Je nach Grad kann die Beziehung **binär, ternär** oder n**-wertig** ***(binary, ternary, n-ary)*** sein.

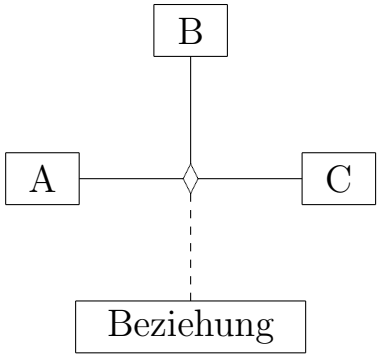
Von der Einführung von mehr als zweiwertigen Beziehungen ist abzuraten. Es sollte versucht werden, diese Beziehungen durch mehrere Zweier-Beziehungen darzustellen. Dadurch werden von vornherein Redundanzen vermieden. Eine n-wertige Beziehung kann immer durch eine Entität dargestellt werden, die binäre Beziehungen zu den n beteiligten Entitäten unterhält.

Zweiwertige Beziehungen werden durch Linien dargestellt, mehrwertige Beziehungen durch eine zentrale Raute. Dabei können Rollen und Multiplizitäten angegeben werden. Der Name der Beziehung steht – eventuell mit einer "Leserichtung" versehen – an der Linie oder in einem eigenen Rechteck, das wie eine Klasse mit Attributen und Operationen notiert werden kann (sogenannte **Assoziationsklasse**).

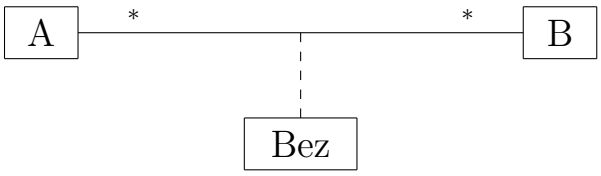
Bei einer mehrwertigen Beziehung müssen im Prinzip die Multiplizitäten zwischen je zwei beteiligten Partnern geklärt werden. Bei einer ternären Beziehung ist das graphisch noch vernünftig machbar.

Assoziationen können Navigationspfeile haben. (Bemerkung: Oft werden bei einer Assoziation ohne Pfeile die Navigationspfeile auf beiden Seiten impliziert und entspricht damit der ***Relationship*** des Modells der ODMG. Es ist selten, dass eine Assoziation ohne Navigation existiert.) Wenn eine Navigationsrichtung ausgeschlossen werden soll, dann wird dies mit einem Kreuz "$\times$" auf der entsprechenden Seite notiert.





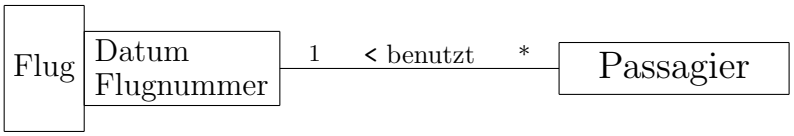
Da Assoziationen meistens als eigene Klassen implementiert werden, wird das oft schon durch eine **reifizierte** (verdinglichte, konkretisierte, ***reified***) Darstellung ausgedrückt. Anstatt von



wird

notiert.

Ein **Assoziations-Endpunkt** ***(association end)*** kann außer einem Rollennamen und einer Multipliziert auch noch einen **Qualifikator** ***(qualifier)*** haben, der eine Teilmenge von Objekten einer Klasse bestimmt, meistens genau ein Objekt.



Normalerweise wäre die Beziehung zwischen `Passagier` und `Flug` eine Many-to-Many-Beziehung. Hier drücken wir aus, dass, wenn `Datum` und `Flugnummer` gegeben sind, es für jeden Passagier genau einen Flug gibt.

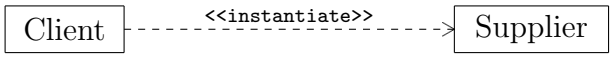
Bemerkung: Beziehungen zwischen Objekten sollten in UML-Diagrammen i.a. durch Assoziationen (also irgendwelche Linien zwischen Klassen) und nicht nur durch Attribute dargestellt werden, obwohl bei den meisten Programmiersprachen die Beziehungen zu Klassenelementen werden. Eine Assoziation macht aber deutlich, dass eventuell eine referentielle Integrität zu beachten ist, d.h. eine Abhängigkeit zwischen Objekten zu verwalten ist.

5.9 Abhängigkeit

Mit der Abhängigkeit ***(dependency)*** können Beziehungen allgemeiner Art zwischen einem Client und Supplier dargestellt werden.

Dargestellt wird die Abhängigkeit durch einen gestrichelten Pfeil vom Client zum Supplier mit stereotypem Schlüsselwort ***(keyword)***. Die Pfeilrichtung ist oft nicht sehr intuitiv. Dabei hilft folgende Überlegung (Navigation): Welche Seite kennt die andere Seite? **Die Client-Seite kennt die Supplier-Seite.**

Der folgende Konstrukt



bedeutet, dass ein Client-Objekt ein Supplier-Objekt erzeugt oder instanziiert. Die Supplier-Klasse stellt ihr Objekt zur Verfügung.

Folgende Abhängigkeiten sind in UML definiert:

access: `<<access>>`
: Ein **Paket** ***(package)*** kann auf den Inhalt eines anderen (Ziel-)Pakets zugreifen.

binding: `<<bind>>`
: Zuweisung von Parametern eines Templates, um ein neues (Ziel-)Modell-Element zu erzeugen.

call: `<<call>>`
: Die Methode einer Klasse ruft eine Methode einer anderen (Ziel-)Klasse auf.

derivation: `<<derive>>`
: Eine Instanz kann aus einer anderen (Ziel-)Instanz berechnet werden.

friend: `<<friend>>`
: Erlaubnis des Zugriffs auf Elemente einer (Ziel-)Klasse unabhängig von deren Sichtbarkeit.

import: `<<import>>`
: Ein Paket kann auf den Inhalt eines anderen (Ziel-)Pakets zugreifen. Aliase können dem Namensraum des Importeurs hinzugefügt werden.

instantiation: `<<instantiate>>`
: Eine Methode einer Klasse kann ein Objekt einer anderen (Ziel-)Klasse erzeugen. Die (Ziel-)Klasse stellt als Supplier das neue Objekt zur Verfügung.

parameter: `<<parameter>>`
: Beziehung zwischen einer Operation und ihren (Ziel-)Parametern.

realization: `<<realize>>`
: Beziehung zwischen einer (Ziel-)Spezifikation und einer Realisierung.

refinement: `<<refine>>`
: Beziehung zwischen zwei Elementen auf verschiedenen semantischen Stufen. Die allgemeinere Stufe ist das Ziel.

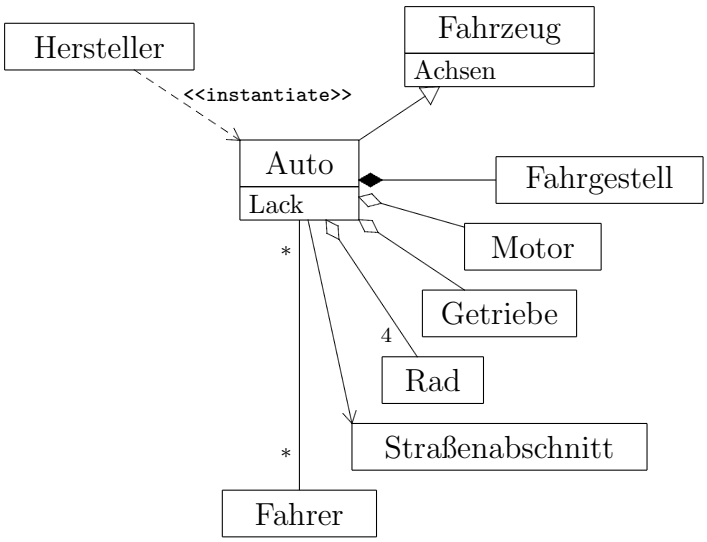
send: `<<send>>`
Beziehung zwischen Sender und Empfänger einer Nachricht.

trace: `<<trace>>`
Es besteht eine gewisse – etwa parallele – Beziehung zwischen Elementen in verschiedenen Modellen.

usage: `<<use>>`
Ein Element setzt das Vorhandensein eines anderen (Ziel-)Elements für seine korrekte Funktionsfähigkeit voraus.

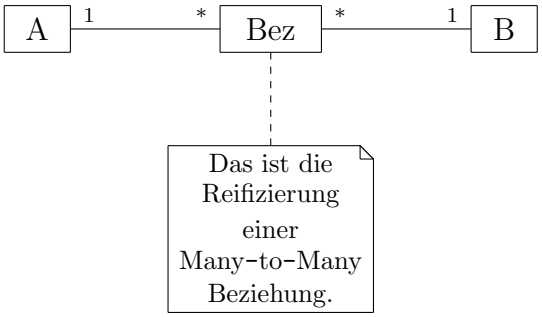
5.10 Zusammenfassung der Beziehungen

Folgendes Beispiel soll nochmal die Unterschiede zwischen Erweiterung, Attribut, Komposition, Aggregation, Benutzung, Abhängigkeit und Assoziation zeigen:



5.11 Notiz

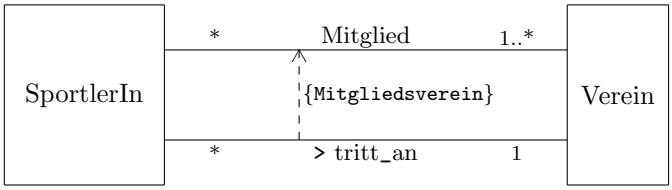
Mit dem Konstrukt der Notiz ***(note)*** können Kommentare, Bemerkungen oder andere textuelle Informationen im Diagramm untergebracht werden. Oft werden in einer Notiz Hinweise zur Implementierung einer Methode gegeben. Der Text wird in einem Rechteck mit Eselsohr dargestellt. Eine gestrichelte Linie gibt an, worauf sich die Notiz bezieht.

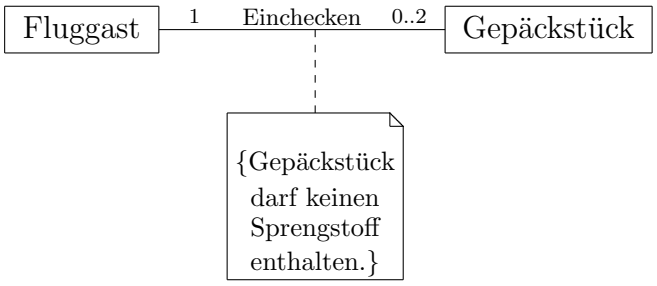


5.12 Einschränkung

Eine Einschränkung ***(constraint)*** ist ein Text in einer natürlichen oder formalen Sprache in geschweiften Klammern. Der Text kann auch in einer Notiz stehen (mit oder ohne geschweifte Klammern).

Dieser Text wird mit einer gestrichelten Linie oder einem gestrichelten Pfeil mit dem Element verbunden, für das die Einschränkung gilt. Wenn ein Pfeil verwendet wird, dann soll der Pfeil vom eingeschränkten zum einschränkenden Element zeigen.





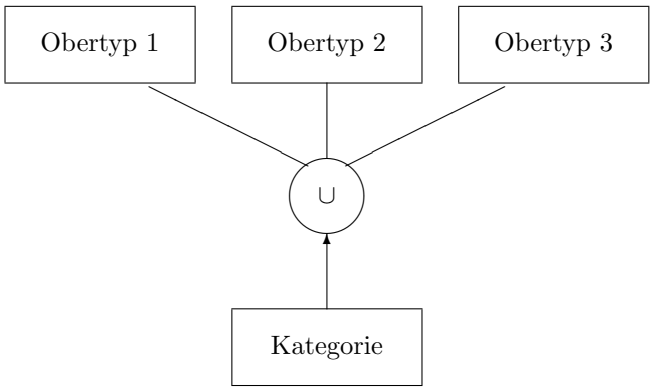
5.12.1 Kategorie *(category)*

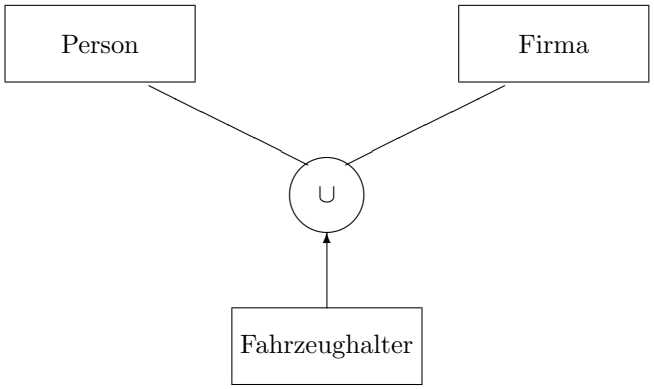
Ein Fahrzeughalter kann eine Person oder eine Firma sein. Da nicht jede Person und nicht jede Firma ein Fahrzeughalter ist, kann Fahrzeughalter kein Obertyp sein.

Fahrzeughalter kann auch kein Untertyp sein. Denn von wem sollte Fahrzeughalter ein Untertyp sein – Person oder Firma?

Fahrzeughalter ist ein Beispiel für eine **Kategorie**, die von verschiedenen Obertypen **exklusiv** erben kann. Mit "exklusiv" ist gemeint, dass die Entität Fahrzeughalter nur von genau einem Obertyp erben kann, dass sie nicht die Kombination mehrerer Obertypen ist, wie das bei Mehrfachvererbung der Fall ist. Eine Kategorie erbt *einen* Obertyp aus einer Vereinigung von Obertypen.

Die graphische Darstellung sieht folgendermaßen aus, wobei der Kreis ein Vereinigungszeichen enthält.

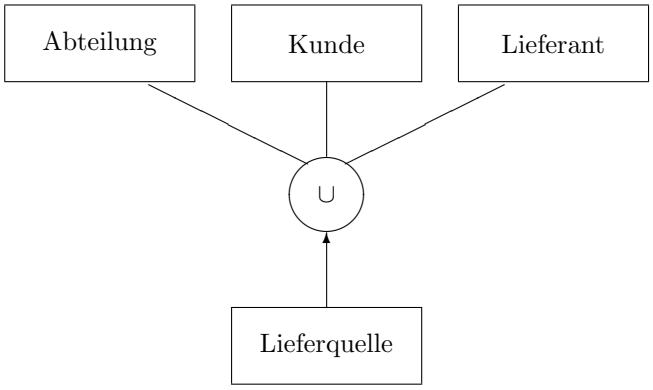




Eine Kategorie kann **partiell** oder **total** sein. Totalität bedeutet, dass jeder Obertyp der Kategorie zur Kategorie gehören muss. In diesem Fall ist eine Darstellung als Obertyp-Untertyp-Beziehung auch möglich, wobei die Kategorie der Obertyp ist, und ist meistens vorzuziehen, insbesondere wenn die Entitäten viele Attribute teilen.

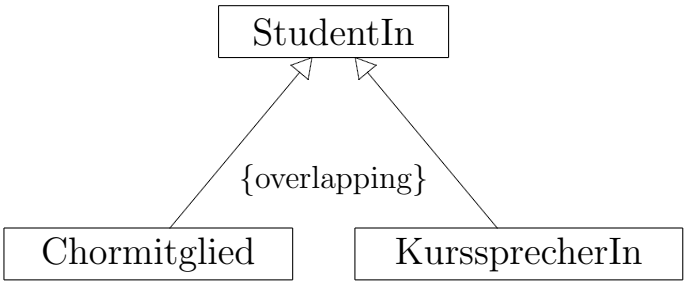
Kategorien kommen häufig vor. Sie sind eine Art von Rolle, die eine Entität spielen kann. Insbesondere Personen sind in den verschiedensten Rollen zu finden.

Beispiel:

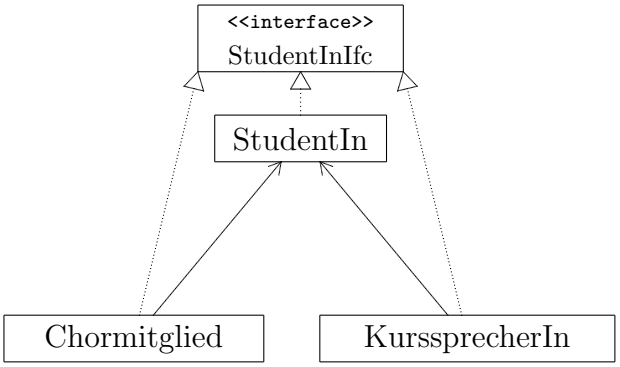


5.13 Beispiele

5.13.1 Nichtdisjunkte Untertypen



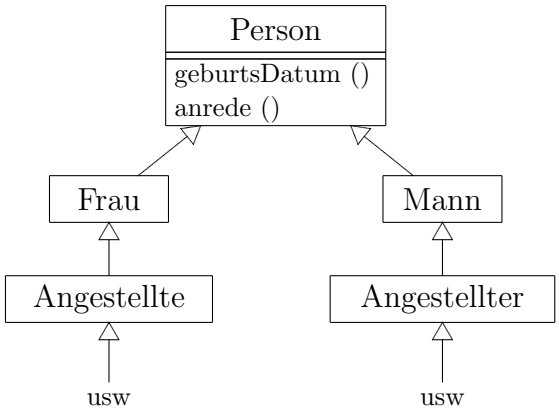
Wie wird das implementiert? Als zwei Kategorien (Chormitglied und Kursprecher):



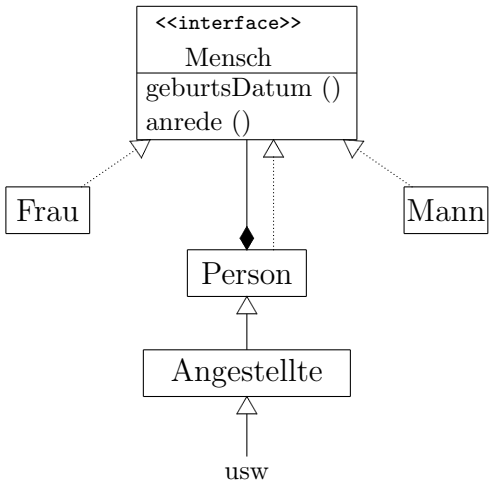
Sollte die Uses-Beziehung nicht besser zur Schnittstelle gehen? Im allgemeinen nein, denn es könnte ja sein, dass das Chormitglied oder der Kurssprecher zur Realisierung der Schnittstelle Methoden oder Datenelemente benötigt, die die Schnittstelle nicht anbietet.

5.13.2 Weiblich – Männlich

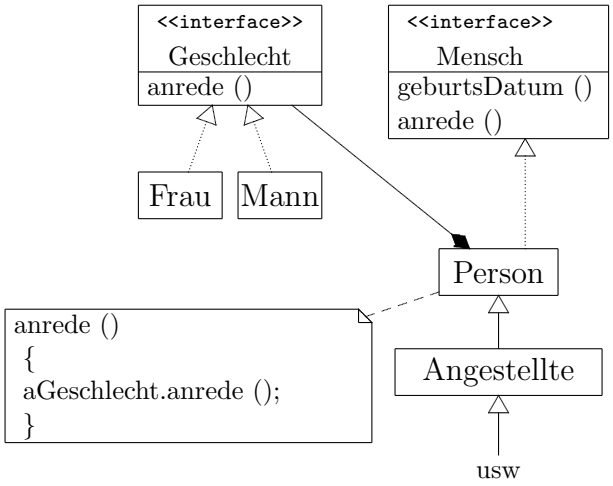
Wenn Frauen und Männer im wesentlichen gleich behandelt werden, dann ist folgende Struktur nicht günstig, weil das zu zwei redundanten Vererbungshierarchien führt:



Stattdessen sollte man eine "Person" als Kategorie einführen:



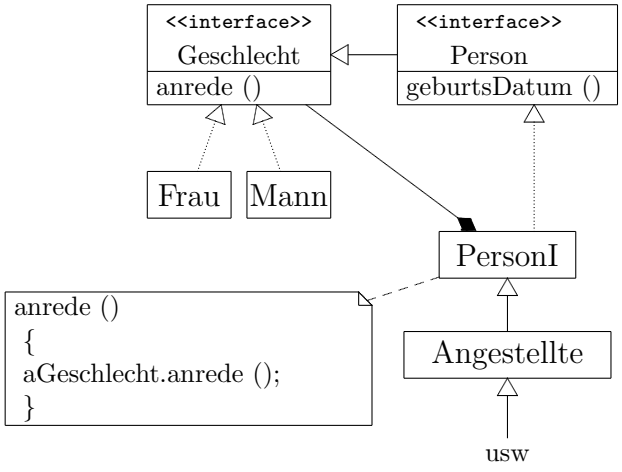
Um zu verhindern, dass sich `Person` wieder eine `Person` als Komponente nimmt, wäre folgende Struktur besser:



Übung: Diskutieren Sie die unterschiedlichen Optionen (Komposition, Aggregation, Benutzung) für die Beziehung zwischen `Person` und `Geschlecht`.

Kategorien haben eine große Ähnlichkeit zum Bridge-Pattern.

Um die "anrede"-Redundanz zu entfernen, wäre noch schöner:



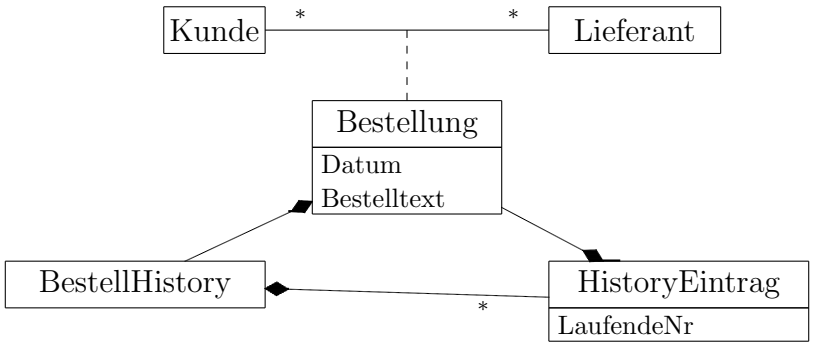
Allerdings kann eine `Person` wieder eine `Person` als Komponente erhalten, was wir aber in Kauf nehmen.

Das ist übrigens eine treue Abbildung der – eventuell in einer Problembeschreibung erscheinenden – Sätze *"Ein Angestellter ist eine Person. Eine Person ist ein Mensch und hat ein Geschlecht."*

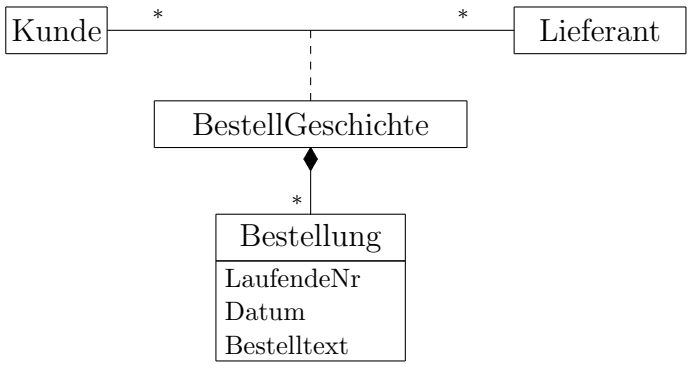
Übung: Diskutieren Sie die unterschiedlichen Optionen (Komposition, Aggregation, Benutzung) für die Beziehung zwischen `Person` und `Geschlecht`.

5.13.3 Many-to-Many-Beziehung mit History

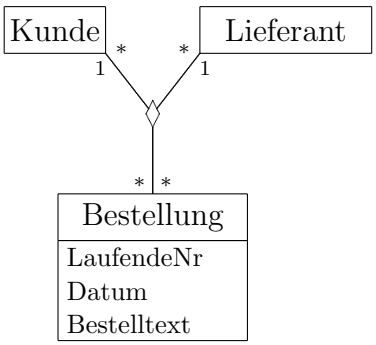
Viele Many-to-Many-Beziehungen zwischen zwei Partnern werden nicht nur einmal, sondern mehrmals eingegangen. Z.B. kann eine Person dasselbe Buch mehrmals leihen. Oder ein Kunde kann bei einer Firma mehrere Bestellungen machen. Um all diese Vorgänge zu verwalten, muss eine sogenannte ***History*** oder "Geschichte" für ein Beziehungsobjekt angelegt werden. Dabei mag es sinnvoll sein, den verschiedenen Vorgängen eine laufende Nummer zu geben. Das könnte folgendermaßen modelliert werden:



Hier wurde darauf geachtet, dass die ursprüngliche Struktur möglichst erhalten bleibt. Das macht es aber ziemlich umständlich. Wahrscheinlich würde man folgende einfachere Struktur vorziehen, die sich auch wesentlich bequemer in ein RDB übersetzen lässt.



Wir können das Ganze auch als eine ternäre Beziehung betrachten, wobei wir ein gutes Beispiel für die unterschiedlichen Multiplizitäten zeigen können:


Kunde
Lieferant
*
1
*
1
* *
Bestellung
LaufendeNr
Datum
Bestelltext

Kapitel 6

Speicherstrukturen

Daten müssen auf einem **Speichermedium** ***(storage medium)*** physisch gespeichert werden. In diesem Kapitel wird die physische Organisation **(Dateiverwaltung)** der gespeicherten Daten besprochen.

Man spricht von einer **Speicherhierarchie**:

- **Primärspeicher** ist ein Speichermedium, mit dem die CPU direkt arbeiten kann. Das sind im wesentlichen der Hauptspeicher (RAM *(random access memory)*) (Zugriffszeit etwa 100 ns) und gewisse schnellere, aber kleinere Cache-Speicher (Zugriffszeit etwa 10 ns). Der Primärspeicher bietet einen schnellen Zugriff auf Daten, hat aber eine begrenzte Größe und ist relativ teuer. Die Lebenszeit der Daten ist begrenzt **(flüchtiger Speicher**, ***volatile storage)***.

- **Sekundärspeicher** oder **Hintergrundspeicher** sind magnetische und optische Platten mit großer Kapazität. Die Kosten für den Speicher sind vergleichsweise gering, d.h. um ein, zwei Größenordnungen kleiner als beim Primärspeicher. Der Zugriff ist wesentlich langsamer (**Zugriffslücke** ***(access gap)*** Größenordnung 10^5, Zugriffszeit etwa 10 ms). Die CPU kann nicht direkt auf diesen Daten arbeiten. Die Daten müssen erst in den Primärspeicher kopiert werden. Die Lebenszeit der Daten gilt als unbegrenzt **(nichtflüchtiger Speicher**, ***non-volatile storage)***.

  Die folgende Diskussion bezieht sich auf Magnetplatten – kurz Platten – als das zur Zeit üblichste sekundäre Speichermedium. Wegen ihrer hohen Zugriffsgeschwindigkeit gehören sie zu den **on-line**-Geräten.

- **Tertiärspeicher** oder **Archivspeicher** sind meistens Magnet-Bänder, die sehr billig (Pfennige pro MB) sind und eine hohe Ausfallsicherheit bieten. Der Zugriff auf Bänder ist sehr langsam. Sie werden für Back-up benutzt. Die Daten sind dann **off-line** gespeichert. CDs und DVDs werden auch als Tertiärspeicher verwendet.

  Man unterscheidet noch die Begriffe **Nearline-Tertiärspeicher**, bei dem die Speichermedien automatisch bereitgestellt werden, und **Offline-Tertiärspeicher**, bei dem die Speichermedien von Hand bereitgestellt werden.

Der DBA muss physische Speichermethoden kennen, um den Prozess des **physischen Datenbankdesigns** durchführen zu können. Normalerweise bietet das DBMS dazu verschiedene Möglichkeiten an.

Typische DB-Anwendungen benötigen nur einen kleinen Teil der DB. Dieser Teil muss auf der Platte gefunden werden, in den Primärspeicher kopiert werden und nach der Bearbeitung wieder auf die Platte zurückgeschrieben werden.

Die Daten auf der Platte sind in **Dateien** ***(file)*** von **Datensätzen** ***(record)*** organisiert. Jeder Datensatz ist eine Menge von Datenwerten, die Tatsachen über Entitäten, ihre Attribute und Beziehungen enthalten.

Es gibt verschiedene **primäre Dateiorganisationen** ***(primary file organization)***, nach denen die Datensätze in der Datei physisch auf die Platte geschrieben werden:

- **Heapdatei** ***(heap file, unordered file)***: Die Datensätze werden ungeordnet auf die Platte geschrieben.
- **Sortierte oder sequentielle Datei** ***(sorted file, sequential file)***: Die Datensätze werden nach einem besonderen Feld geordnet auf die Platte geschrieben.
- **Hash-Datei** ***(hash file)***: Eine Hashfunktion wird benutzt, um die Datensätze in der Datei zu platzieren.
- Sonstige Strukturen wie z.B. Bäume.

6.1 Speichermedien

6.1.1 Magnetplatten

Die Informationseinheit auf der **Platte** ***(disk)*** ist ein Bit. Durch bestimmte Magnetisierung eines Flächenstücks auf der Platte kann ein Bitwert 0 oder 1 dargestellt werden. Normalerweise werden 8 Bit zu einem Byte gruppiert. Die Kapazität der Platte ist die Anzahl Byte, die man auf seiner Fläche unterbringen kann. Typische Kapazitäten sind einige Gigabyte (GB).

Eine "Platte" besteht i.a. aus mehreren **einseitigen** oder **doppelseitigen** Platten ***(platter)*** und bildet einen **Plattenstapel** ***(diskpack)***. Die einzelnen Platten haben Oberflächen ***(surface)***. Die Information auf den Platten ist in konzentrischen Kreisen **(Spur**, ***track)*** gespeichert. Spuren mit demselben Durchmesser bilden einen **Zylinder** ***(cylinder)***. Wenn Daten auf demselben Zylinder gespeichert sind, können sie schneller geholt werden, als wenn sie auf verschiedenen Zylindern liegen.

Jede Spur speichert dieselbe Menge von Information, sodass auf den inneren Spuren die Bit dichter liegen. Die Anzahl der Spuren liegt in der Größenordnung 1000.

Die Unterteilung der Spuren in **Sektoren** ***(sector)*** ist auf der Plattenoberfläche hart kodiert und kann nicht geändert werden. Nicht alle Platten haben eine Unterteilung in Sektoren.

Die Unterteilung einer Spur in **Blöcke** ***(block)*** oder **Seiten** ***(page)*** wird vom Betriebssystem während der **Formatierung** oder **Initialisierung** der Platte vorgenommen. Die Blockgröße (512 bis 4096 Byte) kann dynamisch nicht geändert werden. Die Blöcke werden durch ***interblock gaps***

getrennt. Diese enthalten speziell kodierte Steuerinformation, die während der Formatierung geschrieben wird.

Eine Platte ist ein ***random access*** Medium. D.h. auf jeden Block kann direkt zugegriffen werden, ohne dass davorliegende Blöcke gelesen werden müssen. Die Transfereinheit zwischen Platte und Hauptspeicher ist der Block. Die Adresse eines Blocks setzt sich zusammen aus Blocknummer, Sektornummer, Spurnummer und Plattenseitennummer.

Bei einem Lesekommando wird ein Block in einen speziellen Hauptspeicherpuffer kopiert. Bei einem Schreibkommando wird dieser Puffer auf einen Plattenblock kopiert. Manchmal werden mehrere hintereinanderliegende Blöcke, sogenannte **Cluster**, als Einheit kopiert.

Der eigentliche Hardware-Mechanismus ist der auf einen mechanischen Arm montierte Lese-Schreib-Kopf des **Plattenlaufwerks** ***(disk drive)***. Der mechanische Arm kann radial bewegt werden. Bei Plattenstapeln ist das ein Zugriffskamm mit entsprechend vielen Köpfen.

Um einen Block zu transferieren sind drei Arbeitsschritte notwendig. Zunächst muss der Kopf auf die richtige Spur bewegt werden. Die benötigte Zeit ist die **Suchzeit, Zugriffbewegungszeit** ***(seek time)***. Danach muss gewartet werden, bis sich die Platte an die richtige Stelle gedreht hat: **Latenzzeit, Umdrehungszeit, Umdrehungswartezeit** ***(latency, rotational delay, rotational latency)***. Schließlich werden die Daten während der **Transferzeit, Übertragungszeit** oder **Lesezeit** ***(block transfer time)*** in den oder vom Hauptspeicher transferiert.

Als Größenordnung für Suchzeit und Latenzzeit kann man 10 ms angeben, für die Transferzeit 0,01 ms. Daher wird man immer versuchen viele hintereinanderliegende Blöcke zu lesen. Diese Zeiten sind groß verglichen mit CPU-Verarbeitungszeiten und bilden daher immer einen Flaschenhals. Geeignete Dateistrukturen versuchen, die Anzahl der Blocktransfers zu minimieren.

Für das Disk-I/O ist entscheidend, ob Informationen auf der Platte "hintereinander" liegen und daher mit minimaler Such- und Latenzzeit zugänglich sind ***(sequential I/O, serial I/O, in disk order)*** oder ob die Information auf der Platte verstreut ist und jedesmal eine Kopfbewegung und Latenzzeit investiert werden muss ***(nonsequential I/O)***.

6.1.2 Speicherarrays: RAID

Die für die mechanischen Arbeitsvorgänge einer Festplatte benötigten Zeiten lassen sich nicht mehr wesentlich reduzieren. Die RAID-Technologie ***(redundant array of inexpensive*** – neuerdings – ***independent disks)*** versucht dieses Problem zu lösen, indem viele, relativ billige Laufwerke benutzt werden. Der RAID-***Controller*** sorgt dafür, dass sich die vielen Laufwerke nach außen transparent wie *ein* Laufwerk verhalten. Daher müssen wir hier nicht auf die Einzelheiten dieser Technologie eingehen [26].

6.1.3 Magnetbänder

Magnetbänder sind ***sequential access*** Medien. Um auf den n-ten Block zuzugreifen, muss man erst n – 1 Blöcke lesen. Normalerweise wird ein Byte quer über das Band gespeichert. Die Byte werden dann hintereinander gespeichert. Banddichten liegen zwischen 600 und 3000 Byte pro cm. Blöcke sind gewöhnlich größer als bei Platten um Zwischenräume einzusparen.

Zugriffszeiten sind extrem lang. Daher werden Bänder nur für Back-up verwendet.

6.2 Datenbankpufferung

Die Organisationseinheit des Speichermediums ist der **(physische) Block** ***((physical) block)***. Das Dateisystem des Betriebssystems oder des DBMSs fordert **Seiten** ***(page)*** an. Die Zuordnung zwischen Seite und Block wird mit festen Faktoren vorgenommen: 1, 2, 4 oder 8 Blöcke einer Spur werden auf eine Seite abgebildet. Ohne Beschränkung der Allgemeinheit können wir die Zuordnung "ein Block – eine Seite" wählen und werden im Folgenden diese Begriffe synonym verwenden.

Wenn viele Blöcke benötigt werden, dann kann man im Hauptspeicher spezielle Puffer **(Datenbankpuffer)** reservieren, um den Transfer insgesamt durch parallele Bearbeitung der Daten zu beschleunigen. Außerdem ist es sinnvoll, Blöcke länger als nur für eine Operation im Hauptspeicher zu halten. Anwendungen zeigen oft eine sogenannte **Lokalität**, indem sie mehrmals hintereinander auf dieselben Daten zugreifen.

Der Datenbankpuffer hat eine begrenzte Größe. Daher müssen Blöcke immer wieder ersetzt werden. Dabei werden verschiedene **Ersetzungsstrategien** verwendet, z.B. wird der am längsten nicht mehr gebrauchte Block ersetzt.

Der Datenbankpuffer ist in **Pufferrahmen** gegliedert. Ein Pufferrahmen kann eine Seite zwischenspeichern. Die Pufferverwaltung hat im einzelnen folgende Bereiche:

Suchverfahren: Werden vom DBMS Seiten angefordert, dann werden diese zunächst im Puffer gesucht. Es müssen also effiziente Suchverfahren zur Verfügung stehen. Wenn eine direkte Suche (Überprüfung jedes Pufferrahmens) nicht genügend effizient ist, dann müssen Zusatzstrukturen verwendet werden wie unsortierte oder (nach Seitennummer) sortierte Tabellen oder Listen oder Hash-Tabellen, in denen alle Seiten des Datenbankpuffers vermerkt sind.

Speicherzuteilung: Wenn DB-Transaktionen parallel Seiten anfordern, dann muss der Puffer unter den Transaktionen geschickt aufgeteilt werden. Dafür gibt es verschiedene Strategien:

1. **Lokale Strategie:** Der Puffer wird statisch oder dynamisch in disjunkte Bereiche aufgeteilt, die jeweils einer Transaktion zur Verfügung gestellt werden.
2. **Globale Strategie:** Bereiche für bestimmte Transaktionen sind nicht erkennbar. Jeder Transaktion steht der gesamte Puffer zur Verfügung. Gemeinsam referenzierte Seiten können so besser berücksichtigt werden.
3. **Seitentypbezogene Strategie:** Der Puffer wird in Bereiche für Datenseiten, Zugriffspfadseiten, Data-Dictionary-Seiten oder weitere Seitentypen eingeteilt. Jeder Bereich kann dann sein eigenes Seitenersetzungsverfahren haben.

Seitenersetzungsstrategie: Ist der Puffer voll, so muss entschieden werden, welche Seite **verdrängt**, d.h. nach eventuellem Zurückschreiben auf die Platte überschrieben werden soll. Da dies die wichtigste Aufgabe der Pufferverwaltung ist, werden wir uns dies im folgenden Abschnitt genauer ansehen.

6.2.1 Seitenersetzungsstrategien

Beim Laden von Seiten von der Platte unterscheiden wir zwei Begriffe:

- ***Demand-paging-***Verfahren: Nur die angeforderte Seite wird in den Puffer geladen. Das ist das Standardverfahren.
- ***Prepaging-***Verfahren (oder *read-ahead processing*): Neben der angeforderten Seite werden auch weitere Seiten in den Puffer geladen, die eventuell in nächster Zukunft benötigt werden. Das ist insbesondere bei großen zusammenhängenden Objekten vorteilhaft.

Eine gute Seitenersetzungsstrategie ersetzt die Seiten, die am ehesten in Zukunft nicht mehr gebraucht werden. Die **optimale Strategie** würde also wissen, welche Zugriffe in Zukunft anstehen. Das ist natürlich nicht realisierbar. **Realisierbare** Verfahren besitzen keine Kenntnisse über das zukünftige Referenzverhalten.

Die Effizienz einer einfachen Zufallsstrategie sollte von jedem realisierbaren Verfahren erreicht werden. Gute realisierbare Verfahren benutzen das vergangene Referenzverhalten.

Messbar sind das **Alter** einer Seite im Puffer und die **Anzahl der Referenzen** auf eine Seite im Puffer. Im Einzelnen ist messbar:

- **AG:** Alter einer Seite seit Einlagerung (globale Strategie)
- **AJ:** Alter einer Seite seit dem letzten Referenzzeitpunkt (Strategie des jüngsten Verhaltens)
- **RG:** Anzahl aller Referenzen auf eine Seite (globale Strategie)
- **RJ:** Anzahl nur der letzten Referenzen auf eine Seite (Strategie des jüngsten Verhaltens)

Im folgenden stellen wir einige konkret verwendete Strategien vor:

Zufall: Die zu ersetzende Seite wird erwürfelt. Diese Strategie ist wegen der steigende Größe des DB-Puffers nicht zu verachten und wird daher insbesondere bei Betriebssystemen vermehrt eingesetzt.

FIFO (AG, *first-in-first-out***):** Die Seite (älteste Seite), die am längsten im Puffer steht, wird ersetzt. Bei einem sequentiellen Durchlauf durch eine Datei ist dieses Verfahren günstig, sonst eher schlecht.

LFU (RG, *least-frequently-used***):** Die Seite, die am seltensten referenziert wurde, wird ersetzt. Da dieses Verfahren vom Alter unabhängig ist, kann es sein, dass eine Seite, die vor langer Seite sehr häufig benutzt wurde, nicht mehr aus dem Puffer zu verdrängen ist.

LRU (AJ, RJ, *least-recently-used***):** Die Seite, die am längsten nicht mehr referenziert wurde, wird ersetzt. Das Referenzverhalten in jüngster Zeit wird berücksichtigt. Diese Strategie ist bei Betriebssystemen sehr beliebt, bei Datenbanksystemen in dieser reinen Form allerdings nicht optimal.

Eine *self-tuning* Variante ist LRU-K, wobei die Zeiten der letzten K Zugriffe auf eine Seite berücksichtigt werden.

DGCLOCK (AG, RJ, RG, *dynamic-generalized-clock***)** und

LRD (AJ, AG, RG, *least-reference-density*): Die Kriterien Alter und Referenzhäufigkeit werden kombiniert, wobei durch Gewichtung der Referenzzähler das jüngste Verhalten höher bewertet wird.

Das Verfahren basiert auf einem globalen Referenzzähler für den gesamten Puffer (`GLOB`) und auf einem lokalen Referenzzähler für jeden Pufferrahmen j ($\texttt{LOK}_j$). Diese Zähler werden folgendermaßen verwaltet:

1. Beim Laden einer Seite in den Pufferrahmen j findet eine Neubestimmung des lokalen Referenzzählers statt:

   $\texttt{LOK}_j = \texttt{GLOB}$

2. Bei jeder Referenz auf eine Seite werden die Zähler aktualisiert:

   $\texttt{GLOB} = \texttt{GLOB} + 1$
   $\texttt{LOK}_j = \texttt{LOK}_j + \texttt{GLOB}$

3. Erreicht `GLOB` eine festgelegte Obergrenze, wird `GLOB` auf einen vorgegebene Untergrenze `MINGLOB` gesetzt und die lokalen Zähler werden durch Division mit einer `KONSTANTE`n normiert:

   $\texttt{GLOB} = \texttt{MINGLOB}$
   $\texttt{LOK}_j = \texttt{LOK}_j \texttt{ div KONSTANTE}$ (j = 1, ... n)

4. Schließlich können zum Feintuning von außen Seitengewichte gesetzt oder Seiten festgesetzt werden.

Auf die eher marginalen Unterschiede zwischen DGCLOCK und LRD wird hier nicht eingegangen.

Die mangelnde Eignung des Betriebssystem-Puffers für Datenbankoperationen soll an einer einfachen *nested-loop*-Verbundoperation zwischen zwei Tabellen A und B gezeigt werden. Beide Tabellen bestehen auf dem Sekundärspeicher aus einer Folge von Seiten A_i bzw B_i. Wir laden zunächst Seite A_1 und lesen dort das erste Tupel. Damit referenzieren wird die Seite A_1. Das erste Tupel wird mit allen Tupeln von B verglichen. Dazu laden wird nacheinander die Seiten B_1 usw. Nach Seite B_5 sei der Puffer voll. Für die Seite B_6 muss nun eine Seite verdrängt werden.

- Bei FIFO würde A_1 verdrängt werden, die aber weiterhin benötigt wird, nämlich um das zweite, dritte usw. Tupel zu vergleichen.
- Bei LRU wird ebenfalls A_1 verdrängt, da sie nur beim Lesen des ersten Tupels referenziert wurde. Sie wird aber für die weiteren Tupel noch benötigt. Wenn A_1 wegen des nächsten Tupels wieder geladen wird, würde ausgerechnet Seite B_1 verdrängt werden, die als nächste aber benötigt wird. Bei FIFO und LRU werden tendenziell gerade die Seiten verdrängt, die als nächste benötigt werden.

Ein DBMS muss daher weitere Möglichkeiten haben, die auf der Kenntnis der Semantik und Realisierungskonzepte von DB-Operationen beruhen. Die Möglichkeiten sind:

- Fixieren von Seiten: Ein Seite wird von der Verdrängung ausgeschlossen.
- Freigeben von Seiten: Eine Seite kann zur Verdrängung freigegeben werden.
- Zurückschreiben einer Seite: Eine geänderte Seite kann auf die Platte zurückgeschrieben werden, um sie schneller verdrängbar zu machen.

6.2.2 Schattenspeicher

Das **Schattenspeicherkonzept** ***(shadow page table)*** ist eine Modifikation des Pufferkonzepts. Es ist ein Konzept, um im Fehlerfall den alten Datenbankzustand wiederherzustellen. Wird im Laufe einer Transaktion eine Seite auf die Platte zurückgeschrieben, so wird sie nicht auf die Original-Seite geschrieben, sondern auf eine neue Seite. Für die Pufferrahmen gibt es zwei Hilfsstrukturen V_0 und V_1. In der Version V_0 sind die Adressen der Original-Seiten auf dem Sekundärspeicher vermerkt, in dem Schattenspeicher V_1 werden die neuen Seiten geführt. Diese Hilfsstrukturen heißen auch **virtuelle Seitentabellen**. Transaktionen arbeiten nun auf den Seiten im Puffer oder den über die virtuellen Seitentabellen erreichbaren Seiten. Wird eine Transaktion abgebrochen, dann muss man nur auf die Ursprungsversion V_0 zurückgreifen und hat somit den Zustand vor Beginn der Transaktion wiederhergestellt. Wird die Transaktion erfolgreich abgeschlossen, dann wird V_1 die Originalversion, und V_0 wird als neuer Schattenspeicher verwendet.

6.3 Dateiverwaltung

6.3.1 Datenbank und Betriebssystem

Ein DBMS kann sich unterschiedlich stark auf das Betriebssystem abstützen.

- Das DBMS kann jede Relation oder jeden Zugriffspfad in einer Betriebssystem-Datei speichern. Damit ist die Struktur der DB auch im Dateisystem des Betriebssystems sichtbar.
- Das DBMS legt relativ wenige Betriebssystem-Dateien an und verwaltet die Relationen und Zugriffspfade innerhalb dieser Dateien selbst. Damit ist die Struktur der DB im Dateisystem des Betriebssystems nicht mehr sichtbar.
- Das DBMS steuert selbst das sekundäre Speichermedium als *raw device* an. Das DBMS realisiert das Dateisystem selbst am Betriebssystem vorbei.

  Damit wird das DBMS zwar unabhängig vom Betriebssystem, hängt aber von Gerätespezifika ab. Das DBMS muss sich nicht bezüglich der Dateigröße (4 GB bei 32-Bit-BS) einschränken. Dateien können Medien-übergreifend angelegt werden. Die Pufferverwaltung kann DB-spezifisch gestaltet werden.

6.3.2 Typen

Ein **Datensatz** ***(record)*** besteht aus **Feldern** ***(field)***. Jedes Feld hat einen **Datentyp** ***(data type)*** und einen **Wert** ***(value, item)***. Die Datentypen der Felder ergeben den **Datensatztyp** ***(record type, record format)***.

Zusätzlich zu den üblichen Datentypen `Integer`, `Long-Integer`, `Real`, `String`, `Boolean` gibt es bei DB-Anwendungen `Date`, `Time`, `BLOB` *(binary large object)* und `CLOB` *(character large object)*. Blobs sind große unstrukturierte Objekte, die Bilder, Video oder Audioströme repräsentieren, Clobs repräsentieren Texte.

6.3.3 Dateiaufbau

Eine Datei ist eine Folge von Datensätzen. Wenn jeder Datensatz dieselbe Länge hat, spricht man von ***fixed-length record***, sonst von ***variable-length record***.

Bei fester Länge kann die Adresse eines Datensatzes leicht berechnet werden. Allerdings wird oft Speicher verschwendet, da auch Datensätze mit kurzen Feldinhalten dieselbe Maximalanzahl von Bytes verbrauchen. Bei variabler Länge wird Speicher gespart, aber der Verwaltungsaufwand ist größer. Es gibt verschiedene Möglichkeiten:

- Füllung mit einem speziellen Zeichen auf eine feste Länge.
- Verwendung von speziellen Trennzeichen zwischen Feldern.
- Speicherung der Anzahl Felder und für jedes Feld ein (Feldlänge, Feldwert)-Paar.
- Speicherung der Anzahl Felder, dann für jedes Feld einen Zeiger auf den Wert des Feldes und schließlich für jedes Feld den Feldwert (dort, wo der Zeiger hinzeigt).
- (bei optionalen Feldern) Speicherung von (Feldname, Feldwert)-Paaren, wobei der Feldname geeignet zu kodieren ist.
- *(repeating field)* Ein Trennzeichen wird benötigt, um die Feldwiederholung zu trennen, ein weiteres Trennzeichen wird benötigt, um den Feldtyp abzuschließen.
- (bei unterschiedlichen Datensatztypen) Jeder Datensatz wird durch eine Typkennung eingeleitet.
- Bei **BLOB**s ***(binary large object)***, die beliebige Byte-Folgen (Bilder, Audio- und Videosequenzen) aufnehmen, und **CLOB**s ***(character large object)***, die ASCII-Zeichen aufnehmen, wird als Feldwert ein Zeiger auf den Beginn einer Blockliste verwaltet. Um wahlfrei in das BLOB hineingreifen zu können, eignet sich anstatt des Zeigers die Speicherung eines BLOB-Verzeichnisses, das mehrere Zeiger auf die verschiedenen Blöcke des BLOBs enthält.

6.3.4 Datensätze und Blöcke bzw Seiten

Definitionen:

- b_s ist die Blockgröße *(block size)* des Datenbanksystems in Byte.
- t_{s_R} ist die (mittlere) Größe von Tupeln der Relation R.

Die Datensätze einer Datei müssen Blöcken bzw Seiten auf der Platte zugeordnet werden **(Blocken,** ***record blocking)***. Wenn der Block größer als der Datensatz ist, dann enthält ein Block viele Datensätze. Das abgerundete Verhältnis

$$f_R = \frac{b_s}{t_{s_R}} = \frac{\text{Blockgröße}}{\text{Datensatzgröße}}$$

ist der **Blockungsfaktor** ***(blocking factor)*** einer Datei. Um den ungenutzten Platz zu verwenden, kann ein Teil eines Datensatzes in einem Block, der Rest in einem anderen Block gespeichert werden. Ein Zeiger am Ende des ersten Blocks enthält die Adresse des nächsten Blocks. Diese Organisation heißt **blockübergreifend (Spannsatz,** ***spanned)***. Die Blockverarbeitung wird allerdings einfacher, wenn die Datensätze gleiche Länge haben, die Datensätze kleiner als die Blöcke sind und eine nicht-blockübergreifende **(Nichtspannsatz,** ***unspanned)*** Organisation gewählt wird.

6.3.5 Blöcke einer Datei

Für die Allokierung der Blöcke einer Datei auf der Platte gibt es verschiedene Standardtechniken:

- ***contiguous allocation***: Die Blöcke einer Datei liegen hintereinander. Die Datei kann schnell gelesen und geschrieben werden. Die Erweiterung einer Datei ist schwierig.
- ***linked allocation***: Jeder Block hat am Ende einen Zeiger auf den nächsten Block, der zu dieser Datei gehört. Die Erweiterung einer Datei ist unproblematisch. Wenn auch noch der vorhergehende Block mitverwaltet wird, werden diese Zeiger und andere Informationen über den Block in einen **Kopf** ***(header)*** des Blocks geschrieben.
- ***clusters (segments, extents)***: Nach Möglichkeit werden mehrere hintereinanderliegende Blöcke *(cluster)* verwendet. Das Ende des Clusters zeigt auf den nächsten Cluster.
- **Indexierte Allokierung** ***(indexed allocation)***: Ein oder mehrere Index-Blöcke enthalten Zeiger auf die Datenblöcke der Datei.
- **Kombinationen** der oben genannten Techniken

Wir wollen hier einfach auf den Unterschied zwischen **sequentiellen** ***(sequential, serial, in disk order)*** I/O-Operationen und **nichtsequentiellen** ***(nonsequential)*** I/O-Operationen hinweisen. Sequentielle Operationen sind im allgemeinen wesentlich performanter.

6.3.6 Addressierung von Datensätzen

Datensätze können im Prinzip durch Angabe der Seiten-Nummer und eines Offsets auf der Seite adressiert werden. Dabei sind die Datensätze aber an die Position auf einer Seite fixiert ***(pinned)***. Muss der Datensatz nach einer Aktualisierung verschoben werden, dann müssten alle Referenzen auf diesen Datensatz nachgezogen werden, was viel zu aufwendig ist. Dieses Problem wird mit dem in relationalen DBMS üblichen **TID**-Konzept gelöst **(Tupel-Identifikator)** ***(record identifier, row-ID, RID)***.

Ein TID ist eine Datensatz-Adresse, die aus der Seitennummer und einem Offset besteht und die nicht direkt auf den Datensatz selbst, sondern nur auf einen Datensatz-Zeiger zeigt. Diese Datensatz-Zeiger befinden sich am Anfang einer Seite und enthalten den Offset zum eigentlichen Datensatz.

Wenn ein Datensatz innerhalb der Seite verschoben wird, dann ändert sich nur der Eintrag bei den Datensatz-Zeigern.

Wenn ein Datensatz auf eine andere Seite verschoben wird, dann wird an die alte Stelle des Datensatzes der neue TID (TID2) des Datensatzes geschrieben. Dabei entstehen zweistufige und auch mehrstufige Referenzen.

Daher wird eine solche Struktur in regelmäßigen Abständen reorganisiert.

Für die Diskussionen in den Kapiteln über Zugriffsstrukturen, Basisoperationen und Optimierung ist nur wichtig zu wissen, dass mit dem TID der Zugriff auf ein Tupel nur **ein** oder wenige Disk-I/Os kostet.

6.3.7 Dateikopf

Der **Dateikopf** ***(file header, file descriptor)*** enthält Informationen über einen File. Diese Information wird benötigt, um auf die einzelnen Datensätze zuzugreifen. Das bedeutet, der Dateikopf enthält Blockadressen und Dateisatzformate. Um einen Datensatz zu suchen, müssen Blöcke in den Hauptspeicher kopiert werden. Wenn die Blockadresse nicht bekannt ist, müssen alle Blöcke der Datei durchsucht werden ***(linear search)***. Das kann viel Zeit kosten. Ziel einer guten Dateiorganisation ist die Lokalisierung des Blocks, der den gesuchten Datensatz enthält, mit einer minimalen Anzahl von Blockübertragungen.

6.3.8 Dateioperationen

Unabhängig vom jeweiligen DBMS und Betriebssystem wollen wir hier die wichtigsten Dateioperationen nennen.

Suchen *(find, locate)*: Sucht nach dem ersten Datensatz, der einer gewissen Suchbedingung genügt. Transferiert den Block in einen Hauptspeicherpuffer, wenn der Block nicht schon dort ist. Der gefundene Datensatz wird der **aktuelle Datensatz *(current record)***.

Lesen (holen, *read, get)*: Kopiert den aktuellen Datensatz aus dem Hauptspeicherpuffer in eine Programmvariable oder den Arbeitsspeicher eines Programms. Gleichzeitig wird häufig der nächste Datensatz zum aktuellen Datensatz gemacht.

Weitersuchen *(find next)*: Sucht nach dem nächsten Datensatz, der einer Suchbedingung genügt und verfährt wie bei Suchen.

Löschen *(delete)*: Löscht den aktuellen Datensatz.

Aktualisieren *(modify)*: Verändert einige Werte im aktuellen Datensatz.

Einfügen *(insert)*: Fügt einen neuen Datensatz in die Datei ein, wobei zunächst der Block lokalisiert wird, in den der Datensatz eingefügt wird. Unter Umständen muss ein neuer Block angelegt werden.

Neben diesen ***record-at-a-time*** Operationen gibt es eventuell auch ***set-at-a-time*** Operationen, die mehrere Datensätze betreffen:

Suche alle *(find all)*: Lokalisiert alle Datensätze, die einer Suchbedingung genügen.

Reorganisiere *(reorganize)*: Startet den Reorganisationsprozess z.B. zur Sortierung einer Datei nach einem Datensatzfeld.

6.4 Primäre Dateiorganisation

6.4.1 Datei mit unsortierten Datensätzen

Die einfachste Dateiorganisation ist die **Heap-Datei** ***(heap file, pile file,*** (manchmal auch) ***sequential file)***, wo jeder neue Datensatz am Ende angefügt wird.

Dieses Anfügen geht sehr effizient: Der letzte Block der Datei wird in den Hauptspeicherpuffer kopiert, der neue Datensatz dort angefügt, und der Block wird auf die Platte zurückgeschrieben. Die Adresse des letzten Blocks wird im Dateikopf verwaltet. Aber die Suche nach einem Datensatz mit einer Suchbedingung erfordert einen linearen Durchgang durch – im Schnitt die Hälfte – aller Blöcke.

Zur Löschung eines Datensatzes muss der entsprechende Block gefunden und in den Hauptspeicherpuffer kopiert werden. Dort bekommt der Datensatz eine **Löschmarke** ***(deletion marker)*** oder wird mit einer Defaultinformation überschrieben. Beide Techniken lassen Speicherplatz ungenutzt. Der Block wird wieder auf die Platte zurückgeschrieben. Eine gelegentliche Reorganisation der Datei stellt den verlorenen Speicherplatz wieder zur Verfügung; oder neue Datensätze werden an Stelle der gelöschten Datensätze eingefügt.

Bei einer Datei mit Datensätzen fester Länge, nicht übergreifend belegten, hintereinanderliegenden Blöcken kann ein Datensatz durch seine Position in der Datei direkt angesprochen werden ***(relative file)***. Das hilft zwar nicht beim Suchen, aber erleichtert die Erstellung von Zugriffspfaden.

6.4.2 Datei mit sortierten Datensätzen

Wenn die Datensätze einer Datei physisch nach einem **Sortierfeld** ***(ordering field)*** sortiert sind, dann hat man eine **sortierte Datei** ***(ordered, sequential file)***. Das Sortierfeld kann ein **Schlüsselfeld** ***(key field, ordering key)*** sein, d.h. den Datensatz eindeutig bestimmen **(sequentielle Datei)**.

Bei einer sortierten Datei ist eine **Binärsuche** ***(binary search)*** möglich, wenn die Suchbedingung auf dem Sortierfeld beruht. Wenn die Datei b Blöcke hat, dann werden im Schnitt

$$\log_2(b)$$

Blockzugriffe benötigt, um einen Datensatz zu finden.

Wenn die Suche nicht auf dem Sortierfeld beruht, dann bietet eine sortierte Datei keinen Vorteil, da in diesen Fällen auch linear gesucht werden muss. Ferner ist eine binäre Suche oft praktisch nicht möglich, da man ja nicht weiß, wo denn jeweils der "mittlere" Block liegt. Daher macht diese Organisation meistens nur Sinn, wenn zusätzlich ein Index verwendet wird.

Einfügen und Löschen von Datensätzen sind "teure" Operationen, da im Schnitt die Hälfte aller Blöcke bewegt werden muss. Beim Löschen kann man aber mit Löschmarken arbeiten. Zur Milderung des Einfügeproblems kann man in jedem Block Platz für neue Datensätze reservieren. Eine weitere Technik ist die Verwendung einer zusätzlichen **Transaktionsdatei** ***(transaction, overflow file)***, wobei neue Datensätze dort einfach angehängt werden. Periodisch wird die ganze Transaktionsdatei in die geordnete **Hauptdatei** ***(main, master file)*** eingefügt. Einfügen wird

dadurch sehr effizient, aber der Suchalgorithmus ist komplizierter, da nach einer Binärsuche in der Hauptdatei noch eine lineare Suche in der Transaktionsdatei durchgeführt werden muss.

6.4.3 Hash-Dateien

Hash-Dateien ***(hash, direct file)*** bieten sehr schnellen Zugriff auf Datensätze, wenn die Suchbedingung ein Gleichheitsvergleich mit einem **Hash-Feld** ***(hash field)*** ist. Wenn das Hash-Feld auch ein Schlüssel ist, dann spricht man von einem **Hash-Schlüssel** ***(hash key)***.

Mit Hilfe der **Hash-Funktion** ***(hash function, randomizing function)*** kann aus dem Hash-Feld-Wert die Adresse des Blocks ermittelt werden, in dem der Datensatz gespeichert ist.

Das Speichermedium wird in **Eimer** ***(bucket)*** aufgeteilt. Ein Eimer ist entweder ein Block oder ein Cluster hintereinanderliegender Blöcke. Die Hash-Funktion bildet den Schlüssel auf die relative Eimernummer ab. Eine Tabelle im Dateikopf stellt die Beziehung zwischen Eimernummer und absoluter Blockadresse her.

Normalerweise passen viele Datensätze in einen Eimer, sodass relativ wenige Kollisionen auftreten. Wenn allerdings ein neuer Datensatz in einen vollen Eimer hineinhashed, dann kann eine Variation des Verkettungsmechanismus angewendet werden, bei dem jeder Eimer einen Zeiger auf eine Liste überzähliger Datensätze verwaltet. Die Zeiger in der Liste sind Zeiger auf Datensätze, d.h. enthalten eine Blockadresse mit relativer Datensatzposition in diesem Block.

Für die Hash-Funktion h gibt es verschiedenste Möglichkeiten. Auf irgendeine Art muss der Wert k des Hash-Feldes in einen Integer c verwandelt werden (Hashcode h_c). Die Anzahl der Eimer sei M mit den relativen Eimernummern 0 bis $M-1$. Beispiele für Hash-Funktionen sind dann:

- $c = h_c(k)$ (Hashcode)
- $h(k) = c \bmod M$ (Hashfunktion)
- $h(k) =$ irgendwelche n Stellen von c, wobei $M = 10^n$
  Z.B. sei $n = 3$. Wenn $c = 3627485$, dann werden z.B. die mittleren n Stellen von c, also 274 als $h(k)$ genommen.

Die Suche mit einer allgemeinen Suchbedingung bedeutet lineare Suche durch alle Blöcke der Datei. Modifikationen von Datensätzen sind aufwendig, wenn das Hash-Feld beteiligt ist.

Für die Datei muss ein fester Speicherplatz allokiert werden. Daher betrachten wir jetzt noch Hash-Techniken, die eine dynamische Dateierweiterung erlauben. Diese Techniken beruhen darauf, dass das Resultat der Hash-Funktion ein nicht negativer Integer ist, den wir als Binärzahl ***(hash value)*** darstellen können.

Dynamisches Hashing

Beim **dynamischen Hashing** ***(dynamic hashing)*** ist die Anzahl der Eimer nicht fest, sondern wächst oder schrumpft nach Bedarf. Die Datei beginnt mit einem Eimer (evtl.mit Blattknoten). Wenn der Eimer voll ist, wird ein neuer Eimer angelegt, und die Datensätze werden – abhängig vom ersten (linken) Bit des Hashcodes – auf die beiden Eimer verteilt. Damit wird eine binäre Baumstruktur, der **Index (Verzeichnis**, ***directory, index)***, aufgebaut.

Der Index hat zwei Knotentypen:

- **Interne Knoten** ***(internal node)*** enthalten einen Links(0Bit)- und einen Rechts(1Bit)-Zeiger auf einen internen Knoten oder einen Blattknoten.
- **Blattknoten** ***(leaf node)*** enthalten einen Zeiger auf einen Eimer.

Wenn Eimer leer werden oder wenn benachbarte Eimer so wenige Datensätze enthalten, dass sie in einen Eimer passen, dann wird der Baum zurückgebaut.

Erweiterbares Hashing

Beim **erweiterbaren Hashing** ***(extendible hashing)*** wird der Index als ein Feld von 2^d Eimeradressen verwaltet, wobei d die **globale Tiefe** ***(global depth)*** des Index ist. Der Index, der den ersten d Bit des Hashwertes entspricht, wird als Feldindex zur Bestimmung der Eimeradresse verwendet. Der Trick ist, dass die Eimeradressen nicht alle verschieden sein müssen, d.h. die lokale Tiefe der Eimer kann $d_l < d$ sein. Der Index muss verdoppelt werden, wenn ein Eimer mit lokaler Tiefe $d_l = d$ voll ist und ein neuer Datensatz eingefügt werden soll. Der Index kann halbiert werden, wenn alle Eimer eine lokale Tiefe $< d$ haben. d muss $\leq b$ (Anzahl der Bit im Hashwert) sein.

Lineares Hashing

Das **lineare Hashing** ***(linear hashing)*** erlaubt eine Datei zu vergrößern oder zu verkleinern ohne Verwendung eines Indexes. Angenommen die Datei hat zunächst M Eimer 0 bis $M-1$ und verwendet die Hashfunktion $h_0(k) = h_c(k) \bmod M$. Overflow wegen Kollisionen wird mit (für jeden Eimer) individuellen Overflowlisten behandelt. Wenn es aber zu *irgendeiner* Kollision kommt, dann wird der erste Eimer 0 aufgeteilt in zwei Eimer: Eimer 0 und Eimer M. Die Datensätze des alten Eimers 0 werden auf die beiden Eimer verteilt mit der Hashfunktion $h_1(k) = h_c(k) \bmod 2M$. Eine wichtige Eigenschaft der Hashfunktionen h_0 und h_1 muss sein, dass, wenn h_0 nach dem alten Eimer 0 hashed, h_1 nach den neuen Eimern 0 oder M hashed. Bei weiteren Kollisionen werden in linearer Folge die Eimer $1, 2, 3 \ldots$ verdoppelt (gesplittet). Es muss also die Nummer N *(split pointer)* des als nächsten zu splittenden Eimers verwaltet werden. Wenn die Datei zu leer wird, können Eimer wieder koaleszieren. Wann gesplittet wird, hängt irgendwie vom Füllungsgrad ab. Der als nächster gesplittete Eimer hat somit *keinen* Bezug zu dem Eimer, in den ein Datensatz einzufügen ist. Das ist ein Nachteil des linearen Hashings: Es werden Eimer gesplittet, die gar nicht voll sind.

Diesen Nachteil vermeidet das sogenannte Spiral-Hashing, wobei die Hash-Funktion derart manipuliert wird, dass der Füllgrad der Seiten an der Position des Split-Zeigers am größten ist ([34] Seite 171).

6.5 Übungen

6.5.1 Magnetplatte

Betrachten Sie eine Magnetplatte mit folgenden Eigenschaften: Blockgröße $B = 512$ Byte, interblock gap $G = 128$ Byte, Anzahl Blöcke pro Spur $N = 150$, Anzahl der Spuren pro Oberfläche $Z = 5000$, Anzahl der Plattenoberflächen $P = 20$ (Plattenstapel mit 10 doppelseitigen Platten), Rotationsgeschwindigkeit $R = 10000$ UPM, mittlere Zugriffsbewegungszeit $T_z = 8$ ms.

1. Was ist die Gesamtkapazität einer Spur?
2. Was ist die nutzbare Kapazität einer Spur?
3. Was ist die Anzahl der Zylinder?
4. Was ist die nutzbare Kapazität eines Zylinders?
5. Was ist die nutzbare Kapazität dieser Platte?
6. Was ist die Transferrate in KB/ms?
7. Was ist die Block-Transferzeit in μs?
8. Was ist die mittlere Umdrehungswartezeit in ms?
9. Wielange dauert im Mittel das Finden und Übertragen von 20 willkürlich verstreuten Blöcken.
10. Wielange dauert im Mittel das Finden und Übertragen von 20 hintereinander liegenden Blöcken.

6.5.2 StudentInnenverwaltung 1

Eine Datei auf oben angegebener Platte hat $r = 20000$ STUDentInnen-Datensätze fester Länge. Jeder Datensatz hat die folgenden Felder:

NAME (30 Byte, Name),
MATR (6 Byte, Matrikel),
ADRE (40 Byte, Adresse),
TELE (16 Byte, Telefon),
GEBD (8 Byte, Geburtsdatum),
GESC (1 Byte, Geschlecht),
BERE (3 Byte, Bereichscode),
FACH (3 Byte, Fachrichtungscode),
GRUP (4 Byte, Gruppe),
WAHL (6 Byte, Wahlfächerkombination)

Ferner wird ein Byte als Löschmarke verwendet.

1. Berechnen Sie die Datensatzgröße R.

2. Berechnen Sie den Blockungsfaktor bei einer Blockgröße von 512 Byte bei nicht blockübergreifender Organisation.

3. Berechnen Sie die Anzahl der Blöcke, die diese Datei benötigt.

4. Berechnen Sie die durchschnittliche Zeit, die für das Aufinden eines Datensatzes benötigt wird, wenn die Datei linear durchlaufen wird. Dabei sind noch zwei Fälle zu unterscheiden:

   (a) Contiguous Allocation der Blöcke

   (b) Linked Allocation der Blöcke

5. Angenommen die Datei sei nach Matrikelnummer geordnet. Wielange dauert eine Binärsuche nach einer bestimmten Matrikelnummer?

6. Angenommen die Datei sei eine Hash-Datei, wobei die Hashfunktion MATR mod 6007 verwendet wird. Jeder Eimer ist ein Block. Die Hashfunktion ergibt die Blockadresse (der Einfachheit halber). Bei Kollision wird der nächste Block genommen.

   (a) Wielange dauert die Suche nach einer bestimmten Matrikelnummer?

   (b) Kann die Anzahl der StudentInnen ohne Umorganisation der Datei wachsen? Wenn ja, wie weit?

6.5.3 StudentInnenverwaltung 2

Angenommen nur 30% der STUDentInnen-Datensätze (siehe StudentInnenverwaltung 1) haben einen Wert für das Telefon, nur 60 % einen Wert für die Adresse und nur 20% einen Wert für die Wahlfächerkombination. Daher wollen wir jetzt eine Datei mit variabler Datensatzlänge verwenden. Jeder Datensatz hat aber zusätzlich ein Feldtyp-Byte für jedes mit Werten belegte Feld und ein Byte als Datensatz-Ende-Marke. Ferner benutzen wir eine Spannsatz-Organisation, wobei jeder Block einen 5-Byte-Zeiger auf den nächsten Block hat.

1. Wie groß ist die maximale Länge eines Datensatzes?

2. Wie groß ist die durchschnittliche Länge eines Datensatzes?

3. Wie groß ist die Anzahl der benötigten Blöcke?

6.5.4 Teile-Verwaltung

Angenommen eine Datei hat Teile-Datensätze, die als Schlüssel die Teile-Nummern $T = 2369$, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208 haben. Die Datei benutzt 8 Eimer mit den Nummern 0 bis 7. Jeder Eimer ist ein Plattenblock und kann höchstens zwei Datensätze aufnehmen.

1. Laden Sie diese Datensätze in der angegebenen Ordnung in die Datei unter Verwendung der Hash-Funktion $h(T) = T \bmod 8$. Berechnen Sie die durchschnittliche Anzahl der Blockzugriffe beim Zugriff auf einen beliebigen Datensatz.

2. Benutzen Sie dynamisches Hashing mit der Hash-Funktion $h(T) = \text{mod}32$ und laden Sie die Teile nacheinander in die Datei. Zeigen Sie die Struktur der Datei nach jeder Einfügung. Jeder Eimer kann zwei Datensätze aufnehmen.

3. Benutzen Sie erweiterbares Hashing mit der Hash-Funktion $h(T) = \text{mod}32$ und laden Sie die Teile nacheinander in die Datei. Zeigen Sie die Struktur der Datei nach jeder Einfügung. Lassen Sie die Eimer-Adressen willkürlich bei EA beginnen. Die nächsten Eimer haben dann die Adressen EB, EC usw. Jeder Eimer kann zwei Datensätze aufnehmen.

4. Benutzen Sie lineares Hashing. Jeder Eimer kann zwei Datensätze aufnehmen.

Kapitel 7

Zugriffsstrukturen

In diesem Kapitel werden **Zugriffspfade** oder **Zugriffsstrukturen** ***(access path, access structure)***, sogenannte **Indexe** ***(index)*** besprochen, die die Suche von Dateisätzen bei gewissen Suchbedingungen beschleunigen.

Mit dem Index eines Buches können wir Informationen in dem Buch finden ohne das ganze Buch zu lesen. Die Indexe einer DB erlauben es, Tupel zu finden ohne die ganze Tabelle zu lesen. Ein Index ist im Prinzip eine Liste von Datenwerten mit Adressen derjenigen Tupel, die diesen Datenwert enthalten.

Der Index einer Datei ist immer mindestens eine zusätzliche Datei. Ziel der Einführung von Indexen ist die Reduktion der ("teuren") Blockzugriffe bei der Suche nach einem Datensatz.

Bei den Suchbedingungen unterscheiden wir:

- ***single-match query:*** Für ein Attribut wird ein Wert vorgegeben **(exakte Suche)**.
- ***exact-match query:*** Für jedes Attribut wird ein Wert vorgegeben **(exakte Suche)**.
- ***partial-match query:*** Für einige Attribute wird ein Wert vorgegeben **(exakte Suche)**.
- ***range query:*** Für einige Attribute wird ein Wertebereich vorgegeben **(Bereichsanfrage)**.

Der **Suchschlüssel** oder **Suchfeldwert** oder **Indexfeld** ***(search key, search field value, indexing field)*** ist das, wonach gesucht wird (nicht zu verwechseln mit dem Schlüssel einer Tabelle). Suchschlüssel können *einfach* (ein Attribut) oder (aus mehreren Attributen) *zusammengesetzt* sein. Entsprechend spricht man manchmal auch von **Ein-Attribut-Indexen** ***(non-composite index)*** oder **Mehr-Attribut-Indexen** ***(composite index)***.

Ein Ein-Attribut-Index wird immer als **eindimensionale Zugriffsstruktur** realisiert. Ein Mehr-Attribut-Index kann als **eindimensionale** oder **mehrdimensionale Zugriffsstruktur** ausgeführt werden.

Im eindimensionalen Fall werden die Werte der Attribute eines Mehr-Attribut-Indexes zu einem Wert konkateniert, der wie der Wert *eines* Attributs behandelt wird. Eine exact-match-Anfrage

bezüglich des Mehr-Attribut-Indexes wird daher durch eine eindimensionale Zugriffsstruktur effektiv unterstützt. Für eine partial-match-Anfrage werden mehrdimensionale Zugriffsstrukturen benötigt.

Wenn aber statt eines Mehr-Attribut-Indexes ein eindimensionaler Indexe verwendet wird, kann z.B. bei einem Index auf den Spalten `A, B, C` effektiv nach `A` oder nach `A+B` oder nach `A+B+C` gesucht werden, wobei das `+` eine Konkatenierung bedeutet.

Ansonsten wird eben für jedes Attribut eines Mehr-Attribut-Indexes je ein Index angelegt. Die jeweils resultierenden TID-Listen werden dann geschnitten.

Bemerkung zur Terminologie: Es werden **geclusterte *(clustered)* (index-organisierte Tabelle)** und **ungeclusterte *(non-clustered)*** Indexe unterschieden. Im ersten Fall sind die Tupel der indizierten Relation im Index untergebracht. Geclusterte Indexe bieten Vorteile bei Bereichs-Anfragen. Die im Kapitel "Speicherstrukturen" behandelten sortierten Dateien oder Hash-Dateien haben demnach einen geclusterten Index auf dem Sortierfeld. (Diese "geclusterten" Indexe unterscheiden sich von den im nächsten Abschnitt behandelten "Clusterindexen" nur dadurch, dass das Indexfeld auch ein Schlüssel der Tabelle sein kann, **und,** dass die Tabelle als zum Index gehörig betrachtet wird und keine eigene Existenz hat.)

7.1 Einstufige sortierte Indexe

Der **einstufig sortierte Index *(single-level ordered index)*** orientiert sich an der Technik des Schlagwortverzeichnisses bei Büchern. Der Index wird verwendet, um Blockadressen nachzuschlagen. Der Index wird nach den Werten sortiert, sodass eine Binärsuche durchgeführt werden kann.

Man unterscheidet folgende Indexe:

Primärindex *(primary index)*: Das **Indexfeld *(indexing field)*** ist **Schlüssel** *und* **Sortierfeld** der Hauptdatei. Es kann nur *einen* Primärindex geben.

Bei einem Primärindex genügt es, für jeden Block der Hauptdatei das Indexfeld des ersten (oder letzten) Datensatzes im Block **(Ankerdatensatz, Seitenanführer, *anchor record, block record)*** und die Blockadresse zu speichern. Solch ein Primärindex ist **nicht dicht (dünnbesetzt, *non-dense)***. (Ein **dichter (dichtbesetzt, *dense)*** Index speichert für *jeden* Datensatz Indexfeld und Blockadresse.)

Die Kombination von Hauptdatei und Primärindexdatei wird oft als **indexsequentielle Dateiorganisation** bezeichnet, da normalerweise solch ein Index linear durchsucht werden muss.

Clusterindex *(clustering index)*: Das Indexfeld ist *nur* **Sortierfeld** der Datei, aber kein Schlüssel. Es kann nur *einen* Clusterindex geben.

Bei einem Clusterindex wird für jeden Wert des Indexfeldes die Adresse des ersten Blocks gespeichert, der einen Datensatz mit dem Wert des Indexfeldes enthält. Ein Clusterindex ist ebenfalls *nicht dicht*. Er unterstützt Bereichsanfragen sehr gut, da die Sortierung beibehalten wird.

Sekundärindex *(secondary index)* (sekundäre Zugriffspfade): Indexfeld kann jedes Feld eines Datensatzes sein. Mehrere Sekundärindexe sind möglich, da diese Indexe unabhängig

von der physikalischen Anordnung der Datensätze auf der Platte sind. Sie bieten alternative Suchpfade auf der Basis von Indexfeldern.

Eine Datei, die auf jedem Feld einen Sekundärindex hat, heißt **vollständig invertiert *(fully inverted file)***. (Einen Sekundärindex nennt man auch **invertierte Datei**.)

Bei einem Sekundärindex auf einem *Schlüssel*indexfeld wird für *jeden* Wert des Indexfeldes die Blockadresse oder Datensatzadresse gespeichert. Der Sekundärindex ist daher ein dichter Index und benötigt mehr Speicher und Suchzeit als ein Primärindex.

Wenn das Indexfeld kein Schlüssel ist, dann gibt es pro Wert mehrere Blockadressen. Das kann mit folgenden Möglichkeiten (Optionen) verwaltet werden:

1. Für jeden Datensatz wird ein Indexeintrag gemacht, wobei manche Indexfeldwerte öfter vorkommen.
2. Zu jedem Indexfeldwert wird eine Liste von Blockadressen geführt (etwa in einem Datensatz variabler Länge). Diese Blockadressen zeigen auf Blöcke, wo mindestens ein Datensatz den Indexfeldwert enthält.
3. (Das ist die verbreitetste Methode.) Für jeden Indexfeldwert gibt es genau einen Eintrag mit einer Blockadresse, die auf eine Block zeigt, der Zeiger auf die Datensätze enthält, die diesen Indexfeldwert enthalten. Wird mehr als ein Block benötigt, wird eine verkettete Liste von Blöcken verwendet.

Beim Primär- und Clusterindex muss die Datensatz-Datei sortiert sein. Dies erfordert einen gewissen Aufwand (siehe Kapitel Speicherstrukturen). Daher verwenden manche DBS nur Sekundärindexe.

7.2 Mehrstufige Indexe

Die bisher beschriebenen Indexschemata benutzen *eine* sortierte Indexdatei, in der binär gesucht wird, d.h. der Suchraum wird bei jedem Schritt nur um den Faktor 2 reduziert.

Bei **mehrstufigen *(multilevel)*** Indexen wird der Suchraum bei jedem Schritt um den **Blockungsfaktor** (Anzahl der Datensätze in einem Block) des Indexes reduziert(f_I). In dem Zusammenhang heißt der Blockungsfaktor auch ***fan-out*** (f_o). Wir benötigen daher etwa $\log_{f_o}(b)$ Blockzugriffe, wobei b die Anzahl der Datensätze der ersten Stufe des Indexes ist. Das kann die Anzahl der Blöcke oder die Anzahl der Datensätze der Hauptdatei oder die Anzahl der unterschiedlichen Werte des Indexfeldes sein.

Die erste Stufe ***(first, base level)*** eines mehrstufigen Index ist eine sortierte Datei mit eindeutigen Indexfeldwerten (Schlüsselindexfeld). Daher können wir für die erste Stufe einen Primärindex anlegen. Dieser bildet die zweite Stufe ***(second level)***. Da die zweite Stufe ein Primärindex ist, genügt es, nur auf die Ankerdatensätze zu verweisen. Die dritte Stufe wird dann als Primärindex für die zweite Stufe angelegt. Dieser Prozess wird fortgesetzt, bis die letzte Stufe ***(top index level)*** in einen Block passt.

Um das Einfügen und Löschen von Datensätzen zu erleichtern wird häufig in jedem Block etwas Platz reserviert.

Diese Methode kann auch für Nicht-Schlüssel-Indexfelder adaptiert werden. (Bezieht sich auf Option 1 des vorangehenden Abschnitts.)

Ein Nachteil der bisher besprochenen Methoden ist die Verwaltung von sortierten Index- und eventuell auch Hauptdateien. Dieses Problem wird im folgenden mit Suchbäumen gelöst.

7.3 Dynamische mehrstufige Indexe mit Bäumen

Bezeichnungen: Ein **Baum** ***(tree)*** besteht aus **Knoten** ***(node)***. Jeder Knoten außer der **Wurzel** ***(root)*** hat einen **Elterknoten** ***(parent node)*** und mehrere (oder keinen) **Kindknoten** ***(child node)***. Ein Knoten ohne Kinder heißt **Blatt** ***(leaf)***. Ein nicht-Blatt-Knoten heißt **innerer** ***(internal)*** Knoten. Die **Stufe** ***(level)*** eines Knotens ist immer um Eins erhöht gegenüber dem Elterknoten. Die Wurzel hat Stufe 0. Ein **Unterbaum** ***(subtree)*** eines Knotens besteht aus diesem Knoten und allen seinen Nachkommen.

Ein Baum wird implementiert, indem ein Knoten Zeiger auf alle seine Kinder hat. Manchmal hat er auch noch einen Zeiger auf sein Elter. Zusätzlich zu den Zeigern enthält ein Knoten irgendeine Art von gespeicherter Information. Wenn ein mehrstufiger Index als Baumstruktur implementiert wird, dann besteht diese Information aus Werten eines Indexfeldes zur Führung der Suche nach einem bestimmten Datensatz.

7.3.1 Suchbäume

Bei einem **Suchbaum** ***(search tree)*** der Ordnung (Grad) p enthält jeder Knoten höchstens $p-1$ Suchwerte (Suchschlüsselwerte oder auch Zugriffsattributwerte) K_i und höchstens p Zeiger P_i auf Kindknoten in der Ordnung

$$\langle P_0, K_0, P_1, K_1 \ldots P_{q-1}, K_{q-1}, P_q \rangle,$$

wobei $q < p$.

Die K_i sind Werte aus einer geordneten Menge von eindeutigen Suchwerten **(Suchfeld** ***(search field)*** eines Datensatzes).

Es gilt

$$K_0 < K_1 < \ldots < K_{q-1}$$

und für alle Werte X in einem Unterbaum, auf den der Zeiger P_i zeigt, gilt

$K_{i-1} < X < K_i$ für $0 < i < q$
$X < K_i$ für $i = 0$
$K_{i-1} < X$ für $i = q$

Die Zeiger P_i zeigen auf Kindknoten oder ins Leere. Mit jedem K-Wert ist eine Blockadresse assoziiert.

Ein Problem bei solchen Suchbäumen ist, dass sie meist nicht **ausgeglichen** ***(balanced)*** sind. ("Ausgeglichen" heißt, dass alle Blätter möglichst auf demselben Niveau sind.)

7.3.2 *B*-Bäume

B-Bäume (*B* wie *balanced*) sind immer ausgeglichen. Dies wird durch zusätzliche Bedingungen garantiert, die auch dafür sorgen, dass der durch Löschungen vergeudete Speicherplatz nicht zu groß wird. Die entsprechenden Algorithmen sind normalerweise einfach, werden aber kompliziert, wenn in einen vollen Knoten eingefügt werden muss, oder wenn aus einem halbvollen Knoten gelöscht werden soll.

Zunächst definieren wir einen *B*-Baum der Ordnung p zur Verwendung als Zugriffsstruktur auf ein Schlüsselfeld um Datensätze in einer Datei zu suchen.

1. Jeder innere Knoten hat die Form

   $$\langle P_0, \langle K_0, Pr_0\rangle, P_1, \langle K_1, Pr_1\rangle \ldots P_{q-1}, \langle K_{q-1}, Pr_{q-1}\rangle, P_q\rangle,$$

   wobei $q < p \geq 3$. Jeder Zeiger P_i zeigt auf einen Kindknoten. Jeder Zeiger Pr_i ist ein Datenzeiger.

2. Jedes Pr_i ist ein Datenzeiger, der auf den Datensatz zeigt, dessen Suchfeldwert K_i ist, oder der auf einen Block zeigt, der den Datensatz mit dem Suchfeldwert K_i enthält, oder der – falls das Suchfeld kein Schlüssel ist – auf einen Block von Datensatzzeigern zeigt, die auf Datensätze zeigen, die den Suchfeldwert K_i haben.

   Im Fall eines **geclusterten** Indexes wird Pr_i direkt durch den Datensatz oder die Datensätze, die den Suchfeldwert enthalten, ersetzt.

3. In jedem Knoten gilt $K_0 < K_1 < \ldots < K_{q-1}$.

4. Für alle Suchschlüsselfeldwerte X in dem Unterbaum, auf den P_i zeigt, gilt

   $K_{i-1} < X < K_i$ für $0 < i < q$
   $X < K_i$ für $i = 0$
   $K_{i-1} < X$ für $i = q$

5. Jeder Knoten hat höchstens p Zeiger auf Kindknoten ($q < p$).

6. Jeder Knoten – ausgenommen der Wurzel und der Blätter – hat mindestens $\frac{p}{2}$ Zeiger auf Kindknoten. Die Wurzel hat mindestens 2 Zeiger auf Kindknoten, wenn sie nicht der einzige Knoten im Baum ist.

7. Ein Knoten mit $q+1$ Zeigern auf Kindknoten hat q Schlüsselfeldwerte und q Datenzeiger.

8. Alle Blattknoten sind auf demselben Niveau. Blattknoten haben dieselbe Struktur wie innere Knoten, außer dass die Zeiger P_i auf Kindknoten alle NULL sind.

Ein neues Element wird grundsätzlich immer in ein Blatt eingefügt. Wenn ein Blatt voll ist, dann wird es auf demselben Niveau gespalten, wobei das mittlere Element eine Stufe höher steigt.

Ein *B*-Baum beginnt mit der Wurzel, die zunächst auch ein Blatt ist. Ist die Wurzel voll, dann werden zwei Kindknoten gebildet. Der mittlere Wert bleibt in der Wurzel, alle anderen Werte werden auf die beiden Kinder verteilt.

Wenn ein Knoten voll wird, dann wird er aufgespalten in zwei Knoten auf *demselben* Niveau, wobei der mittlere Wert zusammen mit zwei Zeigern auf die neuen Knoten in den Elterknoten aufsteigt.

Wenn der Elterknoten voll ist, wird auch er auf demselben Niveau gespalten. Das kann sich bis zur Wurzel fortsetzen, bei deren Spaltung eine neue Stufe entsteht.

Wenn durch Löschung ein Knoten weniger als halbvoll wird, wird versucht, ihn mit Nachbarknoten zu verschmelzen. Auch dies kann sich bis zur Wurzel fortsetzen, sodass es zu einer Verminderung der Stufen kommen kann.

Durch Simulation von beliebigen Einfügungen und Löschungen konnte gezeigt werden, dass die Knoten etwa zu 69% gefüllt sind. Dann sind Knotenspaltungen und Knotenrekombinationen ziemlich selten, sodass Einfügungen und Löschungen normalerweise sehr schnell gehen.

B-Bäume werden manchmal für die primäre Dateistruktur verwendet (index-organisierte Tabelle oder geclusterter Index). In dem Fall werden ganze Datensätze in den Knoten gespeichert. Das geht bei wenigen Datensätzen und/oder kleinen Datensätzen gut. Ansonsten wird die Anzahl der Stufen oder die Knotengröße für einen effizienten Zugriff zu groß.

7.3.3 B^+-Bäume

In einem B-Baum erscheint jeder Wert des Suchfeldes genau einmal zusammen mit seinem Datenzeiger. In einem B^+-Baum werden Datenzeiger nur in den Blättern gespeichert. Die innneren Knoten enthalten keine Datenzeiger *(hohler Baum)*. Das bedeutet, dass die Blätter für jeden Suchwert einen Eintrag zusammen mit dem Datenzeiger haben müssen.

Die Blätter sind gewöhnlich durch Zeiger verknüpft, sodass ein schneller Zugriff in der Reihenfolge des Suchfeldes möglich ist. Die Blätter sind ähnlich strukturiert wie die erste Stufe eines Indexes. Die inneren Knoten entsprechen den anderen Stufen eines mehrstufigen Indexes. Einige Suchfeldwerte kommen in den inneren Knoten wieder vor.

Die Struktur des B^+-Baums ist folgendermaßen:

1. Jeder innere Knoten hat die Form

   $$\langle P_0, K_0, P_1, K_1 \ldots P_{q-1}, K_{q-1}, P_q \rangle$$

   mit $q < p_{Knoten}$. Die P_i sind dabei insgesamt $q+1$ Zeiger auf Kindknoten.

2. In jedem Knoten gilt: $K_0 < K_1 < \ldots < K_{q-1}$

3. Für alle Suchfeldwerte X in dem Unterbaum, auf den P_i zeigt, gilt:

   $K_{i-1} < X \leq K_i$ für $0 < i < q$
   $X \leq K_i$ für $i = 0$
   $K_{i-1} < X$ für $i = q$

4. Jeder innere Knoten hat höchstens p_{Knoten} Zeiger auf Kindknoten ($q < p_{Knoten}$).

5. Jeder innere Knoten außer der Wurzel hat mindestens $\frac{p_{Knoten}}{2}$ Zeiger. Die Wurzel hat mindestens 2 Zeiger, wenn sie kein Blatt ist.

6. Ein innerer Knoten mit $q+1$ Zeigern ($q < p_{Knoten}$) hat q Suchfeldwerte.

7. Jedes Blatt hat die Form

   $$\langle \langle K_0, Pr_0 \rangle, \langle K_1, Pr_1 \rangle \ldots \langle K_q, Pr_q \rangle, P_{next} \rangle,$$

wobei $q < p_{Blatt}$, Pr_i Datenzeiger sind und P_{next} auf das nächste Blatt zeigt.

8. Für jedes Blatt gilt: $K_0 < K_1 < \ldots < K_q$
9. Jedes Pr_i ist ein Datenzeiger, der auf den Datensatz zeigt, dessen Suchfeldwert K_i ist, oder der auf einen Block zeigt, der den Datensatz mit dem Suchfeldwert K_i enthält, oder der – falls das Suchfeld kein Schlüssel ist – auf einen Block von Datensatzzeigern zeigt, die auf Datensätze zeigen, die den Suchfeldwert K_i haben.

   Im Fall eines **geclusterten** Indexes wird Pr_i direkt durch den Datensatz oder die Datensätze, die den Suchfeldwert enthalten, ersetzt.
10. Jedes Blatt außer der Wurzel (wenn sie Blatt ist) hat mindestens $\frac{p_{Blatt}}{2}$ Werte.
11. Alle Blätter sind auf derselben Stufe.
12. Ein $P_{previous}$ kann auch eingebaut werden.

Vorgehensweise beim Einfügen eines neuen Elements: Man versucht das neue Element in das richtige Blatt einzufügen. Passt es nicht mehr rein, dann wird das Blatt aufgespalten, wobei der mittlere Schlüssel nach oben wandert. Das kann möglicherweise zur Spaltung des betreffenden Knotens führen.

Zur Implementation von Algorithmen muss eventuell noch weitere Information (Zähler, Knotentyp usw.) verwaltet werden.

Variationen: Anstatt der Halbvoll-Forderung können andere Füllfaktoren angesetzt werden, etwa, dass man auch fast leere Knoten zulässt, ehe sie rekombiniert werden. Letztere Möglichkeit scheint bei zufälligen Einfügungen und Löschungen nicht allzuviel Platz zu verschwenden.

B^+-Bäume werden manchmal für die primäre Dateistruktur verwendet (index-organisierte Tabelle oder geclusterter Index). In dem Fall werden ganze Datensätze in den Blättern gespeichert. Das geht bei wenigen Datensätzen und/oder kleinen Datensätzen gut. Ansonsten wird die Anzahl der Stufen oder die Blattgröße für einen effizienten Zugriff zu groß.

Gegenüber B-Bäumen haben B^+-Bäume folgende Vorteile:

- p_{Knoten} kann in den inneren Knoten wesentlich größer sein als p_{Blatt}. Dadurch hat der Baum weniger Stufen.
- Die Verkettung der Blätter erlaubt schnellen Zugriff in der Reihenfolge des Suchfeldes.

B^*-Bäume und $B^\#$-Bäume sind Varianten des B^+-Baums, bei denen ein voller Knoten nicht sofort aufgespalten wird, sondern zunächst versucht wird, die Einträge auf benachbarte Knoten umzuverteilen. Falls dies nicht möglich ist, wird nicht *ein* Knoten in *zwei* aufgespalten, sondern *zwei* Knoten in *drei* ([34] Seite 150).

B^+-Bäume werden oft zur Verwaltung von BLOBs verwendet. Dabei werden Positionen oder Offsets im BLOB als Schlüsselwerte verwendet ***(positional B^+-tree)***.

Tries, Patricia-Bäume und Präfix-Bäume sind Varianten, die bei Zeichenketten als Suchwerte in den inneren Knoten Speicher- und Vergleichsvorteile haben, da in den inneren Knoten nur Teile des Suchwertes gespeichert werden müssen. Wir gehen hier nicht näher darauf ein ([34] Seite 152).

7.3.4 Andere Indextypen

Es ist möglich Zugriffsstrukturen zu entwerfen, die auf dem Hashing beruhen. Die Indexeinträge $\langle K, Pr\rangle$ oder $\langle K, P\rangle$ können als eine dynamisch expandierbare Hash-Datei organisiert werden.

Bisher sind wir davon ausgegangen, dass die Indexeinträge physikalische Adressen von Plattenblöcken ***(physical index)*** spezifizieren mit dem Nachteil, dass die Zeiger geändert werden müssen, wenn ein Datensatz an eine andere Stelle auf der Platte bewegt wird, was bei Hash-Dateien häufig vorkommt (Aufteilung eines Eimers). Die Nachführung der Zeiger ist nicht einfach.

Dem Problem kann mit einem **logischen Index** ***(logical index)*** begegnet werden, der Einträge $\langle K, K_p\rangle$ hat, wobei K_p der Wert des Feldes ist, das für die primäre Dateiorganisation verwendet wird. Mit dem sekundären Index wird das K_p bestimmt. Dann wird unter Verwendung der primären Dateiorganisation die eigentliche Block- oder Datensatzadresse bestimmt.

Eine sortierte Datei mit einem mehrstufigen primären Index auf dem Sortierschlüsselfeld heißt **indexierte sequentielle Datei** ***(indexed sequential file)***. Einfügen wird durch eine Overflow-Datei behandelt, die periodisch eingearbeitet wird, wobei der Index neu aufgebaut wird. IBM's ***indexed sequential access method*** (ISAM) benutzt einen zweistufigen Index mit enger Plattenbeziehung: Die erste Stufe ist ein **Zylinderindex**, der den Schlüsselwert eines Ankerdatensatzes für jeden Plattenzylinder und einen Zeiger auf den **Spurindex** für diesen Zylinder verwaltet. Der Spurindex hat den Schlüsselwert eines Ankerdatensatzes für jede Spur des Zylinders und einen Zeiger auf die Spur. Die Spur kann dann sequentiell nach dem Block und dem Datensatz abgesucht werden.

Eine andere IBM-Methode ***virtual storage access method*** (VSAM) ist ähnlich einer B^+-Baum-Zugriffsstruktur.

7.4 Objektballung, Cluster

Man versucht, Datensätze, die oft gemeinsam benutzt werden, in einem oder in benachbarten Blöcken unterzubringen ([34] Seite 192).

7.5 Indexierung mit SQL

Die meisten Systeme bieten ein Kommando mit folgender Syntax an:

`CREATE [UNIQUE] INDEX` *index-name*
`ON` *base-table* `(`*column-komma-list*`)`

Beispiel:

```
CREATE UNIQUE INDEX SNRINDEX ON S (SNR);
```

In der Tabelle `S` wird auf der Spalte `SNR` ein Index angelegt. Wenn ein Index aus mehreren Spalten zusammengesetzt ist, heißt er **zusammengesetzt** ***(composite)***. Wenn `UNIQUE` spezifiziert ist, dann wird nach Erstellung des Indexes dafür gesorgt, dass die Zeilen bezüglich der Index-Spalten

eindeutig bleiben. Wenn der Index für eine Tabelle nachträglich definiert wird, dann wird die Definition des Indexes im Fall von Duplikaten bezüglich der Index-Spalten zurückgewiesen.

Manche Systeme bieten an, sogenannte **geclusterte Indexe** anzulegen, bei denen die Daten auch physikalisch nach dem Index sortiert werden, d.h. die ganze Tabelle wird in den Index aufgenommen. Das bedeutet natürlich, dass es pro Tabelle nur *einen* geclusterten Index geben kann. Ein geclusterter Index erhöht die Zugriffsgeschwindigkeit über die Index-Spalte, verlangsamt aber Aktualisierungs-Operationen. Da Many-To-Many-Beziehungen oft keine oder nur wenige Daten enthalten, eignen sie sich besonders gut für einen geclusterten Index.

Generell erhöht ein Index die Zugriffsgeschwindigkeit bei Suchoperationen – oft drastisch – und verringert die Geschwindigkeit bei Aktualisierungen, d.h. Einfüge- und Löschoperationen, da ja der Index mitverwaltet werden muss.

Folgende Regeln ergeben sich bezüglich der Erstellung von Indexen:

- Spalten, die Primärschlüssel sind, sollten einen `UNIQUE` Index haben.
- Spalten, die regelmäßig in Joins verwendet werden, sollten einen Index haben (Fremdschlüssel- und Schlüssel-Spalten).
- Eine Spalte, über deren Werte oft zugegriffen wird, sollte einen Index haben.
- Spalten, bei denen oft Wertebereiche abgesucht werden, sollten einen Index haben. Bereichs-Anfragen sind mit geclusterten Indexen performanter. Aber es ist nur *ein* geclusterter Index pro Tabelle möglich.
- Spalten, deren Wertebereich nur wenige Werte enthält (z.B. ja/nein oder männlich/weiblich/unbekannt, allgemein: Attribute mit geringer Selektivität), sollten keinen Index haben.
- Kleine Tabellen mit wenigen – bis zu 1000 – Zeilen profitieren nicht von einem Index, da das System in diesen Fällen meistens einen ***table scan*** macht.
- Indexfelder sollten möglichst kompakt sein, d.h. der Suchschlüssel sollte aus wenigen Spalten bestehen und/oder wenige Bytes lang sein.

7.6 Bemerkungen

7.6.1 Index Intersection

Manche DBS erlauben mehrere Indexe zu schneiden, um Mehr-Attribut-Anfragen effektiv durchzuführen. Die resultierenden TID-Mengen werden geschnitten. Der Schnitt zweier Mengen hat Komplexität $O(n^2)$ oder, wenn die Mengen vorher sortiert werden, $O(n \log n)$.

Anfragen können ohne Zugriff auf die indexierte Tabelle eventuell performanter durchgeführt werden. Dazu ein kurzes Beispiel:

Die Tabelle T habe einen Index I_A auf Spalte A und einen Index I_B auf Spalte B. Betrachten wir nun folgende Anfrage:

```
SELECT * FROM T WHERE A = 'aWert' AND B = 'bWert'
```

Jeder der beiden Indexe I_A und I_B liefert für die vorgegebenen Werte eine TID-Menge. Die beiden Mengen werden geschnitten und liefern eine resultierende TID-Menge.

7.6.2 Geclusterte Indexe

Wenn für eine Tabelle ein geclusterter und mehrere nicht-geclusterte Indexe angelegt werden sollen, dann sollte man mit dem geclusterten Index beginnen, da in diesen ja die ganze Tabelle aufgenommen wird. (Die ursprüngliche Tabelle wird dabei – wahrscheinlich – gelöscht.) Danach können die nicht-geclusterten Indexe angelegt werden. Bei dieser Vorgehensweise wird die Tabelle auf der Platte wahrscheinlich in der Sortierreihenfolge des geclusterten Indexes angelegt, was deutliche Disk-I/O-Vorteile zur Folge hat.

Zur Realisierung einer Many-to-Many-Beziehung muss ja eine zusätzliche Tabelle angelegt werden, die u.U. nur zwei Fremdschlüssel-Spalten und keine weiteren Daten enthält. Hier bietet sich idealerweise ein geclusterter Index an. Wenn in beiden Richtungen navigiert werden soll, dann sollte man eigentlich zwei geclusterte Indexe auf derselben Tabelle haben, was normalerweise nicht möglich ist oder was die DBSs wohl nicht anbieten. (Dann muss man eventuell zwei Tabellen (mit identischem Inhalt) für die Beziehung anlegen, dort je einen geclusterten Index definieren und selbst für die Konsistenz dieser Tabellen sorgen.)

Nehmen wir an, dass wir zwei Tabellen A und B mit jeweils etwa 100.000 Tupeln und eine Many-to-Many-Beziehung zwischen den beiden Tabellen haben, wobei ein $a \in A$ zu 100 bis 1000 $b \in B$ in Beziehung steht. Dann hätte die Beziehungstabelle größenordnungsmäßig 10 bis 100 Millionen Einträge. In solch einem Fall kann sich eine redundante Beziehungstabelle durchaus lohnen.

7.6.3 Unique Indexe

Unique Indexe werden angelegt um die Eindeutigkeit eines Attributs zu erzwingen.

7.6.4 Indexe auf berechneten Spalten

Manche DBS bieten Indexe auf berechneten Spalten (sogenannte Funktionsindexe) an, wenn der Berechnungsausdruck deterministisch und präzise ist und nicht Texte oder Bilder als Resultat hat. Die Datums- oder Zeitfunktion ist sicher nicht deterministisch, da sie bei gleichen Eingabeparametern unterschiedliche Werte liefern kann. Float-Datentypen sind nicht präzise.

7.6.5 Indexierte Views

Indexierte Views sind solche, deren Werte in der Datenbank persistent gemacht werden. Die Wartung solcher Views kann bei vielen Updates der zugrunde liegenden Tabellen teuer werden. Das kann den Performanzgewinn zunichte machen.

Der erste Index auf einem View wird wahrscheinlich ein geclusterter Index sein, um die Daten des Views persistent zu machen. Danach können beliebig viele nicht-geclusterte Indexe angelegt werden.

7.6.6 Covering Index

Wir sprechen von einem ***covering index*** für eine spezielle Anfrage, wenn alle Attribute, die in dieser Anfrage (bei Projektion und Selektion) verwendet werden, indexiert sind. Damit kann diese spezielle Anfrage oder dieser Anfragetyp wesentlich beschleunigt werden.

7.7 Übungen

7.7.1 B-Baum, B^+-Baum

Fügen Sie in einen B-Baum der Ordnung 3 oder einen B^+-Baum der Ordnung 3 (innere Knoten), Ordnung 2 (Blätter) nacheinander die Buchstaben

A L G O R I T H M U S

ein. Zeigen Sie nach jedem Schritt den ganzen Baum.

Fügen Sie diese Buchstaben in aufsteigender Reihenfolge ein, also:

A G H I L M O R S T U

7.7.2 Angestellten-Verwaltung

Angenommen die Block- und Seitengröße sei 512 Byte. Ein Zeiger auf einen Block (Blockadresse) sei $P = 6$ Byte lang, ein Zeiger auf einen Datensatz (Datensatzadresse) sei $P_r = 7$ Byte lang.

Eine Datei hat $r = 30000$ ANG-Datensätze (Angestellte) fester Länge. Jeder Datensatz hat die Felder:

NAM (30 Byte, Name),
ANR (9 Byte, Angestelltennummer),
ABT (9 Byte, Abteilung),
ADR (40 Byte, Adresse),
TEL (9 Byte, Telefon),
GEB (8 Byte, Geburtsdatum),
GES (1 Byte, Geschlecht),
BER (4 Byte, Berufscode),
GEH (4 Byte, Gehalt)

1. Berechnen Sie die Datensatzlänge.
2. Berechnen Sie den Blockungsfaktor und die Anzahl der benötigten Blöcke bei Nicht-Spannsatz-Organisation.
3. Angenommen die Datei sei nach dem Schlüsselfeld ANR geordnet. Wir wollen einen Primärindex auf ANR konstruieren.
   (a) Berechnen Sie den Index-Blockungsfaktor f_I (gleich dem *index fan-out* f_o).

(b) Berechnen Sie die Anzahl der Einträge im Primärindex und die Anzahl der benötigten Blöcke.

(c) Berechnen Sie die Anzahl der benötigten Stufen, wenn wir den Index zu einem mehrstufigen Index machen, und die Gesamtzahl der für den Index verwendeten Blöcke.

(d) Berechnen Sie die Anzahl der Blockzugriffe, wenn ein Datensatz mit Hilfe der ANR gesucht wird.

4. Angenommen die Datei sei **nicht** nach ANR geordnet. Wir wollen einen Sekundär-Index auf ANR konstruieren.

   (a) Berechnen Sie den Index-Blockungsfaktor f_I (gleich dem *index fan-out* f_o).

   (b) Berechnen Sie die Anzahl der Einträge im Index erster Stufe und die Anzahl der benötigten Blöcke.

   (c) Berechnen Sie die Anzahl der benötigten Stufen, wenn wir den Index zu einem mehrstufigen Index machen, und die Gesamtzahl der für den Index verwendeten Blöcke.

   (d) Berechnen Sie die Anzahl der Blockzugriffe, wenn ein Datensatz mit Hilfe der ANR gesucht wird.

5. Angenommen die Datei sei bezüglich des (Nicht-Schlüssel-) Feldes ABT *nicht* geordnet. Wir wollen einen Sekundär-Index auf ABT konstruieren, wobei die dritte Option (ein Eintrag pro Indexfeldwert, Liste von Blöcken, die Datensatzzeiger enthalten) verwendet werden soll. Angenommen es gäbe 1000 unterschiedliche ABT-Nummern und die ANG-Datensätze seien gleichmäßig über diese Abteilungen verteilt.

   (a) Berechnen Sie den Index-Blockungsfaktor und die Anzahl der auf der ersten Stufe benötigten Blöcke.

   (b) Berechnen Sie die Anzahl der Blöcke, die benötigt werden, um die Datensatz-Zeiger zu speichern.

   Blöcke benötigt.

   (c) Wieviele Stufen werden benötigt, wenn wir den Index mehrstufig machen?

   (d) Wieviele Blöcke werden insgesamt für diesen Index benötigt?

   (e) Wieviele Blockzugriffe werden im Durchschnitt benötigt, um alle zu einer bestimmten ABT gehörenden Datensätze zu lesen?

6. Angenommen die Datei sei nach dem (Nicht-Schlüssel-)Feld ABT geordnet. Wir wollen einen Cluster-Index auf ABT konstruieren, wobei Block-Anker (D.h. jeder neue Wert von ABT beginnt mit einem neuen Block.) verwendet werden. Angenommen es gäbe 1000 unterschiedliche ABT-Nummern und die ANG-Datensätze seien gleichmäßig über diese Abteilungen verteilt.

   (a) Berechnen Sie den Index-Blockungsfaktor und die Anzahl der auf der ersten Stufe benötigten Blöcke.

   (b) Wieviele Stufen werden benötigt, wenn wir den Index mehrstufig machen?

   (c) Wieviele Blöcke werden insgesamt für diesen Index benötigt?

   (d) Wieviele Blockzugriffe werden im Durchschnitt benötigt, um alle zu einer bestimmten ABT gehörenden Datensätze zu lesen?

7. Angenommen die Datei sei *nicht* nach dem Schlüsselfeld ANR sortiert und wir wollten eine B^+-Baum-Zugriffsstruktur auf ANR konstruieren.

   (a) Berechne den Grad p_{Knoten} des Baumes und p_{Blatt}.

   (b) Wieviele Blattknoten werden benötigt, wenn ein Füllungsgrad von 69% gewünscht wird?

   (c) Wieviele Stufen von internen Knoten werden benötigt, wenn auch die internen Knoten nur zu 69% gefüllt sein sollen?

   (d) Wieviele Blöcke werden für den Index insgesamt benötigt?

   (e) Wieviele Blockzugriffe sind nötig, um einen Datensatz zu lesen, wenn ANR gegeben ist?

   (f) Führen Sie diese Analyse für einen geclusterten Index durch.

8. Stelle dieselben Überlegungen für einen B-Baum an und vergleiche die Resultate.

   (a) Berechne den Grad p des Baumes.

   (b) Wieviele Stufen und Knoten werden benötigt, wenn ein Füllungsgrad von 69% gewünscht wird?

   (c) Wieviele Blöcke werden für den Index insgesamt benötigt?

   (d) Wieviele Blockzugriffe sind nötig, um einen Datensatz zu lesen, wenn ANR gegeben ist?

   (e) Vergleich B- und B^+-Baum:

Kapitel 8

Mehrdimensionale Zugriffsstrukturen

Mehrdimensionale Zugriffsstrukturen oder Indexe gibt es bei den gängigen kommerziellen Systemen eher noch als Ausnahme oder Besonderheit.

Hier beschränken wir uns auf die Nennung von verschiedenen Verfahren. Einigen wird auch ein Abschnitt gewidmet. Details sind bei [34] (Seite 175) und der dort genannten Literatur[18] zu finden.

- KdB-Baum
- Grid-File
- Raumfüllende Kurven
- R-Baum
- Geometrische Zugriffsstrukturen

8.1 KdB-Baum

Ein KdB-Baum (k-dimensionaler B-Baum) besteht aus

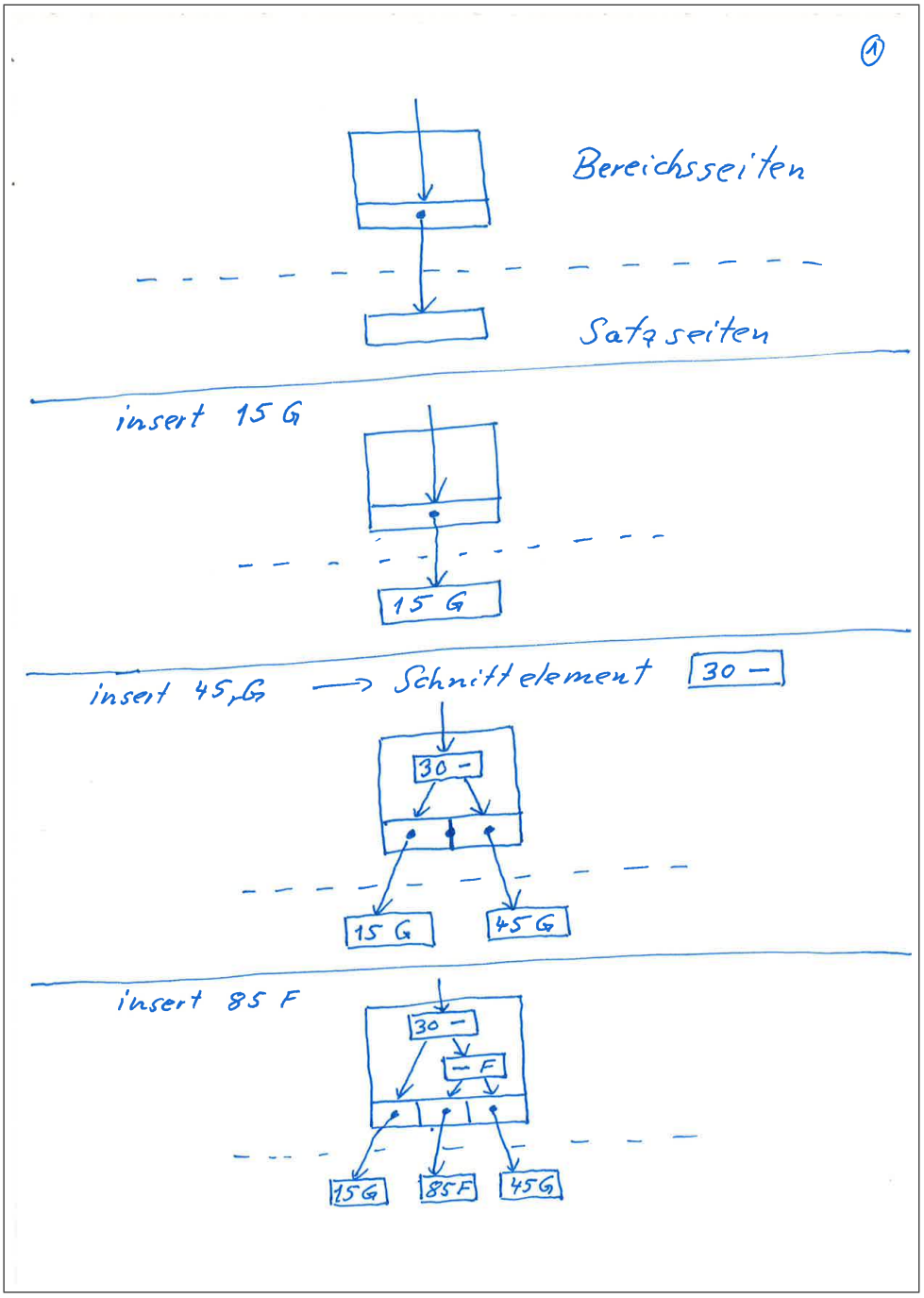
- inneren Knoten (**Bereichsseiten**), die einen binären kd-Baum mit maximal b Knoten (**Schnittelementen**) enthalten, und
- Blättern (**Satzseiten**), die bis zu t Tupel der gespeicherten Relation enthalten.

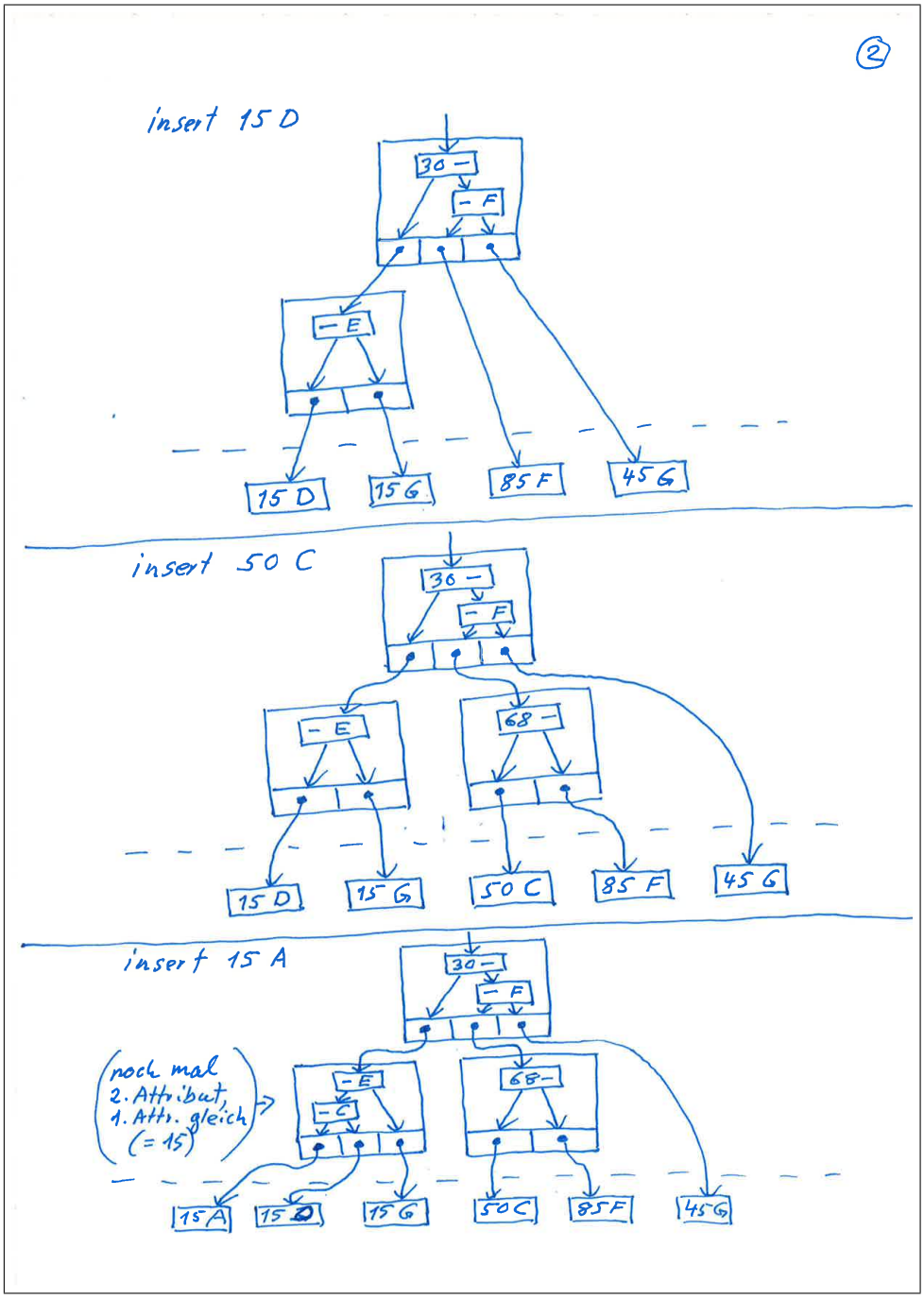
k ist die Anzahl der Zugriffs-Attribute. Ein Schnittelement teilt den Baum bezüglich eines Zugriffs-Attributs. Die Normalstrategie ist, dass die Zugriffs-Attribute in den Schnittelementen zyklisch alteriert werden.

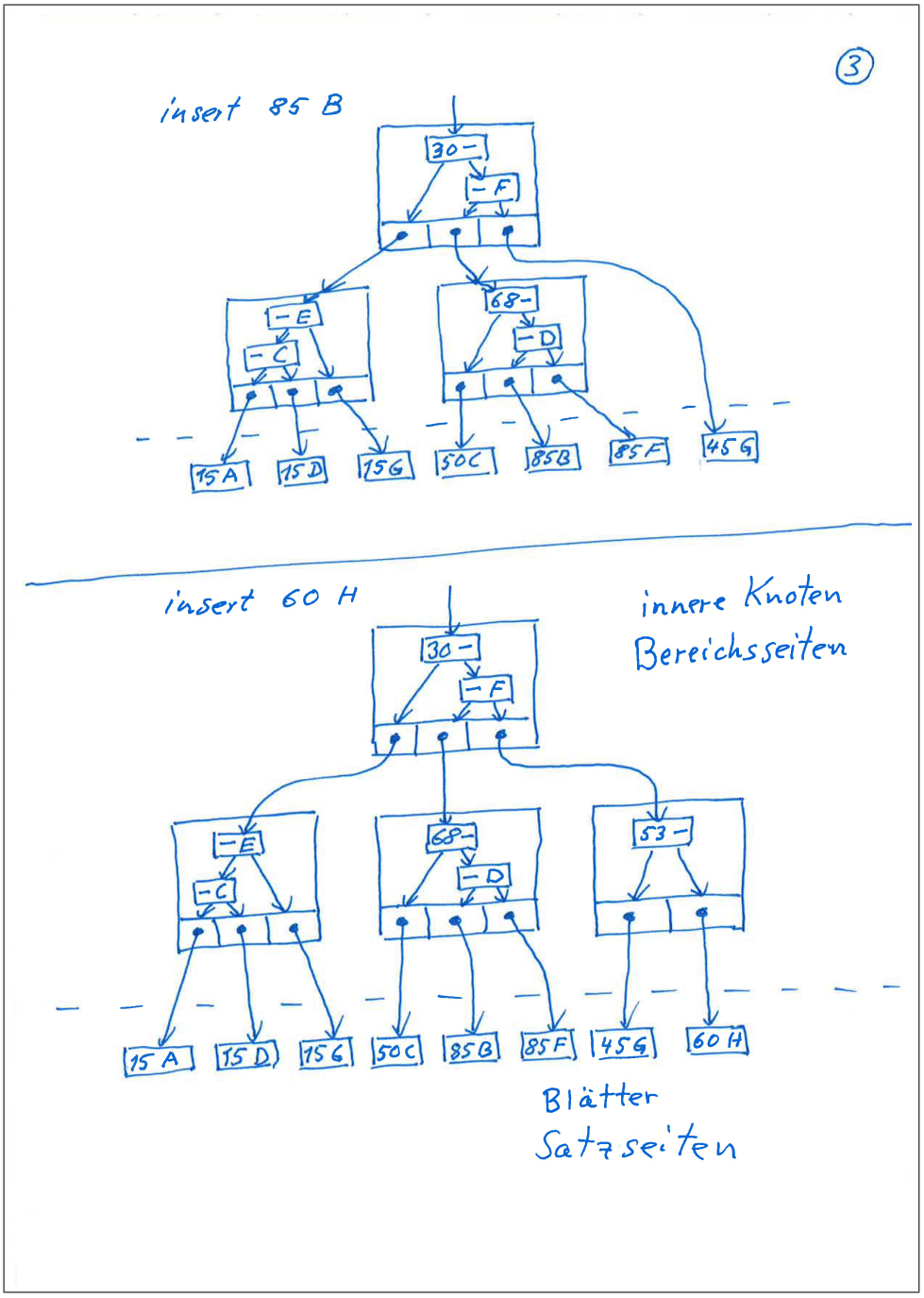
Auf die komplizierten Mechanismen zur Ausgleichung des Baumes kann hier nicht eingegangen werden.

Den Aufbau des Indexes werden wir an Hand eines Beispiels erläutern.

$k = 2$ (2 Dimensionen)
$t = 1$ (Satzseiten enthalten höchsten ein Tupel.)
$b = 2$ (Bereichsseiten enthalten höchstens 2 Schnittelemente.)
Datensätze mit numerischem und alphabetischem Attribut:
`15 G | 45 G | 85 F | 15 D | 50 C | 15 A | 85 B | 60 H`


①
Bereichsseiten
Satzseiten
insert 15 G
15 G
insert 45 G
Schnittelement
30 –
30 –
15 G
45 G
insert 85 F
30 –
– F
15 G
85 F
45 G


2
insert 15 D
30 –
– F
– E
15 D
15 G
85 F
45 G
insert 50 C
30 –
– F
– E
68 –
15 D
15 G
50 C
85 F
45 G
insert 15 A
30 –
– F
– E
– C
68 –
noch mal
2. Attribut,
1. Attr. gleich
(= 15)
15 A
15 D
15 G
50 C
85 F
45 G


3
insert 85 B
30-
-F
-E
-C
68-
-D
15A
15D
15G
50C
85B
85F
45G
insert 60 H
innere Knoten
Bereichsseiten
30-
-F
-E
-C
68-
-D
53-
15 A
15 D
15 G
50C
85B
85F
45G
60 H
Blätter
Satzseiten

Eine eventuell bessere Veranschaulichung des KdB-Baums ist die "Brickwall"-Darstellung, die wir im folgenden mit den Satzseiten-Größen ein und drei Tupel zeigen werden.

Bemerkungen:

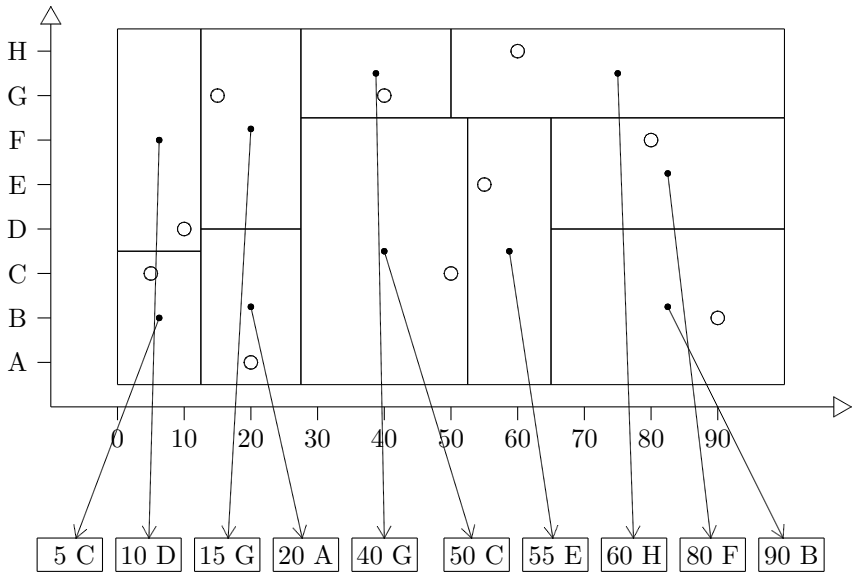
1. Jeder Schritt des Aufbaus der Brickwall (Historie der Zellteilung) muss verwaltet werden, um Datensätze zu finden, einzufügen oder zu löschen. D.h. jede innere Brick-Wand entspricht einem Schnittelement.
2. Die Brickwall besteht aus Quadern sehr unterschiedlicher Größe und hat daher eine unregelmäßige Struktur, die es nicht erlaubt, Hyperebenen oder Regionen für die Suche von Objekten zu definieren.

Wir füllen einen zweidimensionalen KdB-Baum nacheinander mit zweidimensionalen Datensätzen bestehend aus einer ganzen Zahl und einem Großbuchstaben.

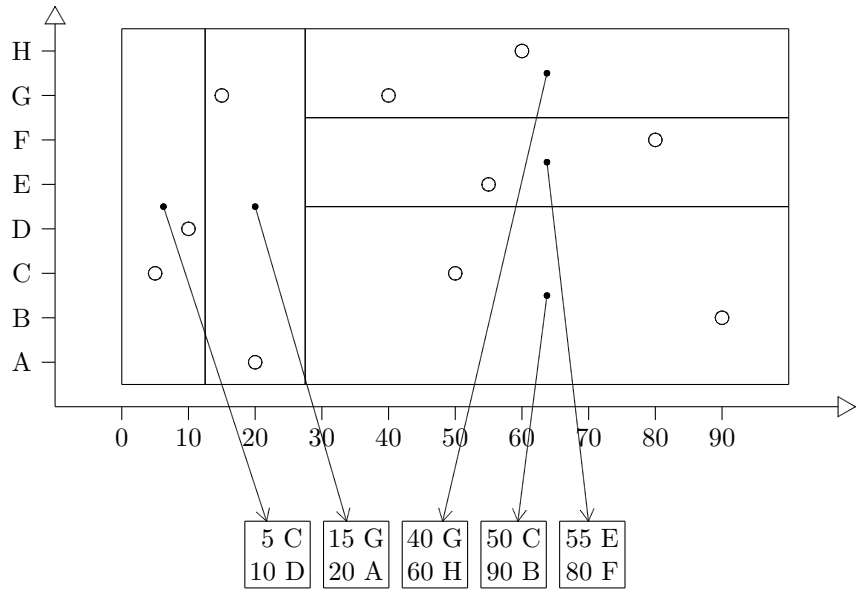
$k = 2$ (2 Dimensionen)
$t = 1$ (Satzseiten enthalten höchsten ein Tupel.)
Datensätze mit numerischem und alphabetischem Attribut:

```
15 G | 40 G | 80 F | 10 D | 50 C | 20 A | 90 B | 60 H | 65 E |  5 C
```

Brickwall (auf eine Seite passt nur ein Datensatz):



Brickwall (eine Seite enthält bis 3 Datensätze):



8.2 Grid-File

Anstatt des unregelmäßigen "Brickwall"-Musters eines KdB-Baums wird der Datenraum gleichmäßig in Grid-Zellen eingeteilt.

- **Skala:** Für jede Dimension gibt es ein Feld von Intervallen des Wertebereichs eines Zugriffs-Attributs.
- Das **Grid-Directory** zerlegt den Datenraum in **Grid-Zellen** (k-dimensionale Quader) gemäß den Intervallen der Skalen.

  Das Grid-Directory ist i.a. so groß, dass es normalerweise auf dem Sekundärspeicher liegt. Allerdings genügen wenige Zugriffe, um eine Zelle zu finden. Bei einer ***exact-match query*** genügt ein Zugriff.
- Die **Grid-Region** besteht aus einer oder mehreren Grid-Zellen. Jeder Grid-Region ist eine **Satzseite** zugeordnet. Die Grid-Region ist ein k-dimensionales *konvexes* Gebilde, d.h. jeder Datensatz auf der Geraden zwischen zwei Datensätzen derselben Region liegt ebenfalls in der Region. Regionen sind paarweise disjunkt.
- Die **Satzseiten** ***(data bucket)*** enthalten die Datensätze und liegen auf dem Sekundärspeicher.

Wir füllen einen zweidimensionalen Grid-File nacheinander mit zweidimensionalen Datensätzen bestehend aus einer ganzen Zahl und einem Großbuchstaben.

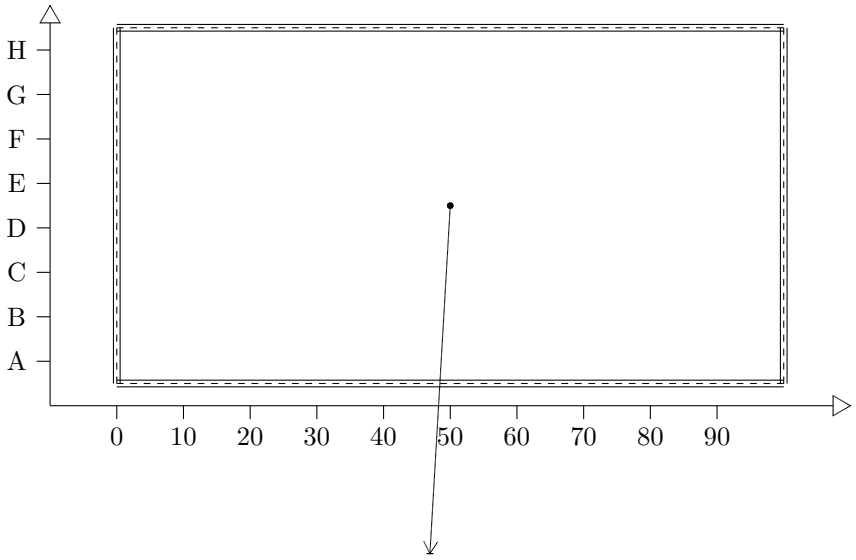
$k = 2$ (2 Dimensionen)
$t = 1$ (Satzseiten enthalten höchsten ein Tupel.)
Datensätze mit numerischem und alphabetischem Attribut:

```
15 G | 40 G | 80 F | 10 D | 50 C | 20 A | 90 B | 60 H | 65 E |  5 C
```

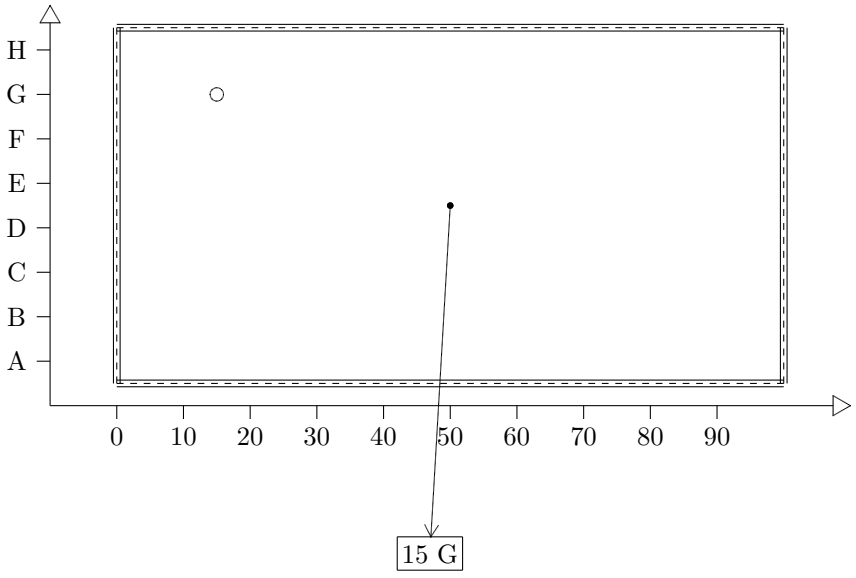
Eine Seite enthält bis 1 Datensätze:

Auf eine Seite passt nur ein Datensatz.

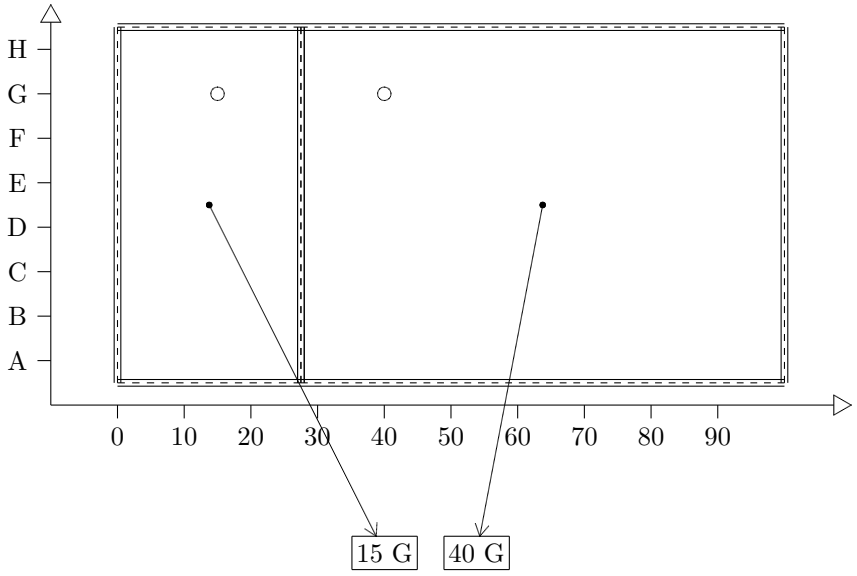
Der leere Grid-File:



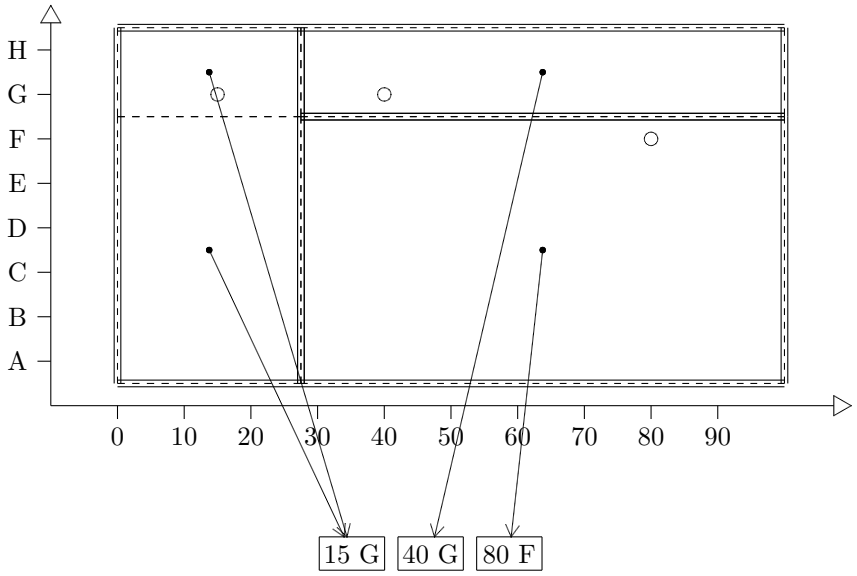
Einfügen von `(15 G)`:



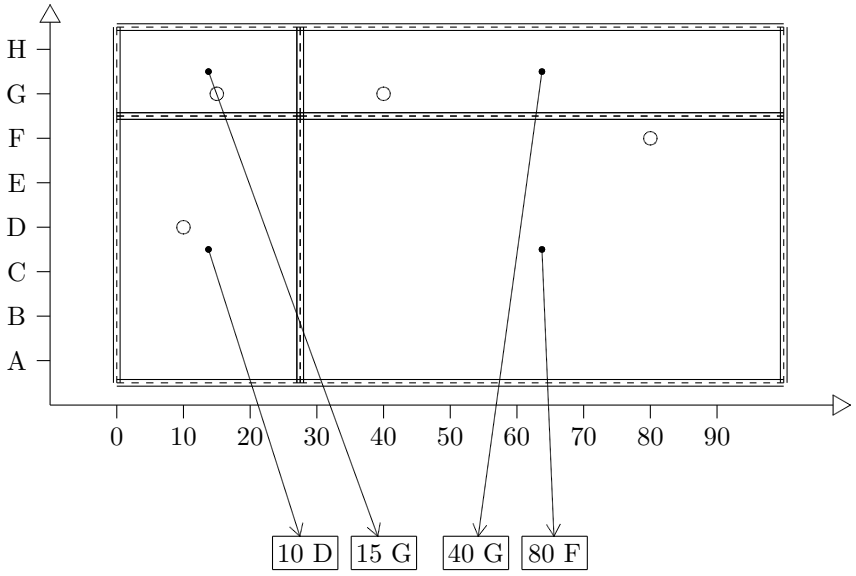
Einfügen von `(40 G)`:



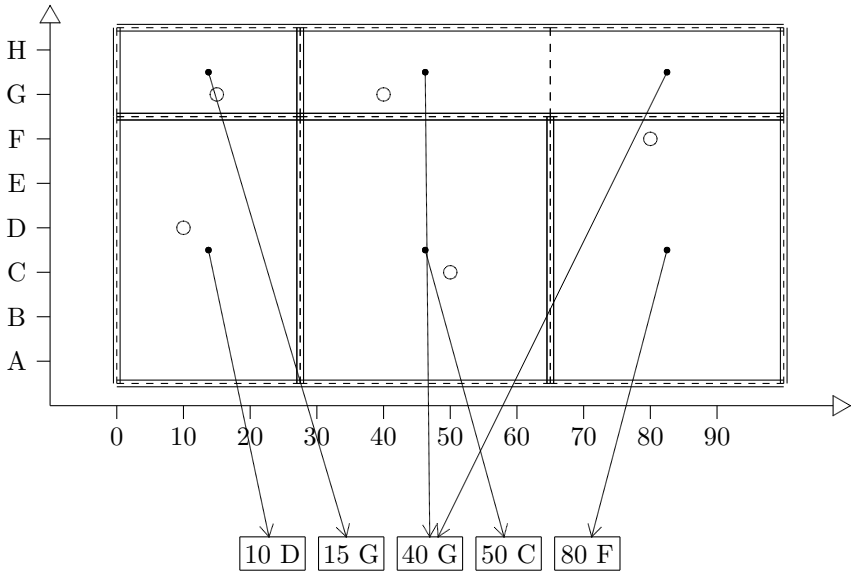
Einfügen von `(80 F)`:



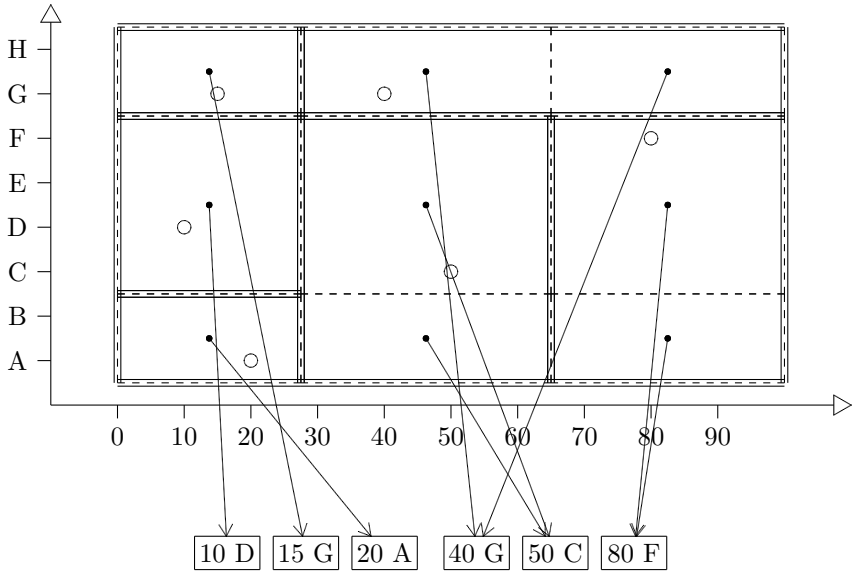
Einfügen von `(10 D)`:



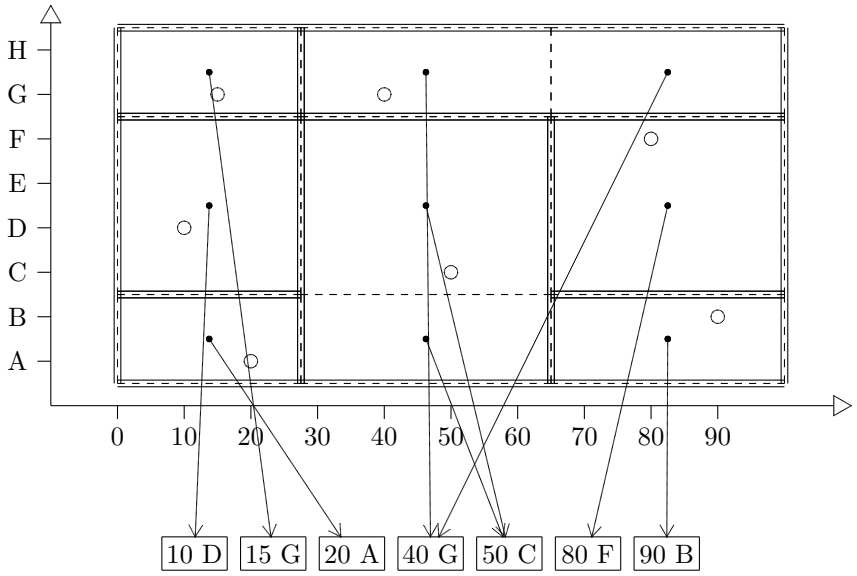
Einfügen von `(50 C)`:



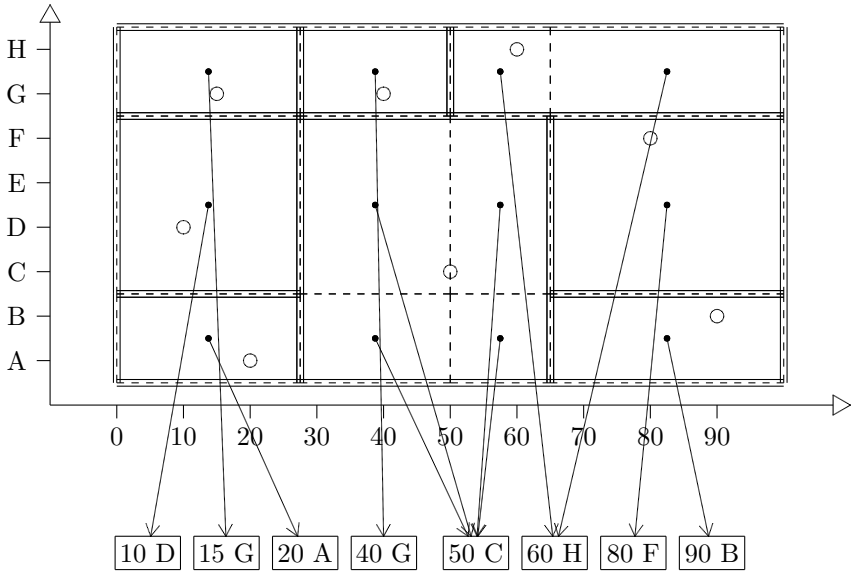
Einfügen von `(20 A)`:



Einfügen von `(90 B)`:

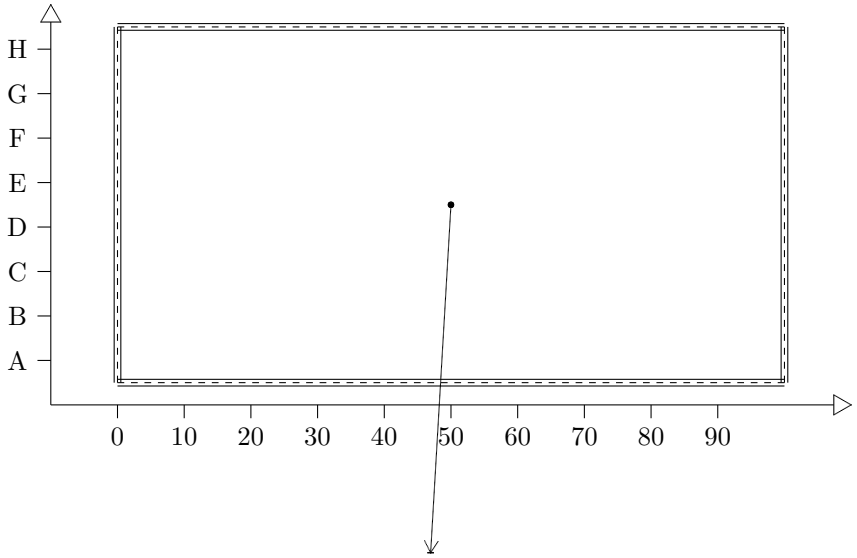


Einfügen von `(60 H)`:

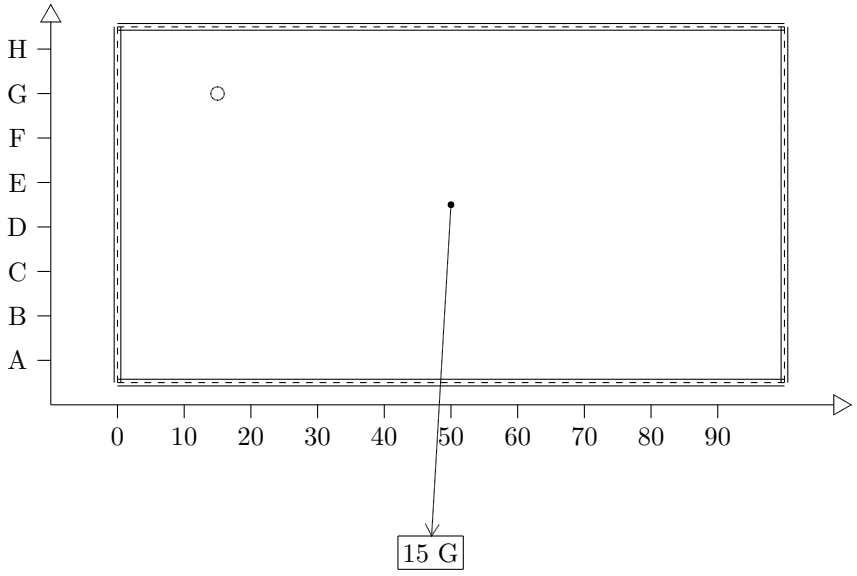


Eine Seite enthält bis 3 Datensätze:

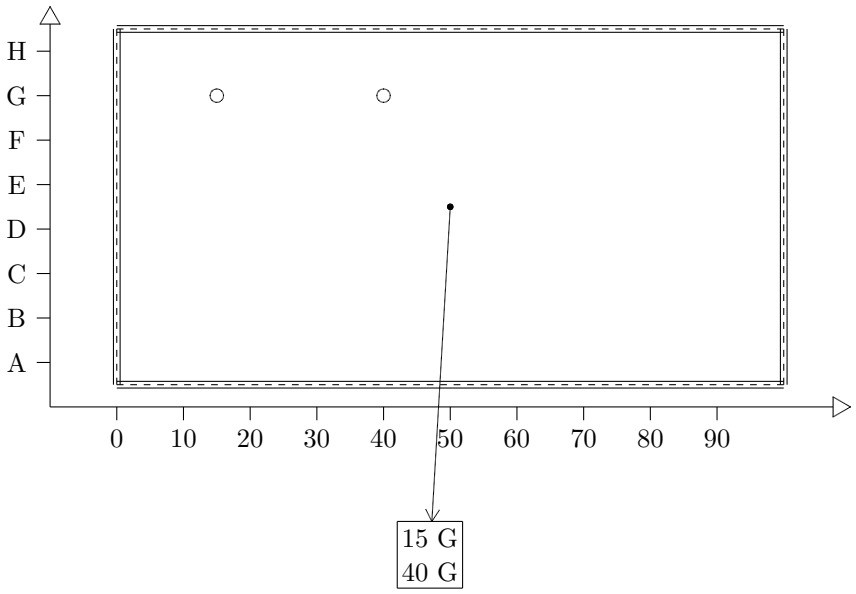
Der leere Grid-File:



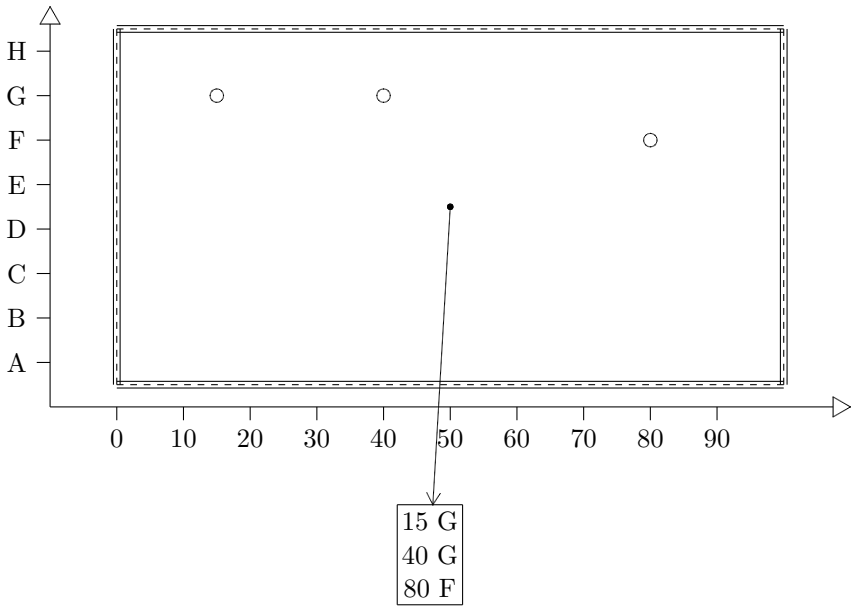
Einfügen von `(15 G)`:



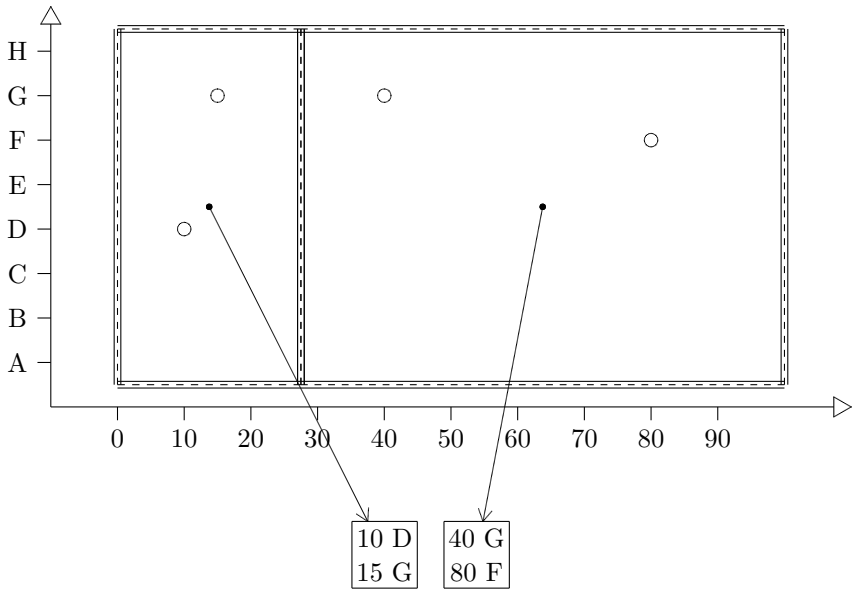
Einfügen von `(40 G)`:



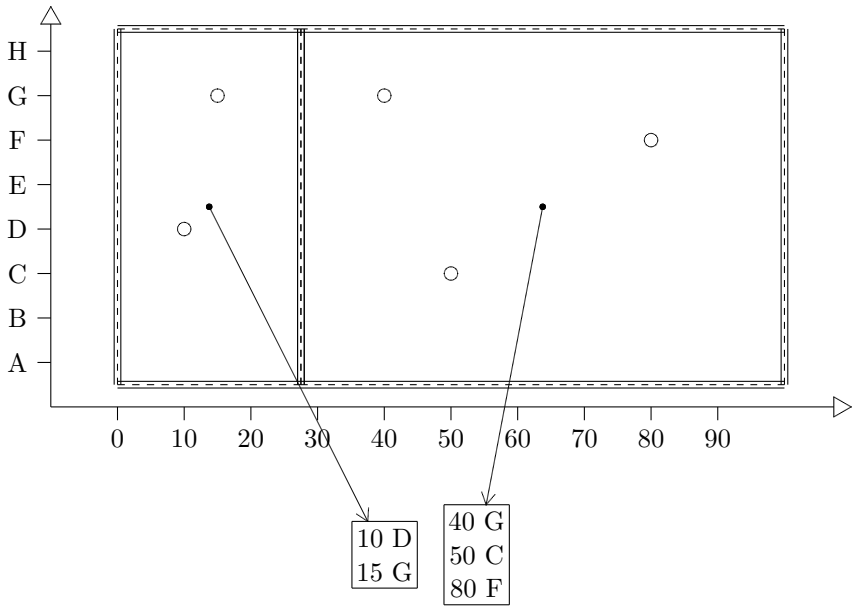
Einfügen von `(80 F)`:



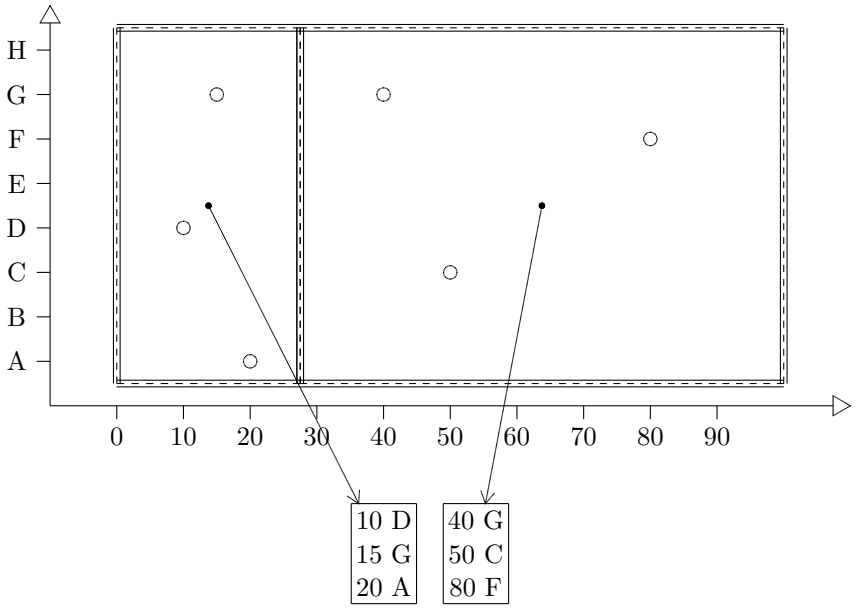
Einfügen von `(10 D)`:



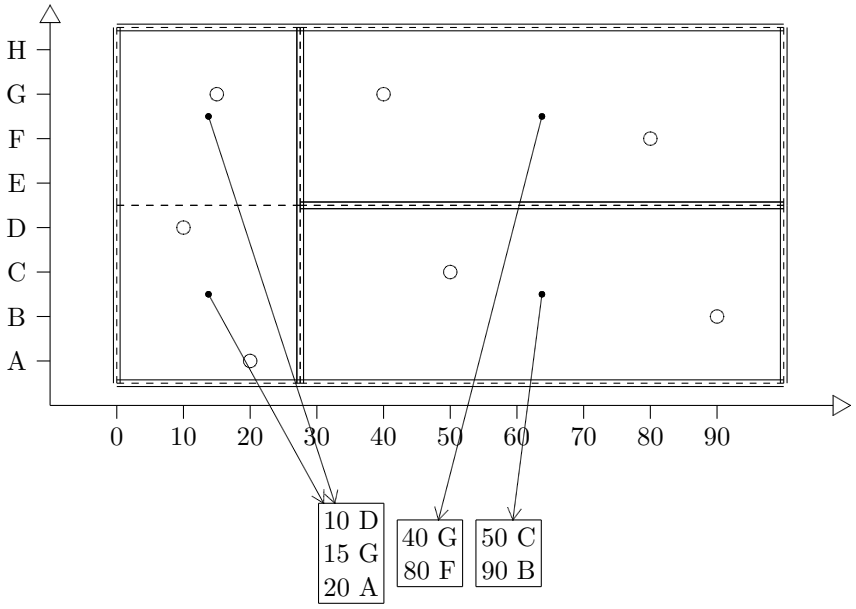
Einfügen von `(50 C)`:



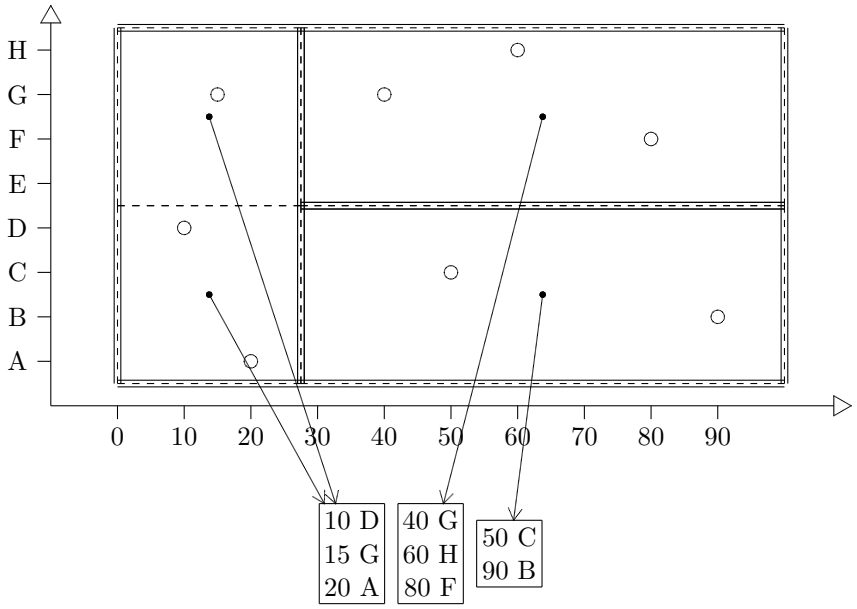
Einfügen von `(20 A)`:



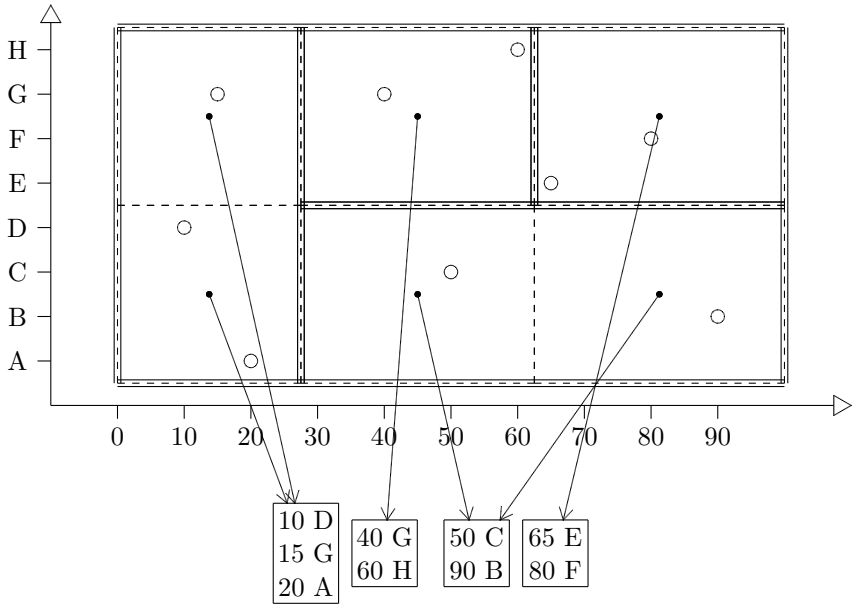
Einfügen von `(90 B)`:



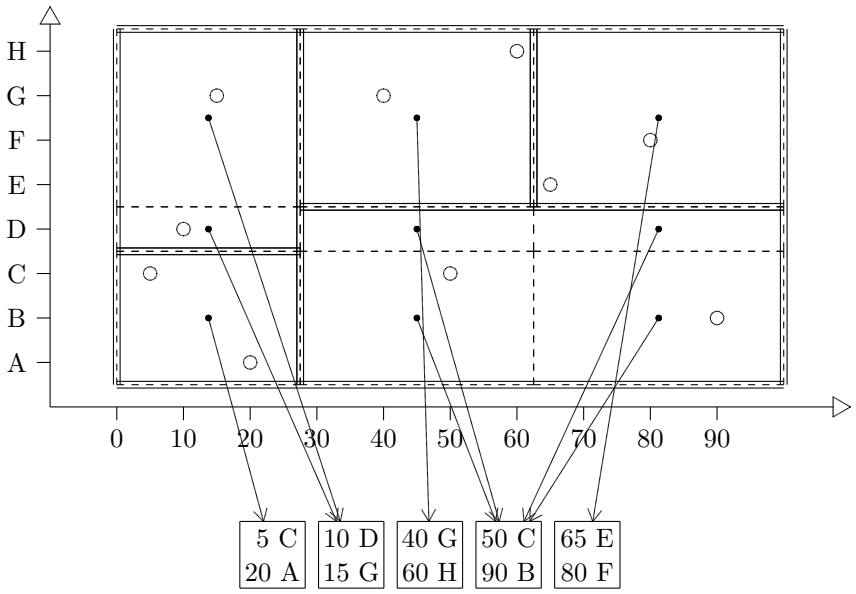
Einfügen von `(60 H)`:



Einfügen von `(65 E)`:



Einfügen von `( 5 C)`:

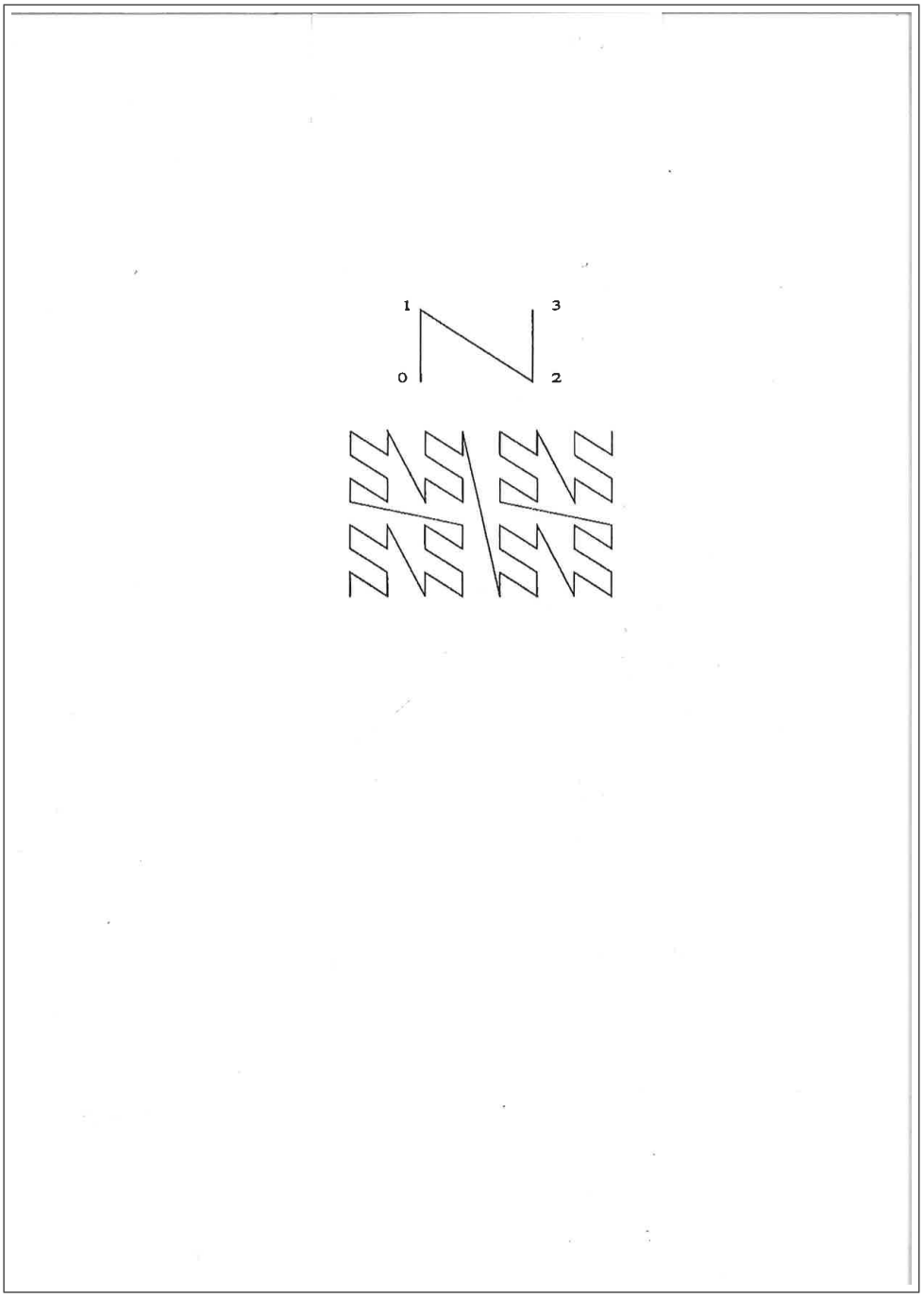


8.3 Raumfüllende Kurven

Warum die Entwicklung von mehrdimensionalen Zugriffsmethoden schwierig ist, liegt im wesentlichen daran, dass es keine totale Ordnung gibt, d.h. keine Abbildung in den ein-dimensionalen Raum, die die räumliche Nähe erhält. Denn dann könnte man wieder B^+-Baumstrukturen zur Indexierung verwenden.

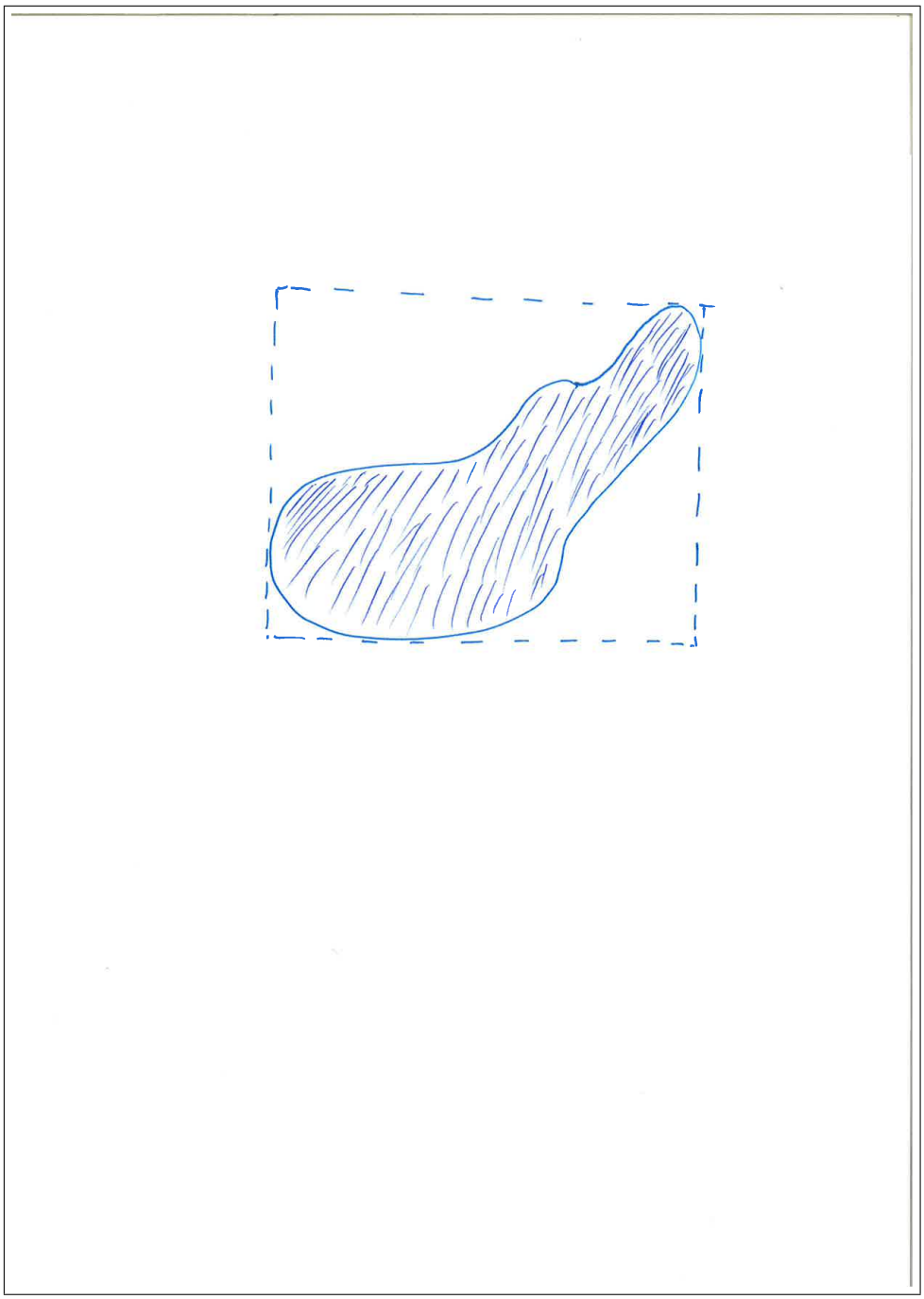
Es gibt einige Methoden, die die räumlichen Beziehungen wenigstens teilweise erhalten. Allen gemeinsam ist, dass sie das k-dimensionale Universum in Zellen eines Gitters partitionieren, und die Zellen so durchnummerieren, dass Nachbarschaftsbeziehungen meistens erhalten bleiben.

Als Beispiel stellen wir hier die Z-Kurve vor (Peano-Kurve, *z-ordering, N-trees*):


1
3
0
2

8.4 R-Baum

Der R-Baum wurde entwickelt zur Indexierung ausgedehnter Objekte im k-dimensionalen Raum. Der Index speichert für jedes Objekt seine MBB ***(minimal bounding box)***. Der MBB ist dann die Referenz auf das Objekt zugeordnet. Im zwei-dimensionalen Fall wäre das z.B.:



Der R-Baum eignet sich aber auch zur mehrdimensionalen Indexierung beliebiger Objekte, wenn man alle Index-Attribute in numerische Werte transformiert. Das sollte meistens möglich sein.

Ein R-Baum ist eine Hierarchie von verschachtelten k-dimensionalen Intervallen I^k (Schachteln, ***boxes***) sogenannten **Regionen Rx**.

Jeder Knoten ν eines R-Baums belegt maximal eine Seite (Transfer-Einheit Platte<—>Hauptspeicher).

Jeder innere Knoten enthält maximal p_{Knoten} Paare der Form:

Ri = (Region I_i^k, Adresse von Knoten ν_i
auf der nächst tieferen Ebene (= Unterbaum))
für $i = 0 \ldots q$ mit $q < p_{Knoten}$

Die Region I_i^k ist die MBB aller Boxen oder Objekte des Unterbaums ν_i. Die I_i^k können sich überlappen!

In der folgenden Darstellung nummerieren wir diese Paare mit R1, R2, R3 ... über alle Knoten durch.

Jeder Blattknoten enthält maximal p_{Blatt} Paare der Form:

(Box I_i^k = MBB eines Datenobjekts, Adressen eines oder mehrerer Datenobjekte)
für $i = 0 \ldots q$ mit $q < p_{Blatt}$

Die Wurzel steht oben und ist am Anfang ein Blatt, bis dieses in zwei Blätter gespalten wird und eine neue Wurzel als innerer Knoten entsteht mit mindestens zwei Region-Adresse-Paaren R1 und R2.

Suche: Wir kennen die k-dimensionale MBB des gesuchten Objekts. (Bei einer Punkt-Suche schrumpft diese auf einen Punkt zusammen. Wegen der schlechten Vergleichbarkeit von Realzahlen muss eventuell eine "Epsilon-MBB" definiert werden.)

Beginnend mit der Wurzel suchen wir die Regionen I_i^k, die die gesuchte MBB enthalten.

Gibt es mehr als eine Region I_i^k, müssen wir mehrere Suchpfade weiterverfolgen. Gibt es keine Region, die die MBB enthält, dann können wir die Suche abbrechen.

Jedenfalls führt uns das zu einem oder mehreren Knoten auf der nächst tieferen Ebene. Dort suchen wir wieder eine enthaltende (überlappende?) Region. Gibt es das Objekt im Index, dann finden wir schließlich in einem Blatt bei einer exakten Suche genau seine MBB oder finden mehrere Objekte, deren MBB mit unserer überlappen. Alle diese Objekte werden als Resultat zurückgegeben.

Gibt es das Objekt nicht im Index, dann kann die Suche u.U. auf jeder Stufe schon beendet sein.

Einfügen: Wir suchen zunächst das Objekt mit seiner MBB. Dabei kommen wir schließlich

1. entweder zu einem inneren Knoten, bei dem es keine Region I_i^k mehr gibt, in die die MBB unseres Objekts hineinpasst. Dann wird die Region I_i^k bestimmt, die am wenigsten vergrößert werden muss, um die MBB des einzufügenden Objekts aufzunehmen. Sie wird vergrößert, so dass die MBB hineinpasst.
   So verfährt man dann mit allen inneren Knoten auf den folgenden Stufen, bis man ein Blatt erreicht. Das Blatt wird wie folgt behandelt.

2. Oder man erreicht ein Blatt.

   (a) Entweder gibt es dort die MBB, aber mit einem anderen zugeordneten Objekt. Dann wird dieser MBB auch das neu einzufügende Objekt zugeordnet.

   (b) Oder die MBB gibt es nicht. Dann wird sie in das Blatt eingefügt mit der Adresse des ihr zugeordneten Datenobjekts. Ferner müssen bis zur Wurzel die jeweiligen I_i^k der darüber liegenden Knoten angepasst, d.h. vergrößert werden.
   Falls das Blatt voll war, wird das Blatt in zwei Blätter aufgespalten. Für jedes Teil-Blatt wird die MBB über alle seine I_i^k bestimmt. Im darüber liegenden inneren Knoten wird die I_i^k und Adresse ν_i des ursprünglichen Blatts durch die beiden neuen MBB und die zugehörigen Blatt-Adressen ersetzt.
   Dadurch bekommt der innere Knoten ein zusätzliches Element. Wenn der Knoten voll war, dann wird auch er gespalten, mit der Folge, dass der darüber liegende Knoten wieder ein weiteres Element bekommt. Das kann sich bis zur Wurzel fortsetzen, so dass schließlich eine neue Wurzel entsteht. So wächst der Baum nach oben.
   Die vernünftigste Spaltung der Region wäre wahrscheinlich die, bei der sich die Teile am wenigsten überlappen. Der entsprechende Algorithmus hätte aber Komplexität $O(2^n)$ und kommt daher nicht in Frage.
   Für die Spaltung einer Region in zwei Teil-Regionen hat sich offenbar folgende Heuristik mit linearer Komplexität $O(n)$ bewährt:
   Man wählt die beiden *extremsten* Einträge E_1 und E_2. Damit sind die Einträge gemeint, die in den Ecken des Raums liegen, oder eventuell auch die am weitesten voneinander entfernt sind.
   Sie werden den Gruppen G_1 und G_2 zugewiesen. Danach wird jeder noch nicht zugewiesene Eintrag derjenigen Gruppe zugewiesen, deren MBB am wenigsten vergrößert werden muss.
   Wenn E_1 gleich E_2 ist, dann wird der besser in der Ecke liegende Eintrag einer Gruppe zugewiesen. Alle anderen Einträge werden der anderen Gruppe zugewiesen.

Löschen: Der Baum kann wieder schrumpfen, wenn Objekte gelöscht werden und zu leere Knoten wieder rekombiniert werden. Dabei hat sich offenbar bewährt, zu leere Knoten ganz zu löschen und deren Elemente wieder einzufügen.

Außerdem müssen bei jeder Objekt-Löschung die Boxen bis zur Wurzel angepasst werden.

Bemerkung: Bei String-wertigen Dimensionen müssen die Strings irgendwie in numerische Werte umgewandelt werden.

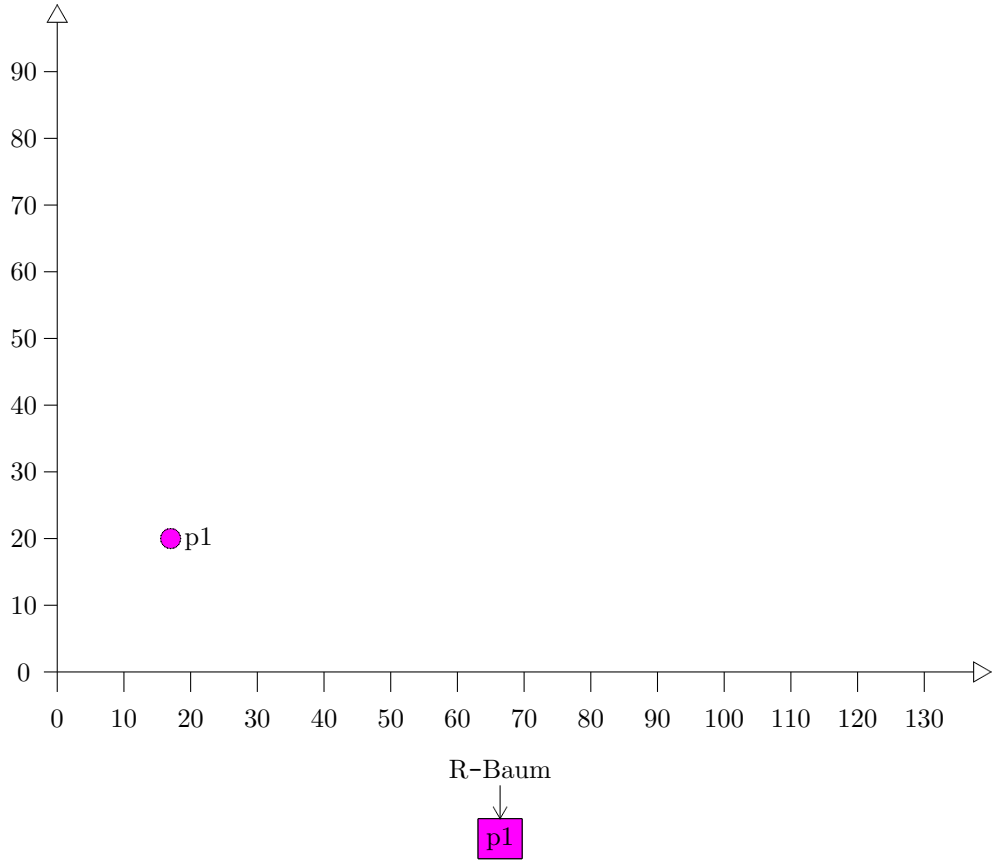
Zusammenfassend kann man sagen, dass ein R-Baum ein B^+-Baum ist, bei dem die Suchschlüssel K_i durch Regionen I_i^k ersetzt sind.

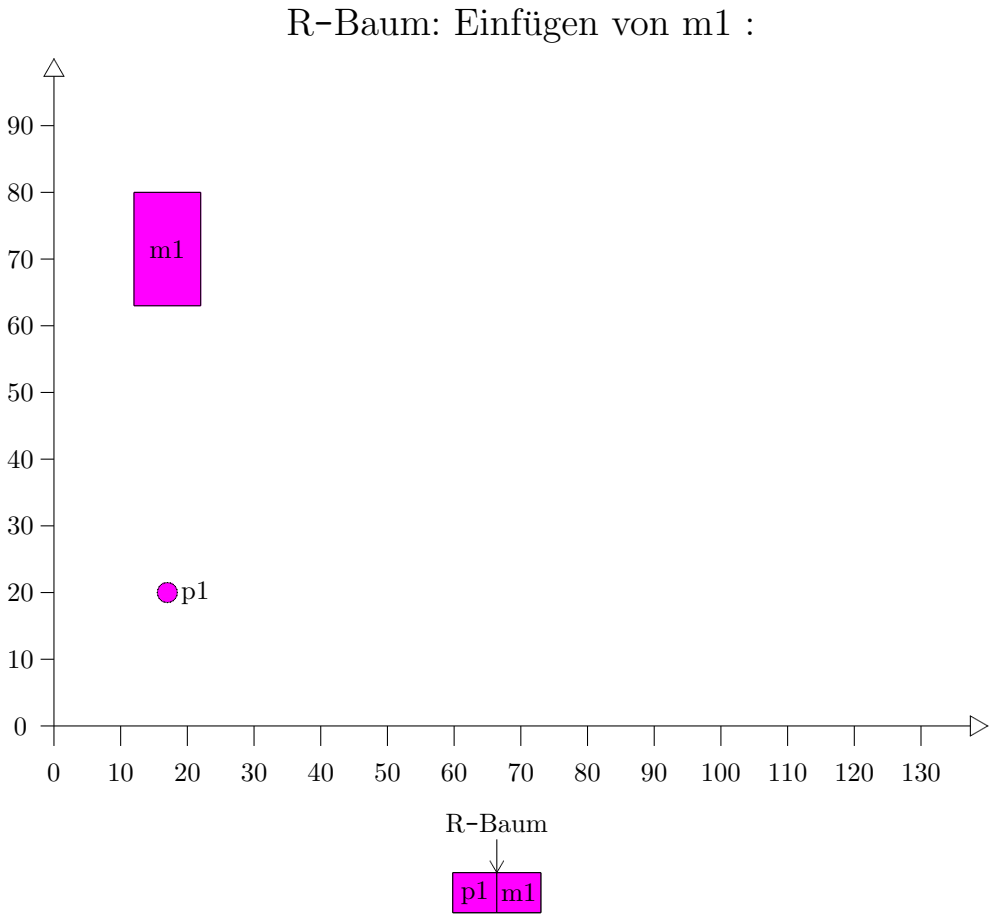
Beispiel: Wir füllen einen zweidimensionalen R-Baum nacheinander mit zweidimensionalen numerischen Datensätzen.

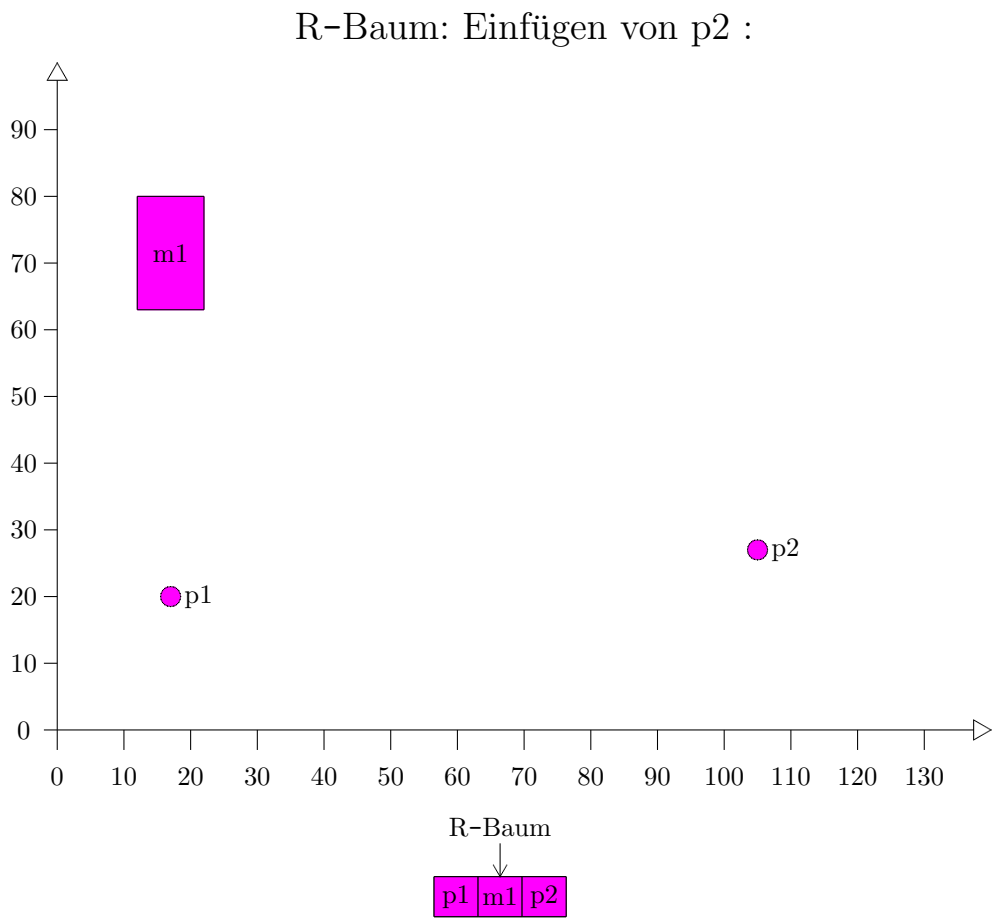
$k = 2$ (2 Dimensionen)
$p_{Knoten} = 3$ (Innere Knoten enthalten höchsten 3 Knoten-Referenzen.)
$p_{Blatt} = 3$ (Blätter enthalten höchsten 3 Tupel-Referenzen.)
Die Datensätze `p1 ... p10` repräsentieren punktförmige Objekte.
Die Datensätze `m1 ... m10` repräsentieren ausgedehnte Objekte.

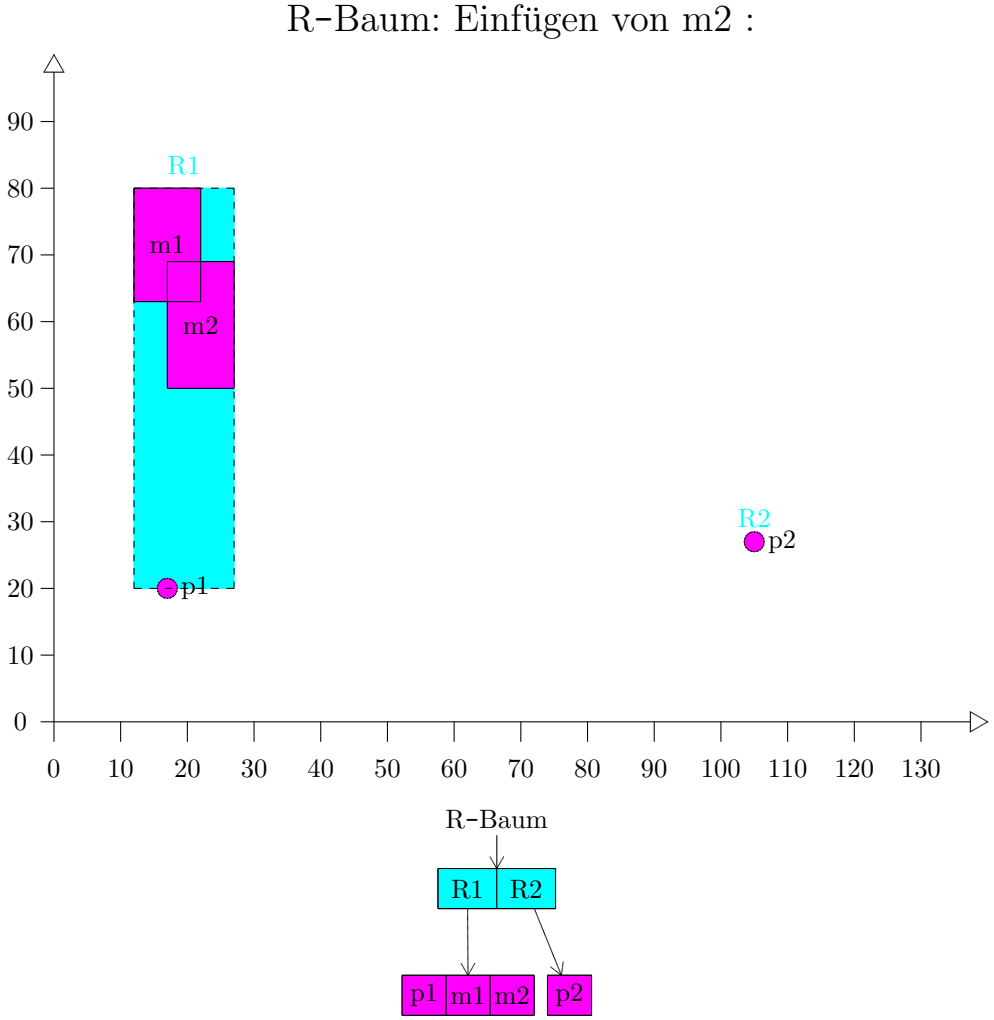
Reihenfolge des Einfügens der Datensätze: `p1, m1, p2, m2 ...`
(Die Beispieldaten sind ähnlich denen von Gaede und Günther [18]. Die Reihenfolge der Einfügung könnte unterschiedlich sein.)

R-Baum: Einfügen von p1 :

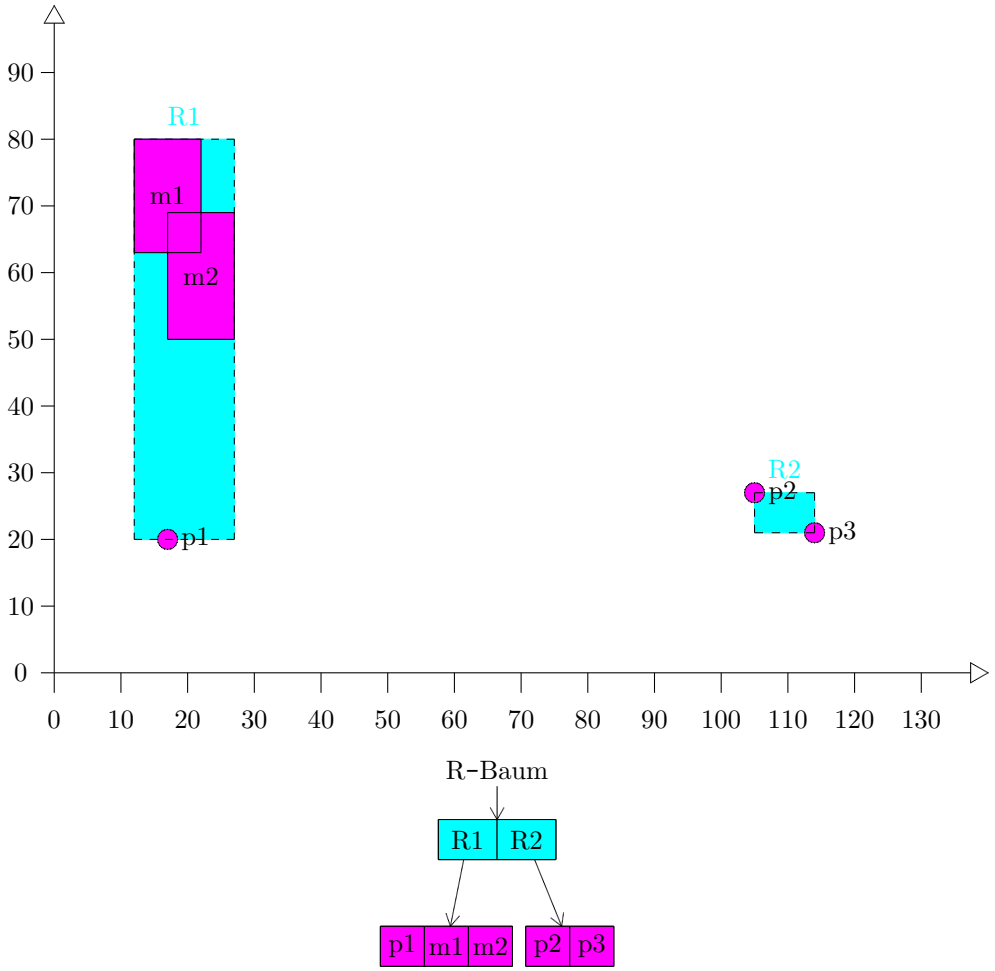


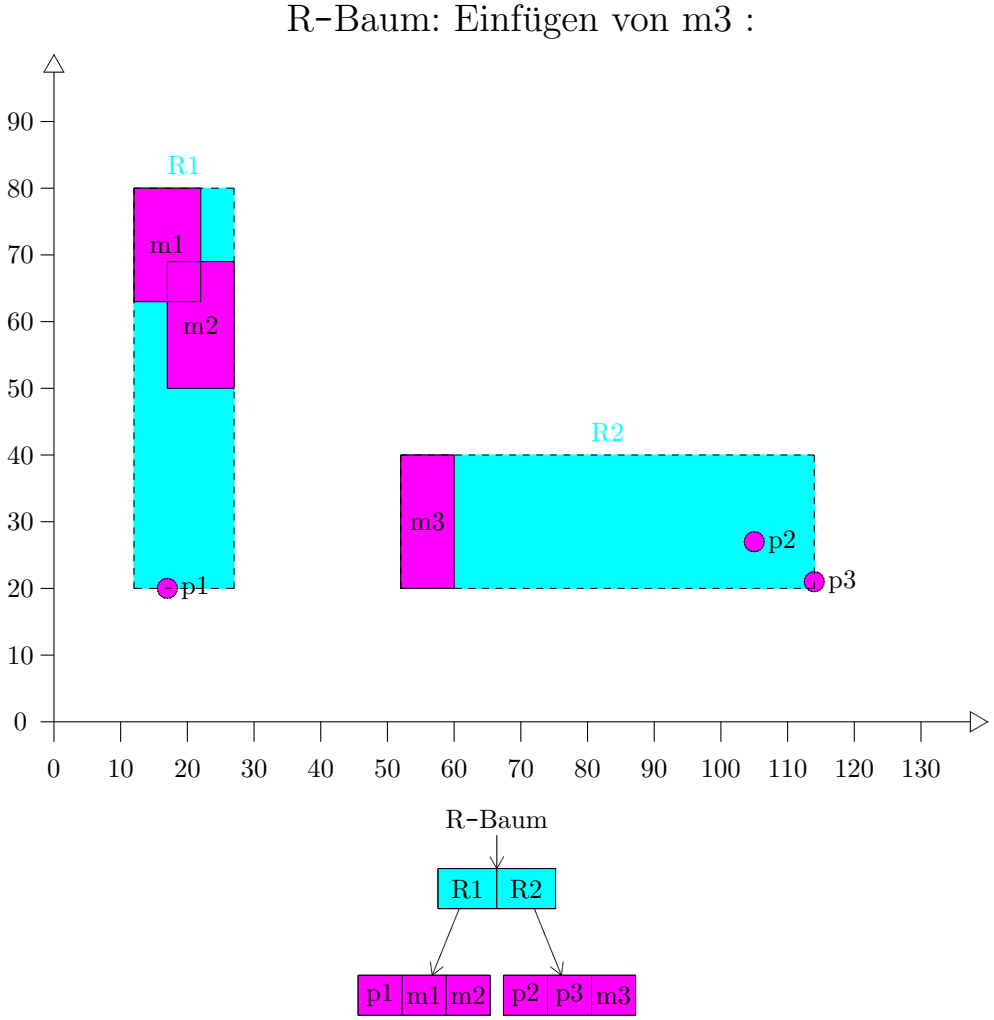

R-Baum: Einfügen von m1 :
90
80
70
60
50
40
30
20
10
0
m1
p1
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
p1
m1


R-Baum: Einfügen von p2 :
90
80
70
60
50
40
30
20
10
0
m1
p1
p2
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
p1
m1
p2

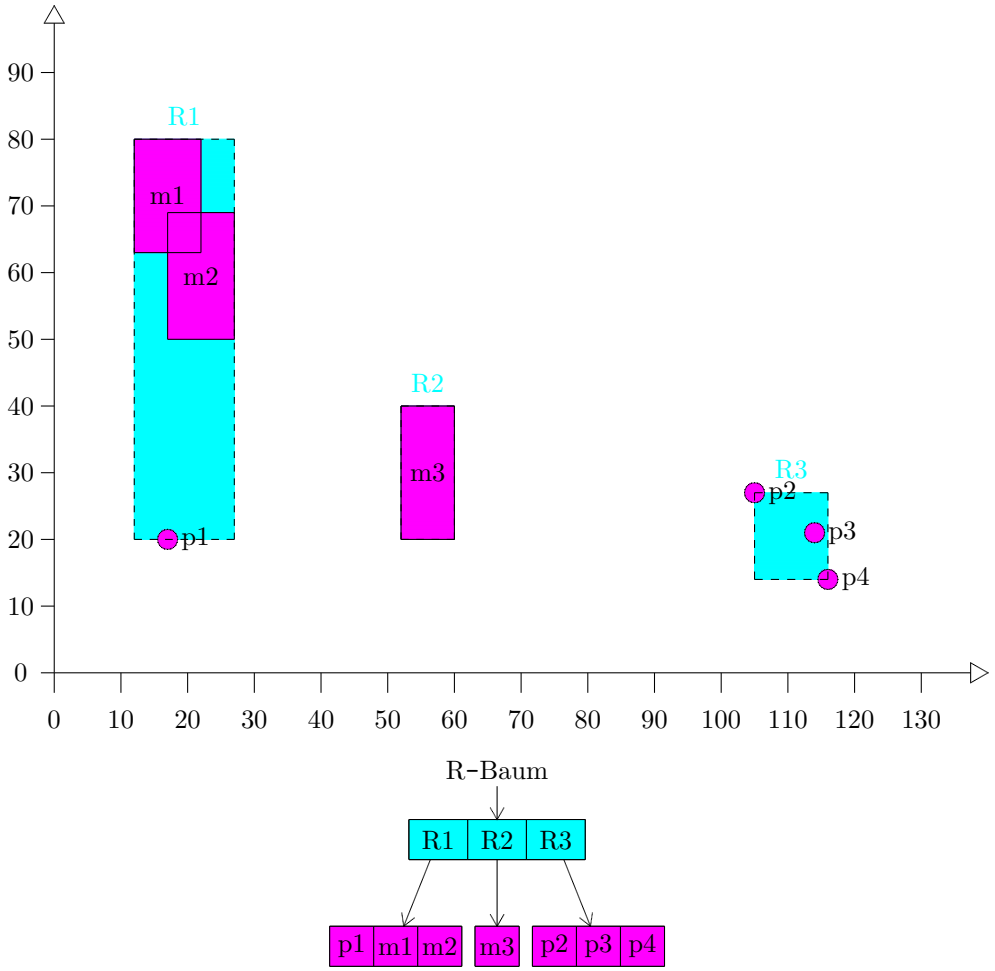

R-Baum: Einfügen von m2 :
R1
m1
m2
p1
R2
p2
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R1
R2
p1
m1
m2
p2

R-Baum: Einfügen von p3 :

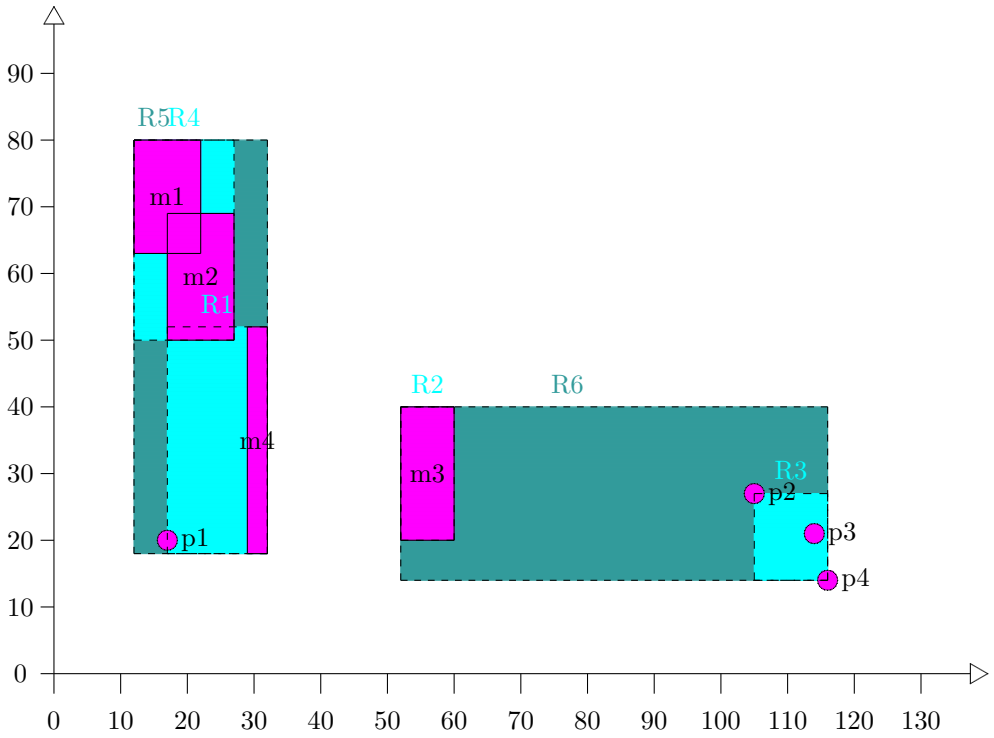


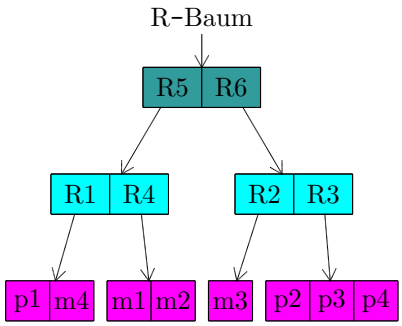

R-Baum: Einfügen von m3 :
R1
m1
m2
R2
m3
p1
p2
p3
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R1
R2
p1
m1
m2
p2
p3
m3

R-Baum: Einfügen von p4 :

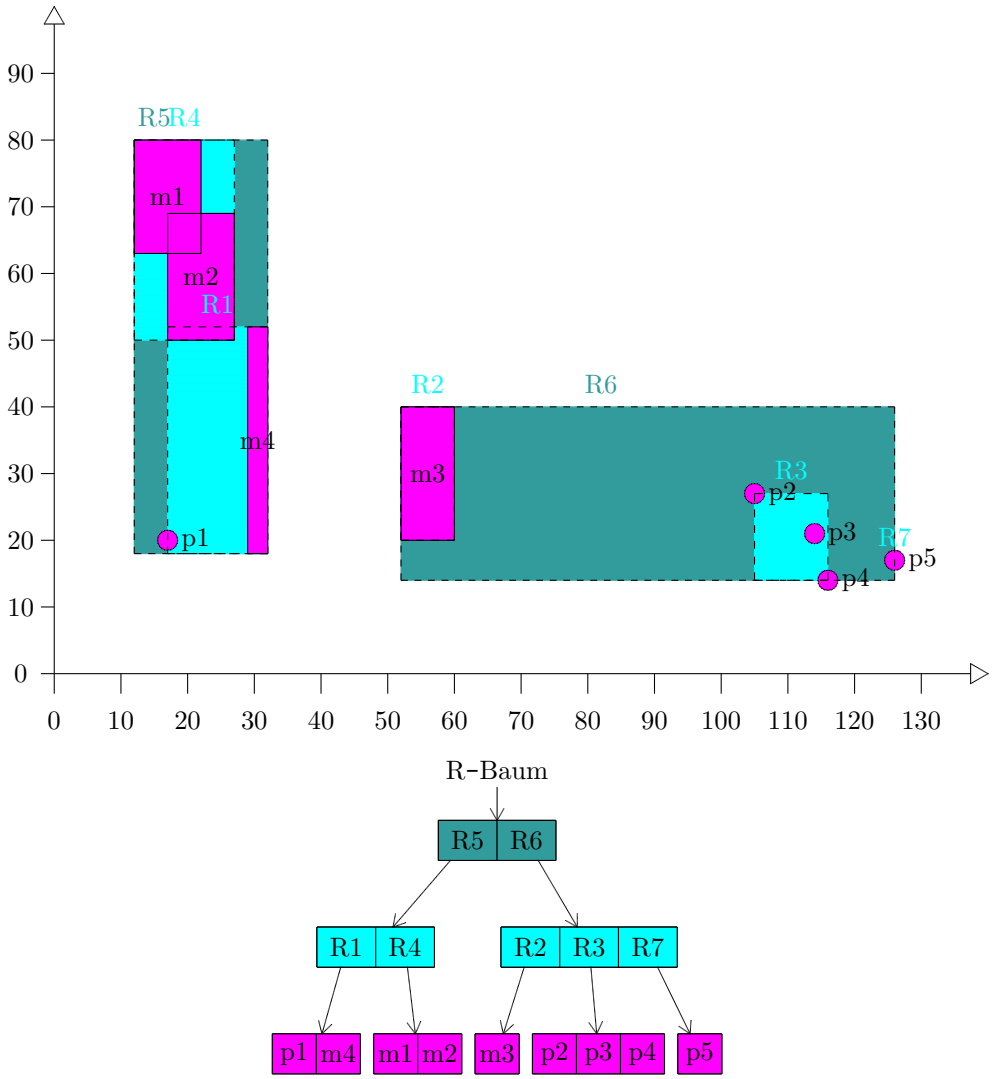


R-Baum: Einfügen von m4 :

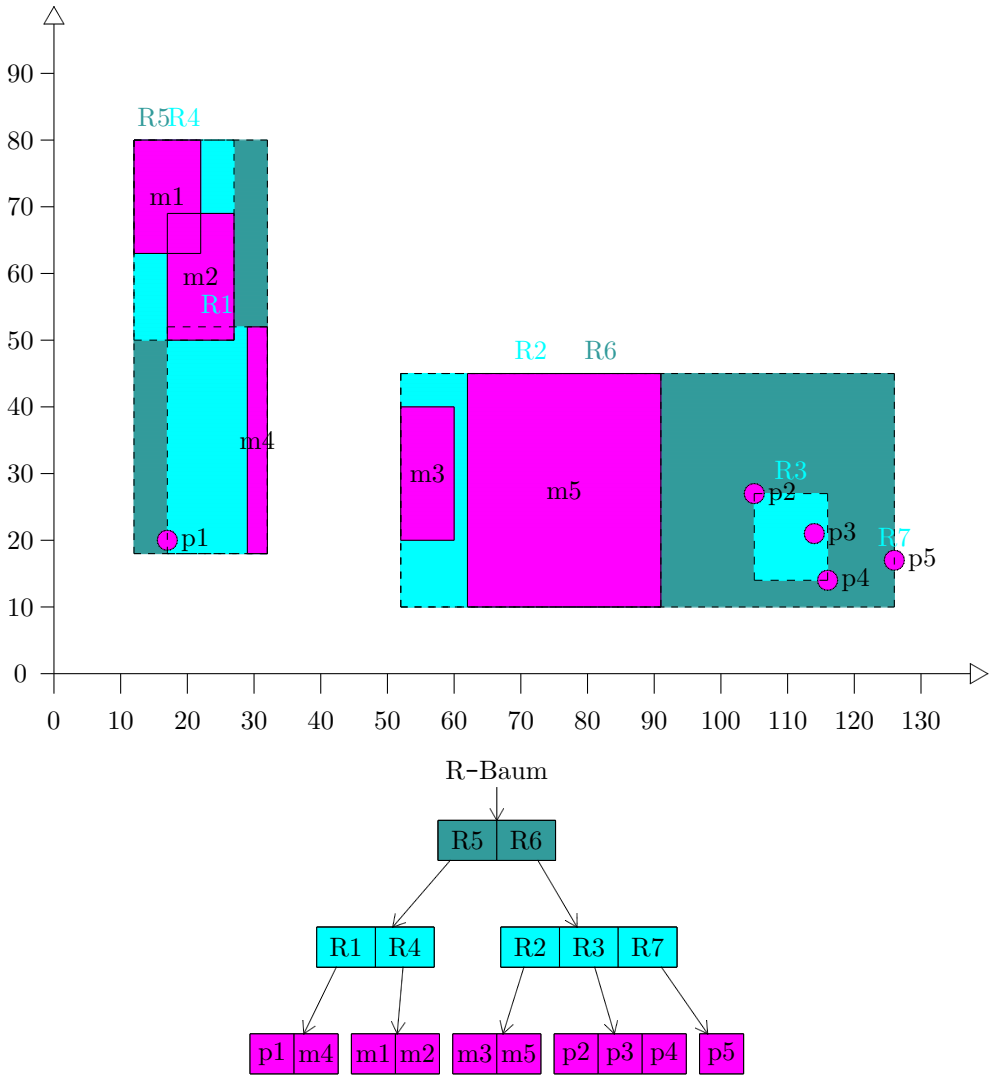




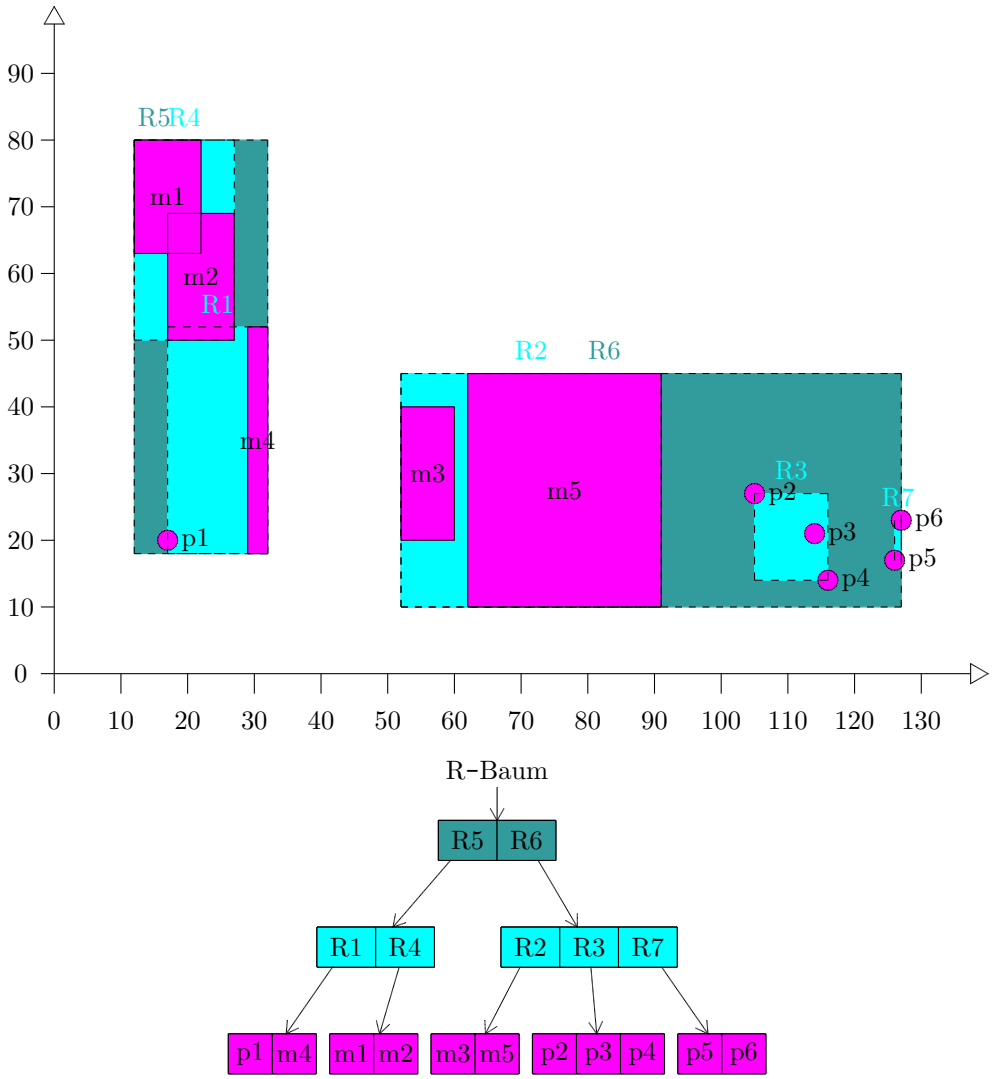
R-Baum: Einfügen von p5 :

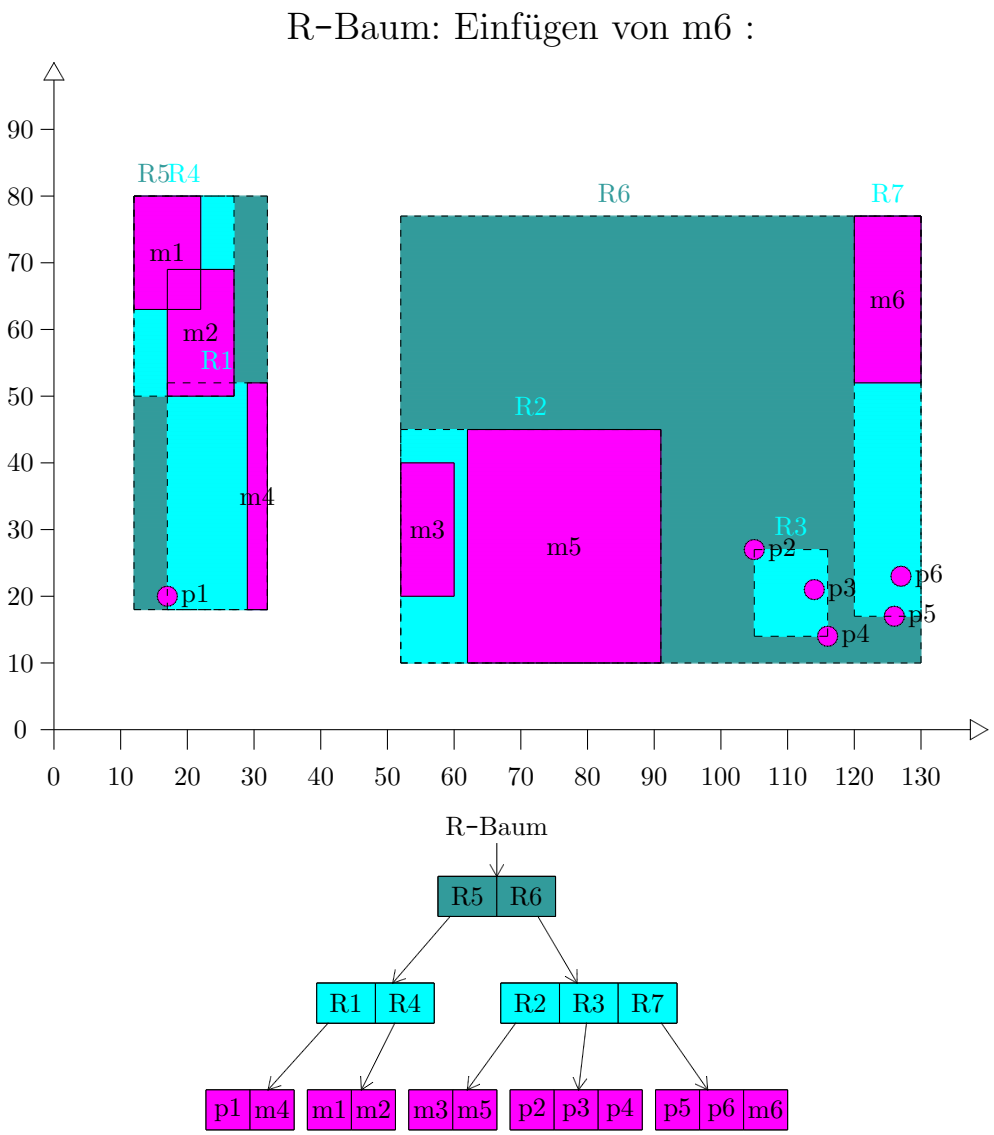


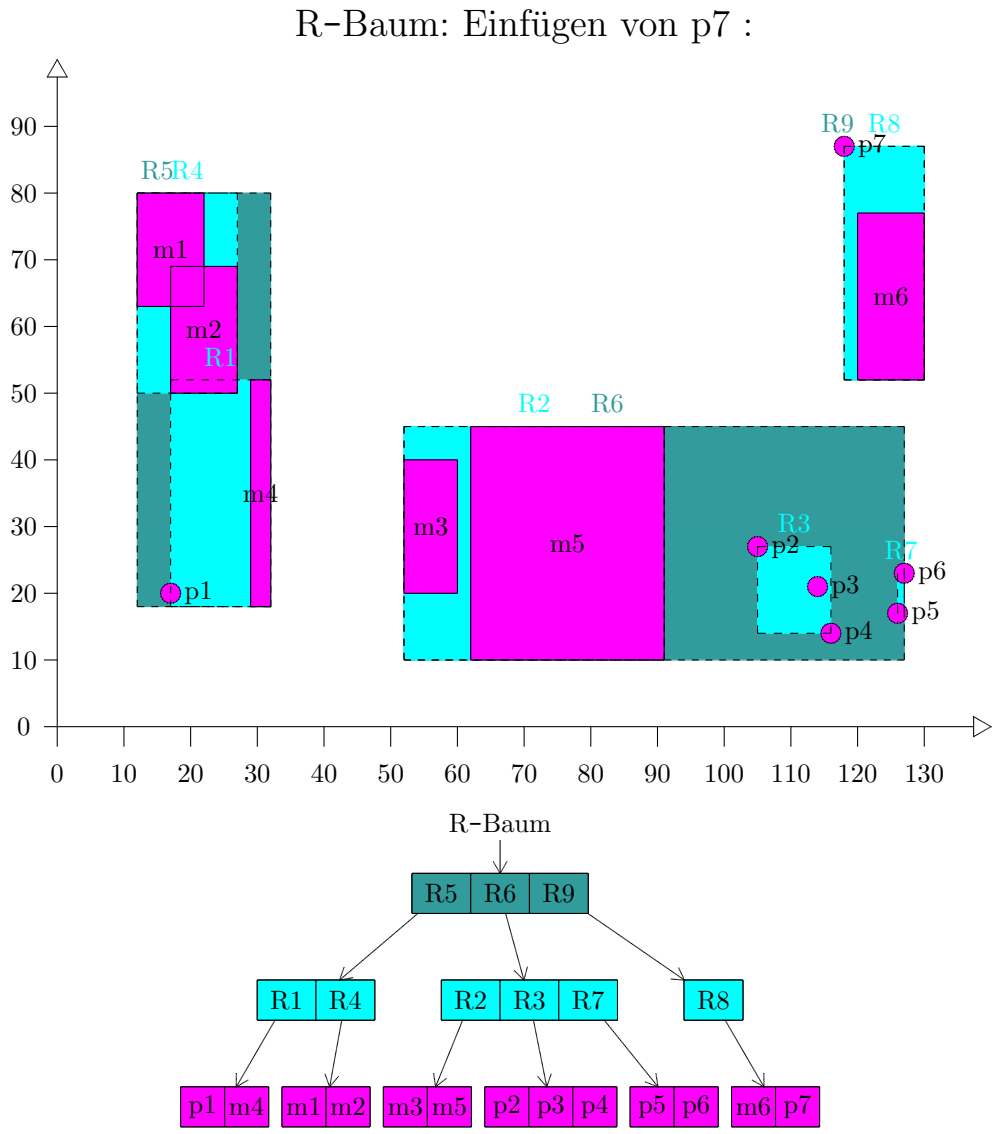
R-Baum: Einfügen von m5 :

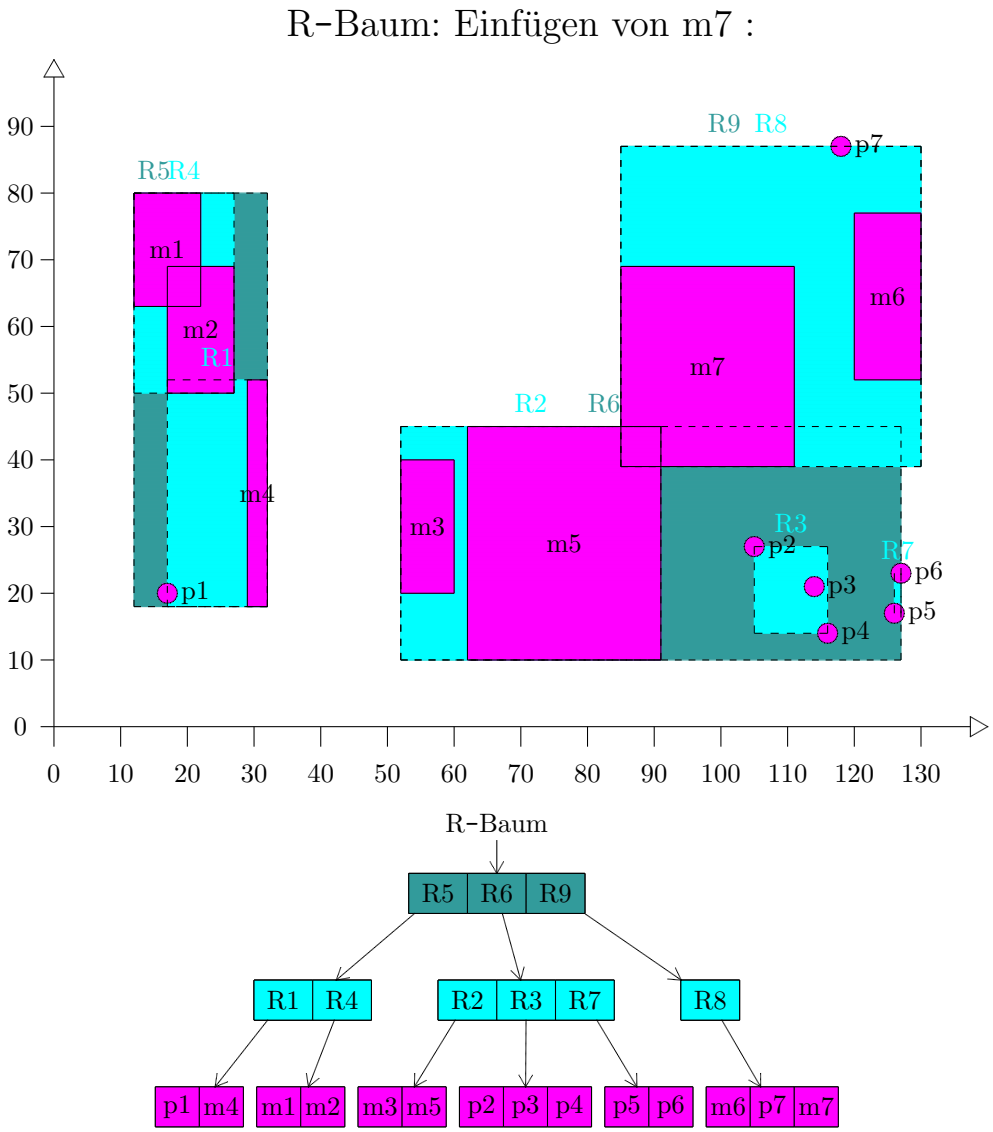


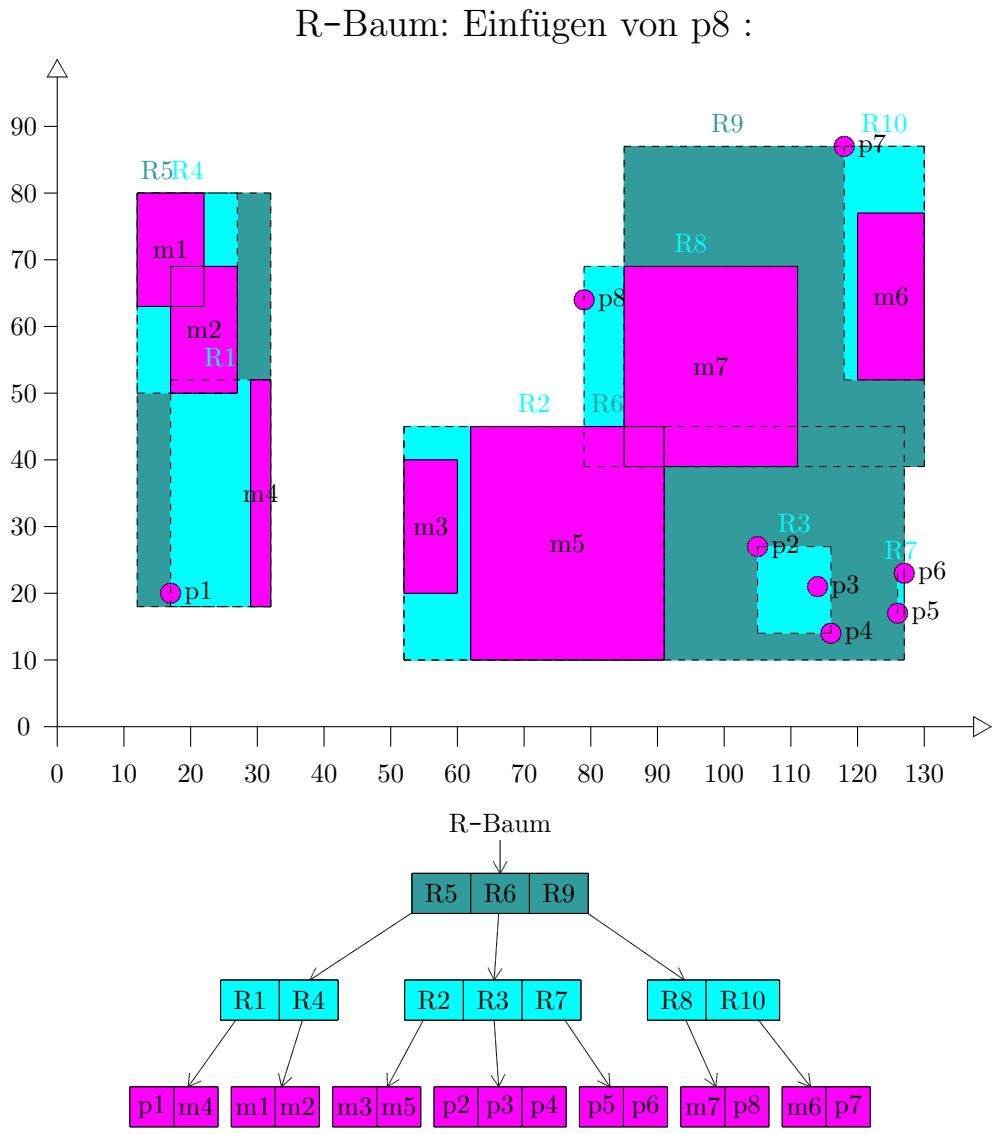
R-Baum: Einfügen von p6 :

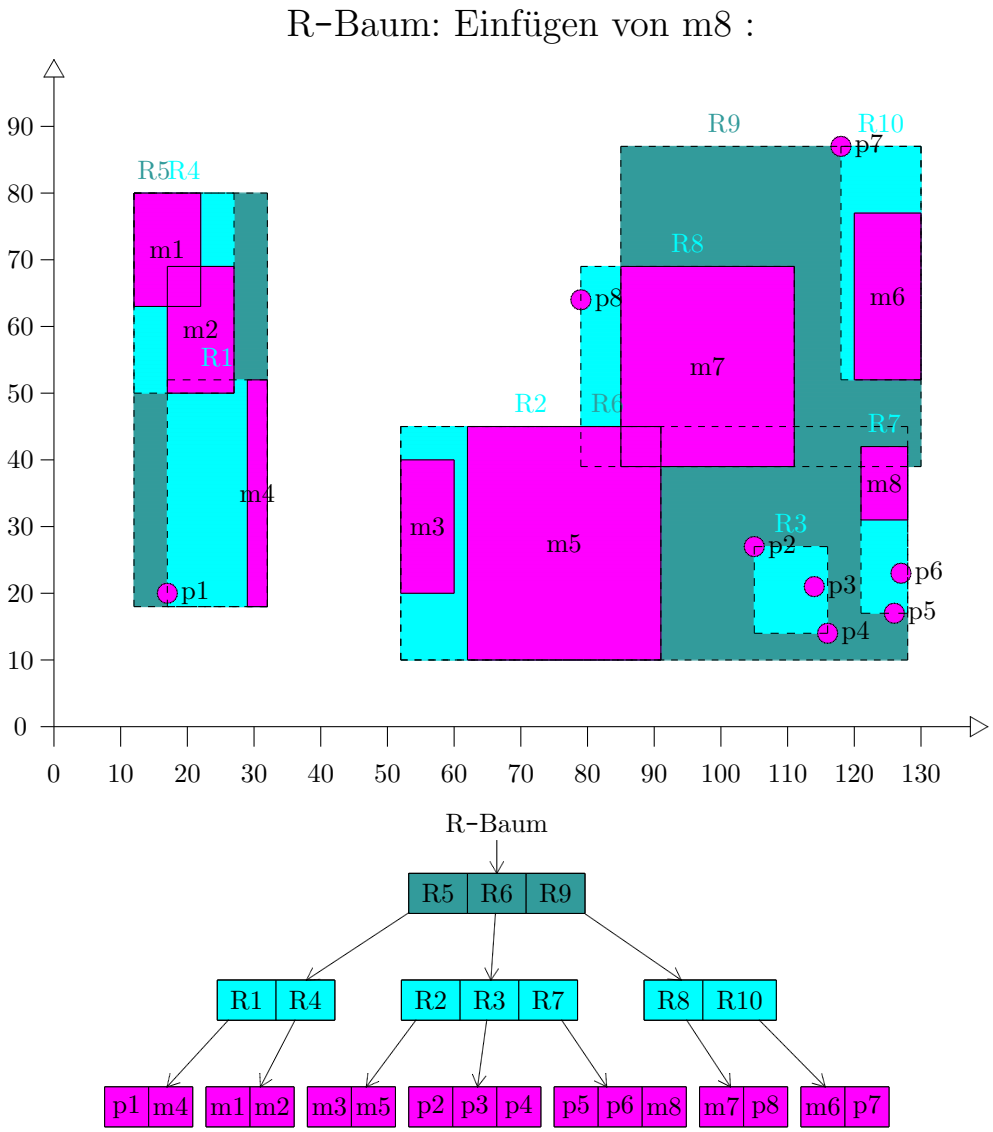



R-Baum: Einfügen von m6 :
R5R4
R6
R7
m1
m2
R1
m4
m6
R2
m3
m5
R3
p2
p3
p4
p6
p5
p1
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R5
R6
R1
R4
R2
R3
R7
p1
m4
m1
m2
m3
m5
p2
p3
p4
p5
p6
m6

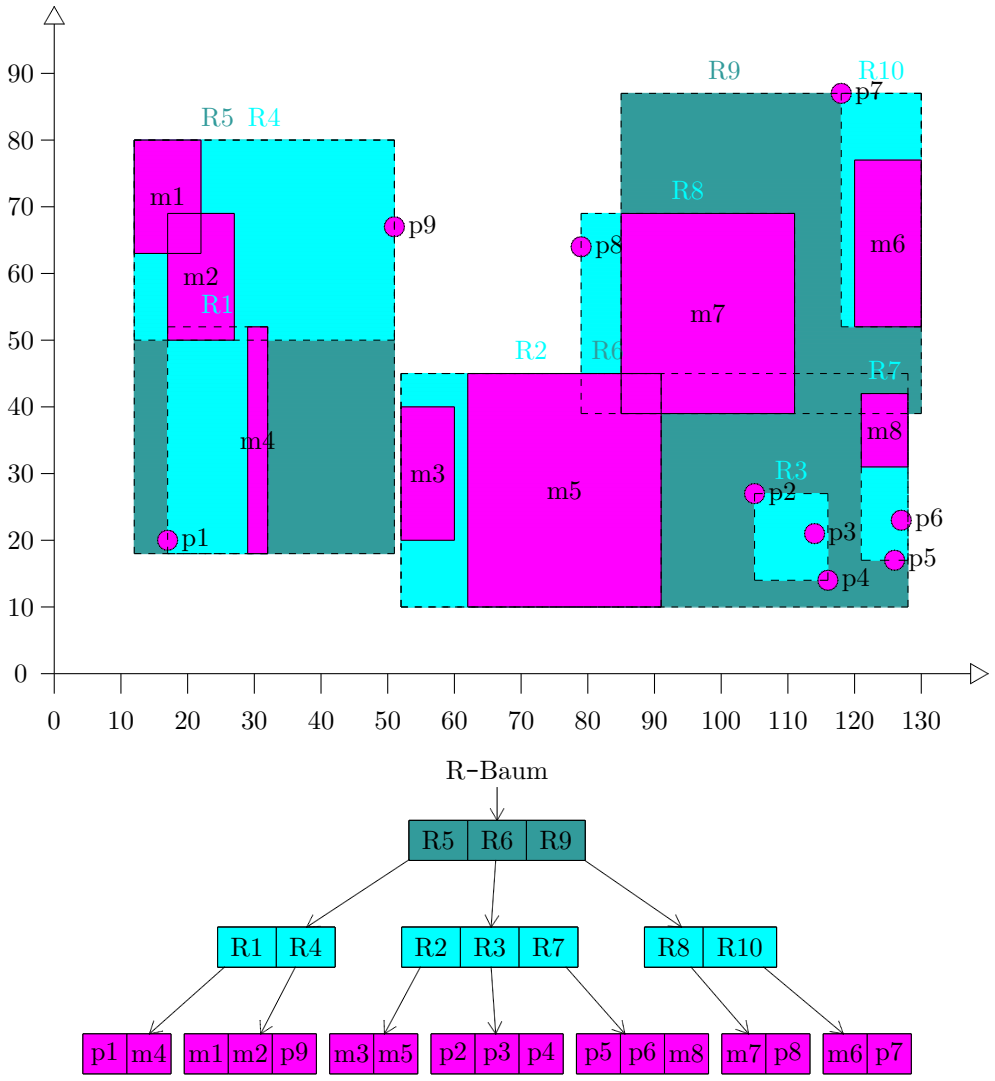

R-Baum: Einfügen von p7 :
R5
R4
R9
R8
p7
m1
m2
R1
m6
R2
R6
m3
m5
R3
p2
R7
p6
p3
p5
p4
m4
p1
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R5
R6
R9
R1
R4
R2
R3
R7
R8
p1
m4
m1
m2
m3
m5
p2
p3
p4
p5
p6
m6
p7

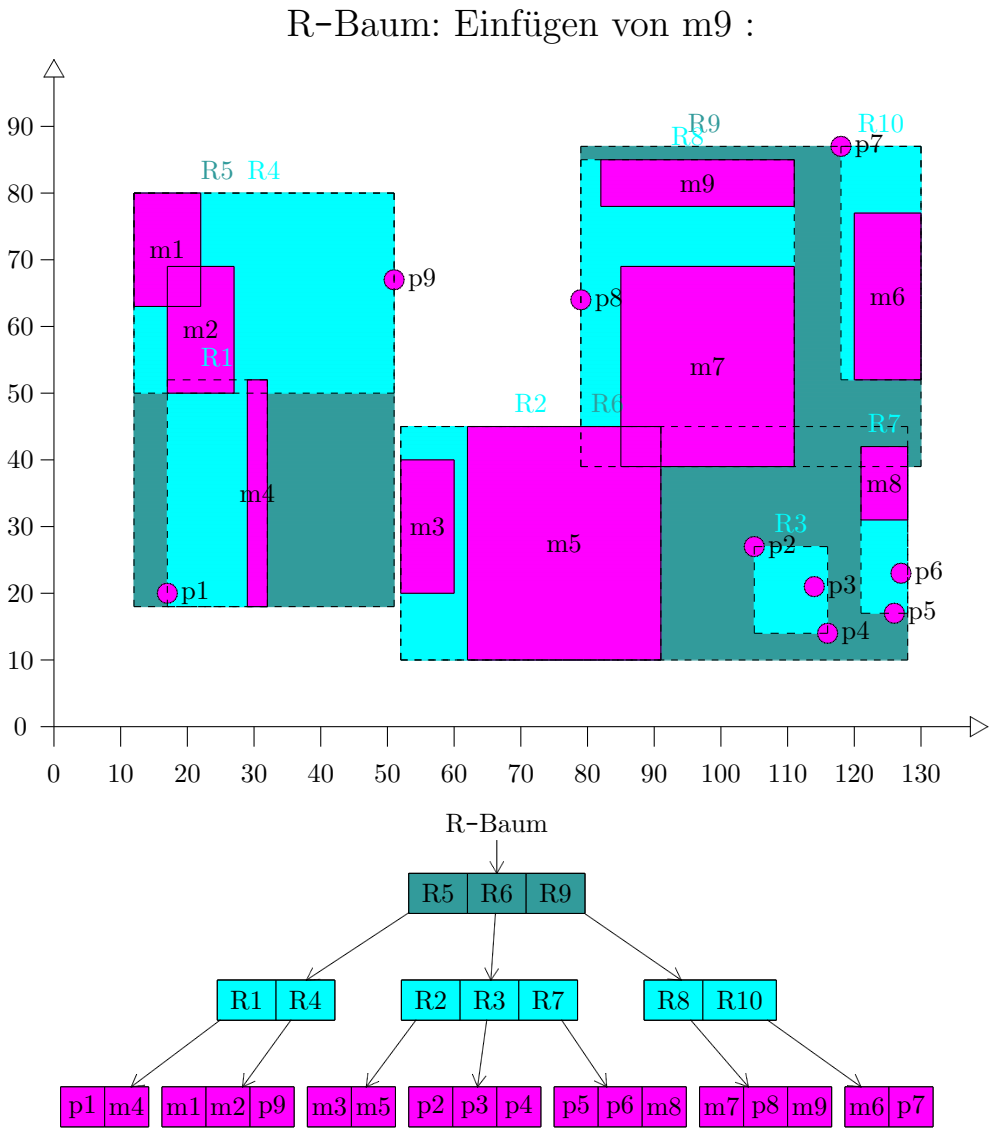

R-Baum: Einfügen von m7 :
R5R4
R9 R8
R1
R2
R6
R3
R7
m1
m2
m3
m4
m5
m6
m7
p1
p2
p3
p4
p5
p6
p7
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R5 R6 R9
R1 R4
R2 R3 R7
R8
p1 m4
m1 m2
m3 m5
p2 p3 p4
p5 p6
m6 p7 m7


R-Baum: Einfügen von p8 :
R5
R4
R9
R10
R8
R1
R2
R6
R3
R7
m1
m2
m4
m3
m5
m7
m6
p1
p2
p3
p4
p5
p6
p7
p8
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R5 R6 R9
R1 R4
R2 R3 R7
R8 R10
p1 m4
m1 m2
m3 m5
p2 p3 p4
p5 p6
m7 p8
m6 p7

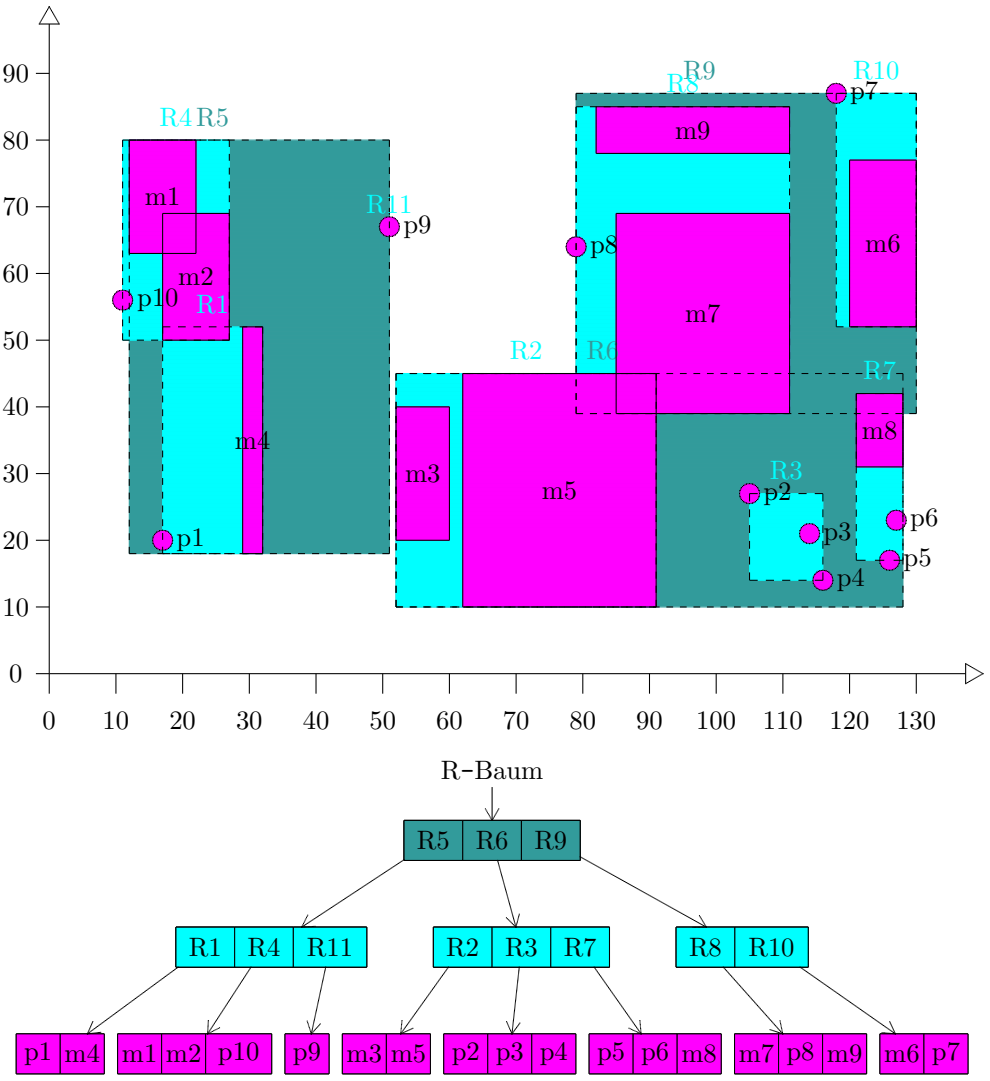

R-Baum: Einfügen von m8 :
R5
R4
R9
R10
R8
R1
R2
R6
R7
R3
m1
m2
m3
m4
m5
m6
m7
m8
p1
p2
p3
p4
p5
p6
p7
p8
0
10
20
30
40
50
60
70
80
90
100
110
120
130
R-Baum
R5 R6 R9
R1 R4
R2 R3 R7
R8 R10
p1 m4
m1 m2
m3 m5
p2 p3 p4
p5 p6 m8
m7 p8
m6 p7

R-Baum: Einfügen von p9 :

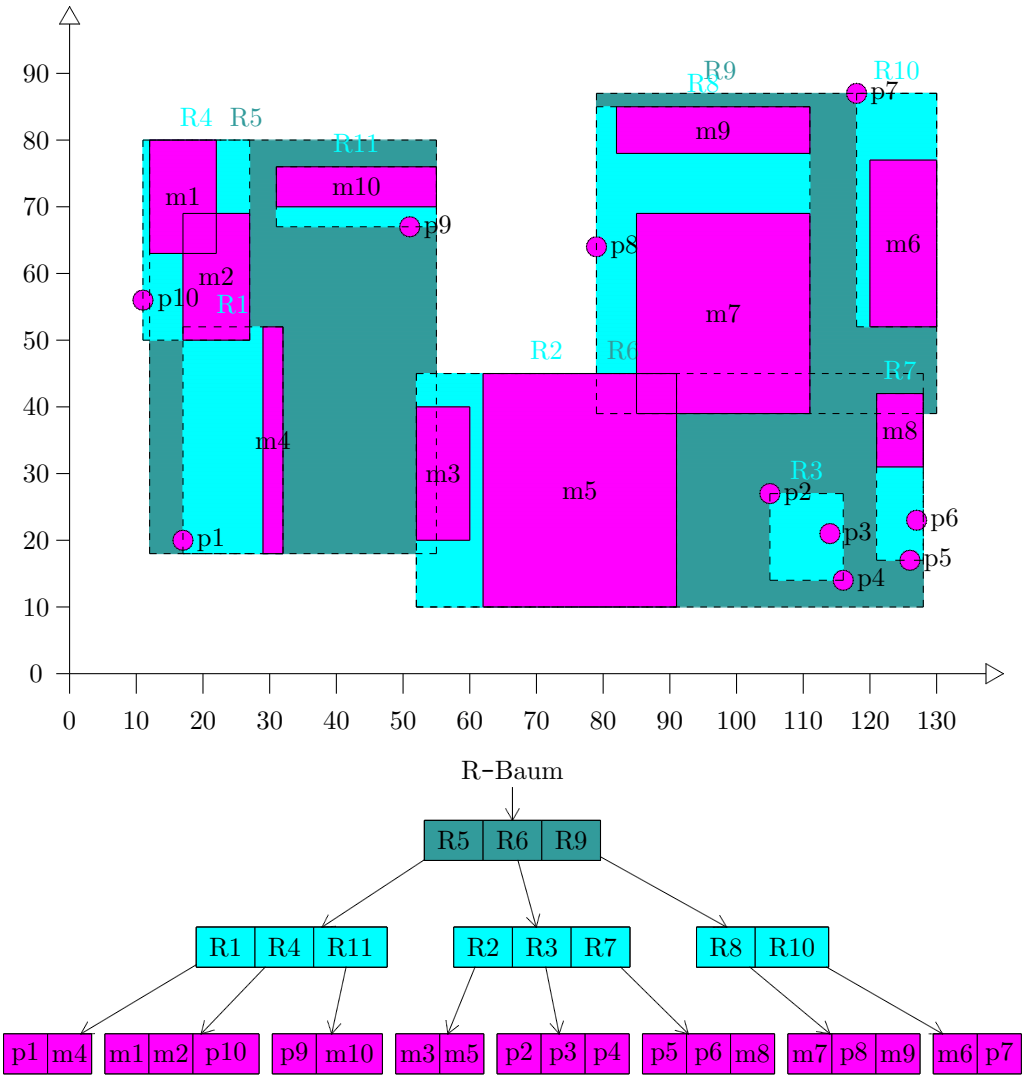



R-Baum: Einfügen von m9 :
R5 R4
R9
R8
R10
R1
R2
R6
R7
R3
m1
m2
m3
m4
m5
m6
m7
m8
m9
p1
p2
p3
p4
p5
p6
p7
p8
p9
0 10 20 30 40 50 60 70 80 90
0 10 20 30 40 50 60 70 80 90 100 110 120 130
R-Baum
R5 R6 R9
R1 R4
R2 R3 R7
R8 R10
p1 m4
m1 m2 p9
m3 m5
p2 p3 p4
p5 p6 m8
m7 p8 m9
m6 p7

R-Baum: Einfügen von p10 :



R-Baum: Einfügen von m10 :



Kapitel 9

Relationale Datenbank (RDB)

In diesem Kapitel werden wir unter Verzicht auf formale Genauigkeit die wichtigsten Aspekte relationaler DBS oder kurz *relationaler Systeme* behandeln.

9.1 Das relationale Modell

Ein **relationales DBS** oder **relationales System** ist ein DBS, das auf dem **relationalen Modell** [11] beruht. Dieses Modell ist eine Vorschrift, die angibt,

- wie Daten repräsentiert werden sollen, nämlich durch Tabellen (Aspekt der Daten**struktur**). Die *Benutzersicht* auf die Daten sind **Tabellen** und *nur* Tabellen.
- welche Regeln die Daten erfüllen müssen (Aspekt der Daten**integrität**).
- wie Daten manipuliert werden sollen (Aspekt der Daten**manipulation**). Die dem Benutzer zur Verfügung stehenden Operatoren erlauben ihm, neue Tabellen aus alten Tabellen zu erzeugen. Dazu gehören mindestens die Operatoren `select` (oder auch `restrict`), `project` und `join`.

Bemerkung zur Terminologie: Wir verwenden hier Begriffe aus dem **relationalen Datenmodell**, die trotz ihrer Ähnlichkeit mit denen der Anfragesprache SQL nicht damit zu verwechseln sind.

Es folgt eine grobe Definition der Operatoren, wobei wir zwei Notationen – eine textuelle und eine symbolische – vorstellen:

`select(restrict)` (σ) **:** extrahiert spezifische Zeilen aus einer Tabelle und generiert daraus eine neue Tabelle.

`project` (π) **:** extrahiert spezifische Spalten aus einer Tabelle und generiert daraus eine neue Tabelle.

`join` ($\bowtie$) **:** (Verbund) verbindet zwei Tabellen auf der Basis von gleichen Werten in gemeinsamen Spalten. Auf eine allgemeinere Definition gehen wir im Kapitel "Relationale Algebra" ein.

Beispiel:

Tabelle ABTEILUNG :

ABTNR	ABTNAME	BUDGET
A1	Marketing	10M
A2	Entwicklung	12M
A3	Forschung	5M

Tabelle MITARBEITER :

MNR	MNAME	ABTNR	GEHALT
M1	Ibsen	A1	80K
M2	Rostand	A1	84K
M3	Wilde	A2	60K
M4	Canetti	A2	70K

Operation: `select` ABTEILUNG `where` BUDGET `>` 6M
(symbolisch: $\sigma_{\texttt{[BUDGET>6M]}}\,\texttt{ABTEILUNG}$)
Resultat:

ABTNR	ABTNAME	BUDGET
A1	Marketing	10M
A2	Entwicklung	12M

Operation: `project` ABTEILUNG `over` ABTNR, BUDGET
(symbolisch: $\pi_{\texttt{[ABTNR,BUDGET]}}\,\texttt{ABTEILUNG}$)
Resultat:

ABTNR	BUDGET
A1	10M
A2	12M
A3	5M

Operation: `join` ABTEILUNG `and` MITARBEITER `over` ABTNR
(symbolisch: `ABTEILUNG` $\bowtie_{\texttt{[ABTNR]}}$ `MITARBEITER`)
Resultat:

ABTNR	ABTNAME	BUDGET	MNR	MNAME	GEHALT
A1	Marketing	10M	M1	Ibsen	80K
A1	Marketing	10M	M2	Rostand	84K
A2	Entwicklung	12M	M3	Wilde	60K
A2	Entwicklung	12M	M4	Canetti	70K

Das `join` bedarf wohl einer Erklärung: Beide Tabellen haben in der Tat eine gemeinsame Spalte ABTNR. Sie können also auf der Basis gleicher Werte in dieser Spalte verbunden werden. D.h. eine Zeile in der Tabelle ABTEILUNG wird nur dann mit einer Zeile aus der Tabelle MITARBEITER erweitert, wenn die beiden Zeilen den gleichen Wert in der Spalte ABTNR haben. Alle möglichen solcher Kombinationen ergeben die resultierende Tabelle. Im Resultat erscheint die gemeinsame Spalte ABTNR nur einmal. Da es keinen Mitarbeiter in der Abteilung A3 gibt, erscheint im Resultat keine Zeile mit A3.

Bemerkungen:

- Die Resultate der drei Operationen sind in der Tat wieder Tabellen. Die relationalen Operationen haben also die Eigenschaft der **Abgeschlossenheit** ***(closure)***. Diese Eigenschaft ist sehr wichtig. Denn die Ausgabe einer Operation kann Eingabe einer weiteren Operation werden, da Ein- und Ausgabeobjekte Objekte gleichen Typs, nämlich Tabellen sind. Ob das System bei verketteten Operationen wirklich Tabellen als Zwischenergebnisse erzeugt oder nicht, spielt für uns hier keine Rolle. Jedenfalls werden für die hier besprochene konzeptuelle Sicht Tabellen erzeugt.

- Die relationalen Operationen sind **Mengen**operationen. Sie liefern immer ganze Tabellen, nicht einzelne Zeilen. Vor-relationale Systeme sind im allgemeinen Zeilen- oder Datensatzorientiert.

- Tabellen sind die **logische** ***(logical)***, besser **konzeptuelle** ***(conceptual)*** **Struktur**, nicht unbedingt die physikalische Struktur in relationalen Systemen. Auf physikalischem Niveau kann das System jede beliebige Speicherstruktur – sequentielle Dateien, Indexierung, Hashing, Zeigerketten, Kompression – verwenden, solange sich diese Strukturen auf der konzeptuellen Ebene in Tabellen abbilden lassen. Tabellen repräsentieren eine *Abstraktion* der physikalischen Speicherstruktur.

  Der Begriff "logische Struktur" betrifft die externe und die konzeptuelle Ebene des ANSI/SPARC-Modells. In relationalen Systemen sind die konzeptuelle und meistens auch die externen Ebenen relational, die interne Ebene kann beliebig sein.

- Der ganze Informationsgehalt einer relationalen DB wird nur in einer Art dargestellt, nämlich explizit als **Datenwerte**. Insbesondere gibt es keine Zeiger, die auf andere Tabellen verweisen. Die Tatsache, dass der Mitarbeiter M1 in der Abteilung A1 arbeitet, wird dadurch repräsentiert, dass in der Spalte ABTNR der Mitarbeitertabelle der Datenwert A1 in der Zeile des Mitarbeiters M1 erscheint.

- Alle Datenwerte sind **atomar** ***(atomic, scalar)***. Jede Zelle einer Tabelle hat genau einen Datenwert und niemals eine **Gruppe von Werten** ***(repeating group)***. Betrachten wir z.B. die Projektion der Tabelle MITARBEITER auf ABTNR und MNR:
  Operation: `project` MITARBEITER `over` ABTNR, MNR
  Das Resultat ist

ABTNR	MNR
A1	M1
A1	M2
A2	M3
A2	M4

  und nicht etwa

ABTNR	MNR
A1	M1, M2
A2	M3, M4

- Integritätsregeln: Wahrscheinlich wird jede DB irgendwelche spezifischen Integritätsregeln zu beachten haben, etwa der Art, dass in der Spalte GEHALT nur positive Beträge kleiner 200K stehen dürfen. Es gibt aber Regeln, die jede DB einhalten muss, wenn sie als relational eingestuft werden will. Diese sind:

  – Hat eine Tabelle einen **Primärschlüssel** ***(primary key)***, dann müssen die Datenwerte der entsprechenden Spalte **eindeutig** ***(unique)*** sein, d.h. jeder Datenwert darf

nur genau einmal vorkommen. In der Tabelle ABTEILUNG ist die Spalte ABTNR der Primärschlüssel, in der Tabelle MITARBEITER ist die Spalte MNR der Primärschlüssel. Primärschlüsselnamen werden unterstrichen.

– Hat eine Tabelle einen **Fremdschlüssel** ***(foreign key)***, d.h. gibt es eine Spalte, die als Datenwerte Primärschlüsselwerte einer Tabelle enthält, dann dürfen dort nur Datenwerte stehen, die auch im Primärschlüssel einer anderen oder derselben Tabelle stehen. In der Tabelle MITARBEITER ist die Spalte ABTNR ein Fremdschlüssel und es wird gefordert, dass jeder Mitarbeiter in einer existierenden Abteilung ist. Ein Wert A5 wäre falsch, da es die Abteilung A5 nicht gibt. Fremdschlüsselnamen werden überstrichen.

- Terminologie: **Relation** ist der mathematische Begriff für **Tabelle**. Relationale Systeme basieren auf dem relationalen Datenmodell, das eine abstrakte Theorie ist, die auf Mengenlehre und Prädikatenlogik aufbaut. Dort wird der Begriff Relation verwendet.

  Die Prinzipien des relationalen Modells hat der Mathematiker E. F. Codd 1969-70 bei IBM aufgestellt [11].

  Das relationale Modell verwendet nicht den Begriff **Datensatz** ***(record)***, sondern den Begriff **Tupel** ***(tuple)***.

  Wir verwenden hier die Begriffe **Tabelle, Zeile, Spalte** synonym zu den Theoriebegriffen **Relation, Tupel, Attribut**.

- Das relationale Modell ist eine Theorie. Ein DBS muss nicht unbedingt alle Aspekte der Theorie abdecken. Es gibt auf dem Markt kein DBS, das das relationale Modell vollständig abdeckt.

9.2 Optimierung

Die relationalen Operationen sind Mengenoperationen. Daher werden Sprachen wie SQL, die diese Operationen unterstützen, als nicht-prozedural eingestuft. Der Benutzer gibt eher das "Was", nicht das "Wie" an. Der Benutzer sagt, welche Tabelle er haben will und überlässt dem System, einen geeigneten Algorithmus zur Erstellung der Tabelle zu finden ***(automatic navigation system)***. Die Begriffe "nicht-prozedural" und "prozedural" dürfen nicht absolut gesehen werden. Eine bessere Charakterisierung von Sprachen wie SQL ist, dass sie auf einem höheren Abstraktionsniveau als Java, C oder COBOL anzusiedeln sind.

Verantwortlich für das automatische Navigieren ist der **Optimierer** ***(optimizer)***, eine DBMS-Komponente. Er muss jede Anfrage möglichst effizient implementieren. Der Optimierer kümmert sich also um das "Wie".

9.3 Katalog

Wie schon erläutert, muss ein DBMS eine Katalog– oder Lexikonfunktion zur Verfügung stellen. Der Katalog enthält Informationen (**Deskriptoren**, ***descriptor)*** über die Schemata auf jeder Ebene, Integritätsregeln, Zugriffsrechte, Tabellen, Indexe, Benutzer. Der Optimierer benötigt für seine Aufgabe diesen Katalog.

In relationalen Systemen besteht der **Katalog** ***(catalog)*** auch aus Tabellen, sogenannten Systemtabellen, die ein berechtigter Benutzer wie andere Tabellen benutzen kann. Z.B. gibt es dort die Tabellen TABLES und COLUMNS, die die Tabellen und Spalten des Systems beschreiben.

Tabelle TABLES :

TABNAME	COLCOUNT	ROWCOUNT	...
ABTEILUNG	3	3	...
MITARBEITER	4	4	...
...	...	...	...
TABLES	...	...	...
...	...	...	...

Tabelle COLUMNS :

TABNAME	COLNAME	...
ABTEILUNG	ABTNR	...
ABTEILUNG	ABTNAME	...
ABTEILUNG	BUDGET	...
MITARBEITER	MNR	...
MITARBEITER	MNAME	...
MITARBEITER	ABTNR	...
MITARBEITER	GEHALT	...
...	...	...
TABLES	TABNAME	...
...	...	...

Der Katalog beschreibt auch sich selbst und enthält daher Informationen über die eigenen Tabellen ***(self-describing)***.

9.4 Basistabellen und Views

Es gibt **abgeleitete** ***(derived tables)*** Tabellen und **Basistabellen** ***(base tables)***. Eine abgeleitete Tabelle ist eine Tabelle, die unter Verwendung der relationalen Operatoren aus anderen Tabellen erzeugt wurde. Viele abgeleitete Tabellen haben keinen Namen. Wenn eine abgeleitete Tabelle einen Namen hat, dann ist es ein **View**.

Eine Basistabelle ist eine nicht abgeleitete Tabelle und hat immer einen Namen. Ein relationales System muss die Möglichkeit bieten, Basistabellen anzulegen. Basistabellen existieren "wirklich", d.h. sind Modelle für real existierende Daten, Views nicht. Views sind einfach eine andere Art auf die real existierenden Daten zu schauen. Wir sprechen von "Modellen", weil die Speicherstruktur der Daten ganz anders als die Basistabellen aussehen kann.

Wenn ein View angelegt wird, dann wird nur eine **virtuelle** ***(virtual)*** Tabelle angelegt. Der den View definierende Ausdruck wird erst ausgewertet, wenn der View tatsächlich benötigt wird. Für den Benutzer verhält sich aber ein View wie eine Tabelle. Änderungen an den Datenwerten im View wirken sich auch auf die entsprechenden Basistabellen aus.

9.5 Die Sprache SQL

SQL wurde in den frühen 70'er Jahren bei IBM entwickelt und hat sich zur Standard-Anfragesprache relationaler Systeme entwickelt. Wegen ihres praktischen Wertes werden wir diese Sprache künftig verwenden anstatt eines relationalen Pseudocodes, von dem wir oben Ansätze (`restrict` usw) gesehen haben. Da der Sprache später ein eigenes Kapitel gewidmet wird, sollen hier an Beispielen ohne formale Syntax nur Grundzüge dargestellt werden, die ad-hoc ergänzt werden. Englische Worte sind Schlüsselworte.

Die Basistabellen ABTEILUNG und MITARBEITER werden folgendermaßen erzeugt:

```
CREATE TABLE ABTEILUNG
   (
   ABTNR CHAR (2),
   ABTNAME  CHAR (20),
   BUDGET   INT,
   PRIMARY KEY (ABTNR)
   );


CREATE TABLE MITARBEITER
   (
   MNR   CHAR (2),
   MNAME CHAR (20),
   ABTNR CHAR (2),
   GEHALT   INT,
   PRIMARY KEY (MNR),
   FOREIGN KEY (ABTNR) REFERENCES ABTEILUNG
   );
```

Die Tabellen werden zeilenweise gefüllt mit:

```
INSERT INTO ABTEILUNG (ABTNR, ABTNAME, BUDGET)
               VALUES ('A1', 'Marketing', 10M);
```

Die relationalen Operationen `restrict`, `project` und `join` sind in SQL:

Operation: `restrict` ABTEILUNG `where` BUDGET > 6M
SQL:

```
SELECT ABTNR, ABTNAME, BUDGET
   FROM ABTEILUNG
   WHERE BUDGET > 6M;
```

Operation: `project` ABTEILUNG `over` ABTNR, BUDGET
SQL:

```
SELECT ABTNR, BUDGET
   FROM ABTEILUNG;
```

Operation: `join` ABTEILUNG `and` MITARBEITER `over` ABTNR
SQL:

```
SELECT ABTEILUNG.ABTNR, ABTNAME, BUDGET, MNR, MNAME, GEHALT
   FROM ABTEILUNG, MITARBEITER
   WHERE ABTEILUNG.ABTNR = MITARBEITER.ABTNR;
```

Die letzte Operation zeigt, dass manchmal qualifizierte Namen notwendig sind, um Ambiguitäten auszuschließen.

Das `SELECT`–Statement unterstützt alle drei grundlegenden relationalen Operationen.

Update-Operationen:

Die Tabelle ZWISCHENERGEBNIS

```
CREATE TABLE ZWISCHENERGEBNIS
   (
   MNR   CHAR (2)
   );
```

wird durch

```
INSERT INTO ZWISCHENERGEBNIS (MNR)
   SELECT   MNR
   FROM  MITARBEITER
   WHERE ABTNR = 'A1';
```

mit den MNR von allen Mitarbeitern gefüllt, die in der Marketing-Abteilung beschäftigt sind. Im folgenden wird allen diesen Mitarbeitern das Gehalt um 10% erhöht:

```
UPDATE MITARBEITER
   SET   GEHALT = GEHALT * 1.1
   WHERE ABTNR = 'A1';
```

Schließlich werden alle Mitarbeiter in der Entwicklungsabteilung entfernt:

```
DELETE
   FROM  MITARBEITER
   WHERE ABTNR = 'A2';
```

Abschließend wird ein View erzeugt:

```
CREATE VIEW VIELVERDIENER AS
   SELECT   MNR, MNAME, GEHALT
   FROM  MITARBEITER
   WHERE GEHALT > 66K;
```

9.6 Die Regeln von Codd

1986 hat Codd strenge Regeln definiert, die ein DBMS erfüllen muss, um als "relational" eingestuft werden zu können. Diese Forderungen sind allerdings so streng, dass kein System sie vollständig erfüllt.

1) The Information Rule: Alle Daten, die in einer Datenbank gespeichert werden, sind auf dieselbe Art dargestellt, nämlich durch Werte in Tabellen.

2) Guaranteed Access Rule: Jeder gespeicherte Wert muss über Tabellennname, Spaltenname und Wert des Primärschlüssels zugreifbar sein, wenn der zugreifende Anwender über hinreichende Zugriffsrechte verfügt.

3) Systematic Treatment of Null Values: Nullwerte müssen datentypunabhängig zur Darstellung fehlender Werte unterstützt werden. Systematisch drückt hierbei aus, dass Nullwerte unabhängig von demjenigen Datentyp, für den sie auftreten, gleich behandelt werden.

4) Dynamic On-line Catalog Based on the Relational Model: Der Katalog soll auch in Form von Tabellen vorliegen. Der Katalog beschreibt die Struktur und die Integritätsregeln der in der Datenbank hinterlegten Tabellen.

5) Comprehensive Data Sublanguage Rule: Für das DBMS muss mindestens eine Sprache existieren, durch die sich die Tabellenstruktur definieren lässt.

6) View Updating Rule: Alle theoretisch aktualisierbaren Sichten müssen durch Aktualisierung der zugrundeliegenden Basistabellen änderbar sein.

7) High-level Insert, Update, and Delete: Innerhalb einer Operation können beliebig viele Tupel bearbeitet werden, d.h. die Operationen werden grundsätzlich mengenorientiert ausgeführt.

8) Physical Data Independence: Änderungen an der internen Ebene dürfen keine Auswirkungen – außer Performanz – auf die Anwendungsprogramme haben.

9) Logical Data Independence: Änderungen des konzeptuellen Schemas dürfen keine Auswirkungen auf die Anwendungsprogramme haben, es sei denn, dass sie direkt von der Änderung betroffen sind. (Wegen ihnen sind die Änderungen vorgenommen worden.)

10) Integrity Independence: Alle Integritätsregeln müssen ausschließlich durch die Sprache des DBMS definierbar sein. Ferner gilt:

- Kein Attribut, das Teil eines Primärschlüssels ist, darf NULL sein.
- Ein Fremdschlüsselattribut muss als Wert des zugehörigen Primärschlüssels existieren.

11) Distribution Independence: Die Anfragesprache muss so ausgelegt sein, dass Zugriffe auf lokal gehaltene Daten identisch denen auf verteilt gespeicherte Daten formuliert werden können.

12) Nonsubversion Rule: Definiert ein DBMS neben der High-Level Zugriffssprache auch eine Schnittstelle mit niedrigerem Abstraktionsniveau, dann darf durch diese keinesfalls eine Umgehung der definierten Integritätsregeln möglich sein.

Rule 0: (Zusammenfassung aller zwölf Regeln) Alle Operationen für Zugriff, Verwaltung und Wartung der Daten dürfen nur mittels relationaler Fähigkeiten abgewickelt werden.

Es gibt am Markt kein DBS, das alle zwölf Regeln vollständig umsetzt. Insbesondere sind die Regeln 6, 9, 10, 11 und 12 in der Praxis schwer umzusetzen.

9.7 Lieferanten–und–Teile–Datenbank

Die Lieferanten–und–Teile–Datenbank *(suppliers–and–parts database)* besteht aus drei Tabellen S ***(suppliers)***, P ***(parts)*** und SP und wird in den folgenden Kapiteln häufig als Beispiel verwendet. Die Lieferanten S und die Teile P sind Entitäten, während die Lieferungen SP Beziehungen zwischen Lieferanten und Teilen sind.

Tabelle S:

SNR	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Tabelle P:

PNR	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

Tabelle SP:

SNR	PNR	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Übersetzung :

Englisch	Deutsch
Nut	Mutter
Bolt	Schraube mit Gewinde für Mutter (Maschinenschraube)
Screw	Schraube (Holzschraube)
Cam	Nocken
Cog(-wheel)	Zahn(-rad)

Kapitel 10

Relationale Datenobjekte: Relationen

Beim relationalen Modell spielen drei Aspekte eine Rolle: Datenstruktur oder –objekte, Datenintegrität, Datenmanipulation (relationale Algebra und relationales Kalkül). In diesem Kapitel beschäftigen wir uns mit der Datenstruktur bzw den Datenobjekten, nämlich Relationen und Wertebereichen.

Zunächst nennen wir – teilweise wiederholend – die im relationalen Modell wichtigen Begriffe in folgender Tabelle:

Relationales Modell		Informeller Begriff		Erklärung
Deutsch	Englisch	Deutsch	Englisch	
Relation	relation	Tabelle	table	
Tupel	tuple	Zeile	row	
Kardinalität	cardinality			Anzahl der Tupel
Attribut	attribute	Spalte	column	
Grad	degree		arity	Anzahl der Attribute
Primärschlüssel	primary key			Eindeutiger Identifikator
Wertebereich	domain	Typ	type	Werte, die ein Attribut annehmen kann
Skalar	scalar	Wert	value	Wert eines Attributs in einem Tupel

10.1 Wertebereiche

Das Konzept des **Wertebereichs** ***(domain)*** ist ein Beispiel dafür, daß nicht alle relationalen DBS alle Aspekte des relationalen Modells unterstützen. Die meisten DBS unterstützen Wertebereiche nicht.

Der **Skalar** ***(scalar)*** ist die kleinste semantische Dateneinheit. Er ist atomar insofern, als er *bezüglich des relationalen Modells* keine interne Struktur hat, was nicht bedeutet, daß ein Skalar überhaupt keine Struktur hat.

Ein Wertebereich ist eine benannte *Menge* von Skalaren. Diese Menge enthält alle Skalare, die ein Attribut annehmen kann. Andere Werte darf ein Attribut nicht annehmen. Jedes Attribut muß auf einem Wertebereich definiert sein.

Die Bedeutung der Wertebereiche ist ähnlich der Bedeutung der Typen in Programmiersprachen oder der Einheiten in der Physik. Postleitzahlen, Ortsnetz-Kennzahlen und Matrikelnummern sind eben unterschiedliche Typen von Zahlen.

Wertebereiche spielen eine Rolle bei:

Datenmanipulation: Ein Benutzer kann einem Attribut nur Werte aus dem Wertebereich des Attributs geben. Dies trägt wesentlich zur Datenintegrität bei.

Vergleiche (domain-constrained comparisons): Betrachte folgende SQL-Statements:

```
SELECT PNAME, QTY          |  SELECT PNAME, QTY
   FROM P, SP              |     FROM P, SP
   WHERE P.PNR = SP.PNR;   |     WHERE P.WEIGHT = SP.QTY;
```

Die linke Anfrage macht intuitiv Sinn. Denn hier werden Werte verglichen, die aus demselben Wertebereich kommen, nämlich Teilenummern. Die rechte Anfrage ist unsinnig, da hier zwar ganzzahlige Größen verglichen werden, aber diese Größen aus verschiedenen Wertebereichen stammen (Gewichte und Quantitäten), verschiedene Arten von ganzen Zahlen sind.

Wenn also zwei Attribute auf demselben Wertebereich definiert sind, dann machen solche Vergleiche (, die in vielen relationalen Operationen vorkommen,) einen Sinn. Wenn ein DBS daher das Wertebereichskonzept unterstützt, können viele Fehler vermieden werden.

Anfragen, die auf dem Wertebereich basieren: Wenn Wertebereiche unterstützt werden, kann man dem Katalog z.B. folgende Frage stellen:

"Welche Tabellen in der DB enthalten irgendwelche Information über Lieferanten?"

Ohne Wertebereichskonzept muß man sich bei solch einer Fragestellung darauf verlassen, daß die Attributnamen so gewählt sind, daß sie einen Hinweis auf Lieferanten enthalten.

Definition von Wertebereichen: Die Wertebereiche können in der DB elementweise gespeichert sein oder nur konzeptioneller Natur sein, z.B. ganzzahlig. Ein Wertebereich hat innerhalb der DB einen eindeutigen Namen, der aus Gründen der Benutzerfreundlichkeit ähnlich dem oder den Attributnamen ist. Der Name des Wertebereichs kann gleich dem Namen des Attributs sein. Jede Attributdefinition sollte eine Referenz auf einen Wertebereich enthalten. Mehrere Attribute können sich auf denselben Wertebereich beziehen.

Wertebereiche können eingerichtet (`create domain`) werden und wieder gelöscht (`destroy domain`) werden.

SQL unterstützt das Wertebereichskonzept leider *nicht*!

10.2 Relationen

10.2.1 Definition einer Relation

Definition: Eine Relation R auf den – nicht notwendig unterschiedlichen – Wertebereichen $D_1, D_2 \cdots D_m$ besteht aus zwei Teilen:

1. Der **Kopf** ***(heading)*** besteht aus einer *Menge* von Attributen, genauer ⟨Attributname : Wertebereichname⟩–Paaren,
   $\{\langle A_1 : D_1\rangle, \langle A_2 : D_2\rangle \cdots \langle A_m : D_m\rangle\}$
   sodaß zu jedem Attribut A_j genau ein Wertebereich $D_j (j = 1, 2 \cdots m)$ gehört. Die Attributnamen sind alle verschieden.
2. Der **Körper** ***(body)*** besteht aus einer *Menge* von Tupeln $\{t_1, t_2 \cdots t_n\}$. Jedes Tuple t_i besteht aus einer *Menge* von ⟨Attributname : Attributwert⟩–Paaren
   $t_i = \{\langle A_1 : v_{i1}\rangle, \langle A_2 : v_{i2}\rangle \cdots \langle A_m : v_{im}\rangle\}$
   $(i = 1, 2 \cdots n)$, wobei n die Anzahl der Tupel in der Menge ist. Die v_{ij} sind Werte aus dem Wertebereich D_j.

m ist die Anzahl der Attribute und heißt Grad der Relation, n ist die Anzahl der Tupel und heißt Kardinalität der Relation.

Eine Tabelle ist eine abgekürzte, bequeme, anschauliche Schreibweise für eine Relation! Diese Schreibweise suggeriert aber einige Dinge, die für Relationen nicht gelten:

- Zeilen sind in einer Tabelle geordnet und können als Duplikate vorkommen. In einer Relation gibt es keine Duplikate und keine Ordnung.
- Die Spalten haben in einer Tabelle eine bestimmte Reihenfolge. Die Attribute in einer Relation haben keine Ordnung.
- In einer Tabelle wird nicht deutlich, daß die Werte aus Wertebereichen stammen.

Der Körper einer Relation ist ihr **Wert** ***(value)***, der sich im Lauf der Zeit ändern kann. Jede Wertänderung ergibt eigentlich eine neue Relation. Das ist lästig. Deshalb werden wir Relationen häufig wie Variable eines Typs behandeln (Codd: ***time-varying relations)***, wobei der Typ einer Relation durch den Kopf der Relation gegeben ist.

Der **Grad** ***(degree*** oder auch ***arity)*** einer Relation ist die Anzahl der Attribute. Man spricht von *unären (unary)*, *binären (binary)*, *ternären (ternary)* usw Relationen.

Ein DBMS muß das Erzeugen und Zerstören von Relationen erlauben.

10.2.2 Eigenschaften von Relationen

Die folgenden Eigenschaften folgen aus der Definition der Relation:

- Es gibt keine Tupelduplikate: Das folgt aus der Tatsache, daß eine Relation eine Menge von Tupeln ist. SQL erlaubt leider Tupelduplikate, wohl aus Performanzgründen. Auf die "guten" Gründe, warum Tupelduplikate schlecht sind, kann hier nicht eingegangen werden.

- Die Tupel sind nicht geordnet: Das folgt wieder aus der Mengeneigenschaft. Aussagen wie etwa "das fünfte Tupel" oder "das nächste Tupel" oder "das letzte Tupel" sind sinnlos. Der Benutzer wird gezwungen, die Tupel bei "Namen" zu nennen.
- Die Attribute sind nicht geordnet: Es gibt also auch kein "erstes", "nächstes" oder "letztes" Attribut. In der Praxis bedeutet das, daß man gezwungen wird, die Attribute bei Namen zu nennen, was die Fehlermöglichkeiten gewaltig reduziert.
- Alle Attributwerte sind atomar: Das leitet sich ab aus der Tatsache, daß die Wertebereiche atomar sind. Relationen haben keine ***repeating groups***. Man sagt auch, eine Relation ist **normalisiert** ***(normalized)*** oder ist in der **ersten Normalform** ***(first normal form)***.

Bemerkung: In der Literatur findet man auch andere Begriffsbildungen. Bei Heuer und Saake[22] ist der Kopf einer Relation das **Relationenschema** und der Körper einer Relation heißt **Relation**. Was wir "Relation R" nennen kann dann nur umständlich als "Relationenschema R und seine Relation r" oder "Relation r zum Relationenschema R" bezeichnet werden. Diese Trennung von "Kopf" und "Körper" halten wir für sehr hinderlich. Sie führt zu umständlichen Bezeichnungsweisen, auf die auch bei Heuer und Saake oft genug verzichtet wird.

10.2.3 Relationsarten

Benannte ***(named)*** **Relation** ist eine Relation, die unter einem Namen ansprechbar ist (Basisrelation, View, Schnappschuß).

Basisrelation ***(base relation)*** ist eine benannte, autonome *(autonomous)*, d.h. nicht abgeleitete Relation.

Abgeleitete ***(derived)*** **Relation** wird durch andere Relationen in relationalen Ausdrücken definiert. Sie wird letztlich von Basisrelationen abgeleitet.

Ausdrückbare ***(expressible)*** **Relation** ist eine Relation, die mit relationalen Mitteln erhalten werden kann. Die ausdrückbaren Relationen sind alle Relationen, Basis– und abgeleitete Relationen.

View ist eine benannte, abgeleitet Relation. Views sind virtuell, da sie im DBS nur durch ihre Definition in Termen anderer Relationen repräsentiert werden.

Schnappschuß ***(snapshot)*** ist eine benannte abgeleitete Relation, die aber nicht virtuell ist, sondern die – konzeptionell – wenigsten durch ihre eigenen Daten repräsentiert wird. Schnappschüsse können daher veralten und müssen eventuell erneuert ***(refresh)*** werden. Das kann periodisch erfolgen.

Anfrageresultat ***(query result)*** ist eine nicht benannte abgeleitete Relation, die das Resultat einer Anfrage an die DB ist. Sie sind nicht persistent.

Zwischenresultat ***(intermediate result)*** ist eine nicht benannte abgeleitete Relation, die das Resultat eines relationalen Ausdrucks ist, der in einen anderen Ausdruck geschachtelt ist.

Gespeicherte ***(stored)*** **Relation** ist eine Relation, die direkt und effizient physikalisch gespeichert ist. Es gibt nicht unbedingt eine Eins-zu-Eins-Beziehung zwischen Basisrelationen und gespeicherten Relationen.

10.3 Prädikate

Für die praktische Anwendung hat jede Relation eine Interpretation oder Bedeutung *(interpretation, meaning)*. Z.B. hat die Relation S folgende Bedeutung:

> *Ein Lieferant mit einer bestimmten Lieferantennummer (SNR) hat den spezifizierten Namen (SNAME) und den spezifizierten Statuswert (STATUS) und sitzt in der angegebenen Stadt (CITY). Ferner haben verschiedene Lieferanten nicht dieselbe Lieferantennummer.*

Diese zwei Sätze sind ein Beispiel für ein **Prädikat** ***(predicate)*** oder eine **Wahrheitswertfunktion** von – in diesem Fall vier – Argumenten (SNR, SNAME, STATUS, CITY). Das Einsetzen von Werten für die Argumente ist äquivalent zum Aufruf einer Funktion (Instanziierung des Prädikats) und ergibt eine **Proposition**, die entweder wahr oder falsch ist.

Z.B. ergibt die Substitution

```
SNR = 'S1'; SNAME = 'Smith'; STATUS = 20; CITY = 'London'
```

eine wahre Proposition und die Substitution

```
SNR = 'S1'; SNAME = 'Maier'; STATUS = 45; CITY = 'Stuttgart'
```

eine falsche Proposition.

Eine Relation enthält zu jeder Zeit nur solche Tupel, die die entsprechenden Propositionen zu der Zeit wahr machen.

Das DBMS muß bei der Manipulation von Tupeln dafür sorgen, daß die entsprechenden Prädikate nicht verletzt werden, d.h. daß die entsprechenden Propositionen wahr bleiben.

Das Prädikat einer Relation ist das **Kriterium für die Akzeptanz einer Manipulation** ***(criterion for update acceptability)***.

Das DBMS kennt das Prädikat von vornherein nicht und es ist i.a. nicht möglich, dem DBMS das Prädikat vollständig zu beschreiben. Denn u.U. müßte man alle möglichen Tupel und Tupel-Kombinationen auflisten.

Aber man kann dem DBMS Regeln geben, die eine vernünftige Annäherung an das Prädikat sind. In unserem Beispiel sind das etwa folgende Regeln:

1. SNR muß ein Wert aus dem Wertebereich der Lieferantennummern sein.
2. SNAME muß einen Wert aus dem Wertebereich der Namen haben.
3. CITY muß ein Wert aus dem Wertebereich der Stadtnamen sein.
4. SNR muß eindeutig sein bezüglich der anderen SNR-Werte in der Relation.

Diese Regeln heißen **Integritätsregeln** ***(integrity rules)*** und definieren die "Bedeutung" einer Relation für das DBMS.

10.4 Relationale DB

Eine relationale DB ist eine DB, die für den Benutzer eine Ansammlung von normalisierten variablen Relationen verschiedenen Grads ist.

Folgende Entsprechungen mit traditionellen DB-Begriffen können gesehen werden:

relational		traditionell	
Deutsch	Englisch	Deutsch	Englisch
Relation	relation	Datei	File
Tupel	tuple	Datensatz	record
Attribut	attribute	Feld	field

Kapitel 11

Relationale Datenintegrität: Schlüssel

Zur Definition des Begriffs Integrität betrachten wir folgendes:

1. In jedem Zeitpunkt enthält eine DB eine bestimmte Konfiguration von Daten, die die "Wirklichkeit darstellt", d.h. die ein Modell oder eine Repräsentation eines Teils der realen Welt ist.

2. Gewisse Datenkonfigurationen machen keinen Sinn, indem sie nicht einen möglichen Zustand der realen Welt darstellen. Z.B. können Gewichte keine negativen Werte annehmen.

3. Die DB–Definition muß daher bezüglich gewisser **Integritätsregeln** ***(integrity rules)*** erweitert werden. Diese Regeln informieren das DBMS über Randbedingungen der realen Welt. Damit kann das DBMS verbotene Datenkonfigurationen verhindern.

In unserem Beispiel könnten die Regeln folgendermaßen aussehen:

- Lieferantennummern müssen die Form S*nnnn* haben, wobei *nnnn* für vier Zahlen zwischen 0 und 9 steht.
- Teilenummern müssen die Form P*nnnnn* haben.
- Lieferantenstatuswerte liegen zwischen 1 und 100.
- Lieferanten- und Teilestädte müssen aus besonderen Listen genommen werden.
- Teilegewichte müssen größer Null sein.
- Alle roten Teile gibt es nur in London.
- usw

All diese Integritätsregeln sind *DB-spezifisch* in dem Sinn, als Sie nur für eine DB gelten. Es gibt aber auch allgemeine Integritätsregeln und diese haben mit **Schlüsseln** ***(keys)*** zu tun, speziell **Schlüssel** oder **Kandidatenschlüssel** ***(candidate key)*** und **Fremdschlüssel** ***(foreign key)***.

Abgeleitete Relationen erben die Integritätsregeln der Basisrelationen. Sie haben möglicherweise zusätzliche Integritätsregeln. Für einen View kann etwa ein zusätzlicher Schlüssel definiert werden.

11.1 (Kandidaten-)Schlüssel

Definition: Ein **(Kandidaten-)Schlüssel** ***(candidate key)*** K ist auf einer Relation R definiert als eine Teilmenge des Kopfes (der Menge der Attribute) von R mit den Eigenschaften:

1. **Eindeutigkeit** ***(uniqueness)***: Es gibt keine zwei unterschiedliche Tupel von R mit demselben Wert für K.
2. **Nicht-Reduzierbarkeit** ***(irreducibility***, auch ***minimality)***: Keine echte Teilmenge von K hat die Eigenschaft der Eindeutigkeit.

Bemerkungen und Folgerungen:

1. Jede Relation hat mindestens *einen* Schlüssel. Da eine Relation keine Duplikate enthält, hat die Kombination aller Attribute die Eigenschaft der Eindeutigkeit und hat entweder die Eigenschaft der Nicht-Reduzierbarkeit. Dann ist sie ein Schlüssel. Oder sie ist reduzierbar. Dann existiert mindestens eine echte Teilmenge aller Attribute mit der Eigenschaft der Eindeutigkeit. Für diese Teilmenge gilt wieder dasselbe, wie für alle Attribute. Man kann also soweit reduzieren, bis man einen Schlüssel findet, d.h. eine Teilmenge findet, die beide Eigenschaften hat.

2. Reduzierbare Schlüssel heißen auch **Superschlüssel** ***(superkey)*** oder **Oberschlüssel**.

3. Wenn wir sagen, daß die Attribut-Kombination {SNR, PNR} ein Schlüssel der Relation SP ist, dann meinen wir, daß das nicht nur für einen bestimmten Wert der Relation SP gilt, sondern für alle möglichen Werte der Relation SP gilt.

4. Bilden alle Attribute den Schlüssel, dann nennt man die Relation eine "all-key"–Relation. Wenn man z.B. das Attribut QTY aus der Relation SP entfernt, dann hat man eine all-key-Relation.

5. Viele Relationen haben in der Praxis nur einen Schlüssel. Fälle mit mehr Schlüsseln sind selten. Wenn man z.B. Automarken verwaltet und den Marken Identifikationsnummern vergibt, dann ist wahrscheinlich diese Identifikationsnummer und der Name der Automarke ein Schlüssel, da mir keine zwei Automarken mit demselben Namen bekannt sind.

6. Ein Schlüssel, der mehr als ein Attribut enthält, heißt **zusammengesetzt** ***(composite)***, sonst **einfach** ***(simple)***. Schlüssel sind Mengen und sollten daher immer mit geschweiften Klammern dargestellt werden. Bei einfachen Schlüsseln werden wir aber häufig die geschweiften Klammern weglassen, d.h. anstatt {SNR} schreiben wir nur SNR.

7. Ein Attribut, das zu einem Schlüssel gehört, heißt **primär** ***(prime)***.

8. Die Forderung der Nicht-Reduzierbarkeit ist notwendig, da das DBMS z.B. bei dem reduzierbaren Schlüssel {SNR, CITY} nur darauf achten würde, daß die SNR innerhalb einer Stadt eindeutig sind, anstatt daß SNR global eindeutig ist. {SNR} wäre dann kein Schlüssel mehr.

9. Das Konzept "Schlüssel" sollte nicht mit dem Konzept "eindeutiger Index" auf der internen Ebene verwechselt werden. D.h. es soll nicht impliziert werden, daß es auf physikalischer Ebene einen Index auf den Schlüssel gibt. Natürlich wird sich in der Praxis die Implementation eines Indexes häufig an den Schlüsseln orientieren.

10. Wozu sind Schlüssel gut? Sie sind der zentrale Mechanismus zur Adressierung von Tupeln. Durch Angabe eines Schlüsselwerts kann ein Tupel eindeutig spezifiziert werden. Das ist der tiefere Grund, warum Tupel-Duplikate verboten sind. Schlüssel sind für das Funktionieren eines relationalen Systems ähnlich wichtig wie Hauptspeicheradressen für das Funktionieren eines Rechners.

11.2 Primärschlüssel

Genau ein Schlüssel kann als **Primärschlüssel** ***(primary key)*** definiert werden. Ob ein Primärschlüssel benötigt wird, ist unklar. Die meisten DBMS lassen nur die Definition von Primärschlüsseln zu. Bei Relationen mit *einem* Schlüssel ist diese Unterscheidung hinfällig. Ob diese Unterscheidung bei Relationen mit mehreren Schlüsseln wichtig ist, ist zur Zeit unklar und für die Praxis unwichtig.

Schlüssel, die nicht Primärschlüssel sind, heißen **Alternativschlüssel** ***(alternate key)*** oder **Sekundärschlüssel** ***(secondary key)***.

Der Primärschlüssel wird in den Tabellen **unter**strichen.

11.3 Fremdschlüssel

Betrachten wir das Attribut SNR in der Relation SP. Die DB kann nur dann in einem Zustand der Integrität sein, wenn jeder Wert von SNR in SP auch als Wert in der Relation S unter dem Attribut SNR vorkommt. Es macht keinen Sinn die Lieferung eines Lieferanten in SP aufzuführen, der nicht in S enthalten ist. Ähnliches gilt für das Attribut PNR. Die Attribute SNR und PNR in SP sind Beispiele für **Fremdschlüssel** ***(foreign key)***. Daß diese beiden Fremdschlüssel den (Primär-)Schlüssel für SP bilden ist mehr oder weniger Zufall bzw gehört nicht zur Fremdschlüsseleigenschaft.

Fremdschlüssel passen i.a. zu einem Primärschlüssel. Es kann aber auch Fälle geben, wo ein Alternativschlüssel Sinn macht. Die folgende Definition von Fremdschlüssel zeigt das:

Definition: K sei ein Schlüssel der Basisrelation R1. R2 sei eine (nicht notwendigerweise von R1 verschiedene) Basisrelation. FK ist ein **Fremdschlüssel** in der Basisrelation R2, wenn für alle aktuellen Werte von R1 und R2 gilt:

1. FK ist eine Teilmenge des Kopfes (der Attribute) von R2.
2. Jeder Wert von FK in R2 ist identisch mit einem Wert von K in R1.

Fremdschlüssel werden in den Tabellen **über**strichen.

Bemerkungen:

1. Jeder Wert eines Fremdschlüssels in R2 muß zu einem Schlüsselwert in R1 passen. Das Umgekehrte muß nicht der Fall sein.

2. Der Fremdschlüssel ist nur dann zusammengesetzt, wenn auch der dazugehörige Schlüssel zusammengesetzt ist.

3. Die Wertebereiche der Attribute müssen im Fremdschlüssel und im dazugehörigen Schlüssel gleich sein.

4. Die Namen der Attribute können im Fremdschlüssel und im dazugehörigen Schlüssel unterschiedlich sein. Insbesondere wenn R1 gleich R2 ist, dann müssen sie unterschiedlich sein.

5. Ein Fremdschlüssel von R2 muß nicht die Komponente eines Schlüssels von R2 sein. Beispiel sind die Tabellen ABTEILUNG und MITARBEITER.

6. Begriffe: Ein Fremdschlüssel steht für eine **Referenz** ***(reference)*** auf das Tupel mit dem passenden Schlüssel (**referenziertes Tupel**, ***referenced tuple, target tuple)***. Das Problem oder die Aufgabe, dafür zu sorgen, daß ein Fremdschlüssel immer gültige Werte enthält, ist bekannt unter dem Namen **Problem der refenziellen Integrität** ***(referential integrity problem)***. Es muß die **referenzielle Einschränkung** ***(referential constraint)*** beachtet werden. Der Fremdschlüssel ist in der **referenzierenden** ***(referencing)***, der dazugehörige Schlüssel in der **referenzierten** ***(referenced, target)*** Relation definiert.

7. **Referenzielle Diagramme** ***(referential diagrams)***: Die Bezüge zwischen den Relationen werden graphisch mit Pfeilen dargestellt:

   $R_1 \leftarrow R_2$

   $S \leftarrow SP \rightarrow P$

   Dabei können die Pfeile noch mit den entsprechenden Attributen versehen werden, wenn Ambiguitäten auftreten.

8. Eine Relation kann referenzieren und referenziert werden. Dadurch ergeben sich **referenzielle Pfade** ***(referential path)***.

9. Es gibt Relationen, die sich selbst referenzieren *(self-referencing)*. Sie sind ein Spezialfall von **referenziellen Zyklen** ***(referential cycle)***:

   $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \rightarrow R_1$

10. Schlüssel – Fremdschlüsselbeziehungen sind die wichtigsten Beziehungen zwischen Relationen in einer DB. Es gibt aber auch andere Beziehungen: Z.B. kommt das Attribut CITY in S und P vor. Man kann nun fragen, ob ein Lieferant und ein Teil in derselben Stadt sind. CITY ist kein Fremdschlüssel. Diese Art von Fragen kann aber nur eindeutig beantwortet werden, wenn CITY ein Schlüssel wird.

11. Referentielle Integrität ist ein Begriff, mit dem Abhängigkeiten oder Beziehungen zwischen Objekten beschrieben werden und hat über RDBs hinausgehende allgemeine Bedeutung für andere Datenbanktypen, Programmiersprachen und Modellierung.

11.3.1 Fremdschlüsselregeln

Wir haben uns noch nicht damit beschäftigt, was das DBMS tun soll in den Fällen, wo eine DB-Manipulation die referenzielle Integrität verletzt. Für jeden Fremdschlüssel sollte folgende Frage beantwortet werden:

Was passiert, wenn versucht wird, das Target einer Fremdschlüsselreferenz zu manipulieren, d.h. zu löschen oder zu verändern? Z.B. Löschung eines Lieferanten, der in der Relation SP vorkommt, oder Veränderung des Schlüsselwerts eines Lieferanten, der in SP vorkommt. Es gibt verschiedene Möglichkeiten, darauf zu reagieren:

1. **beschränkt** ***(restricted)***: Die Manipulation wird auf die Fälle ohne Referenz beschränkt und in anderen Fällen zurückgewiesen.
2. **kaskadiert** ***(cascades)***: Die Manipulation kaskadiert insofern, als sie alle referenzierenden Einträge auch manipuliert.
3. Eintrag einer NULL oder eines Defaultwerts in den Fremdschlüssel. Das ist nur vernünftig, wenn der Fremdschlüssel nicht Komponente eines Schlüssels ist.
4. Frage den Benutzer, was zu tun ist.
5. Rufe eine installationsabhängige Prozedur auf.

Diese Fremdschlüssel-Regeln ***(foreign key rules)*** müssen bei der Definition eines Fremdschlüssels festgelegt werden.

11.4 Nicht definierte Werte (NULL)

NULL steht für einen nicht definierten Wert oder fehlende Information *(missing value or missing information)*. Sie haben in SQL den Wert `NULL`. Es gibt über den Sinn oder Unsinn von NULLs eine ausgedehnte und wiedersprüchliche Literatur.

Defaultwerte sind nicht NULLs, sondern gültige Werte.

NULLs sind notwendig, da häufig Information einfach nicht bekannt ist.

NULLs unterminieren die ganze schöne relationale Theorie.

Wir stellen einige Regeln auf über die Verwendung von NULLs:

1. Nach Möglichkeit sollten anstatt von NULLs Defaultwerte verwendet werden.
2. Ein Schlüssel darf keine NULLs enthalten.
3. Ein Fremdschlüssel darf nur NULLs enthalten, wenn er nicht Komponente eines Schlüssels ist.

Kapitel 12

Relationale Operatoren I: Relationale Algebra

Die relationale Algebra, wie sie von Codd definiert wurde, besteht aus acht Operatoren:

- Traditionelle Mengenoperationen: *Vereinigung* (`union`, $\cup$), *Schnitt* (`intersect`, $\cap$), *Differenz* (`minus`, $-$) und *kartesisches Produkt* (`times`, $\times$ oder $\bowtie$), die alle etwas auf Relationen angepasst wurden.
- Spezifisch relationale Operationen: `select` (σ), `project` (π), `join` ($\bowtie$) und `divideby` ($/$). (Anstatt `select` wurde ursprünglich von Codd `restrict` verwendet.)

Dazu kommen zwei weitere wichtige Operationen `extend` und `summarize`, ferner Zuweisungs- und Update-Operationen und relationale Vergleiche.

Die Operationen sind so definiert, dass die Menge der Relationen bezüglich dieser Operationen abgeschlossen ist. D.h. die Eingabe der Operationen sind Relationen und die Ausgabe der Operationen sind wieder Relationen. Das erlaubt die Konstruktion **verschachtelter Ausdrücke *(nested expressions)***.

Insbesondere werden die Operationen so definiert, dass die resultierenden Relationen vernünftige Köpfe haben. Dazu mag es notwendig sein, Attribute umzubenennen. Wir führen daher den Operator `rename` ein mit der Syntax:

Relation `rename` alterAttributname `as` neuerAttributname

Symbolische Notation:

$$\beta_{\text{neu}\leftarrow\text{alt}} R$$

Dieser Ausdruck ergibt die Relation R mit geändertem Attributnamen.

12.1 Traditionelle Mengenoperationen

Die traditionellen Mengenoperationen sind Vereinigung, Schnitt, Differenz und kartesisches Produkt.

Bei Vereinigung, Schnitt und Differenz müssen die Operanden typkompatibel sein. Zur Not muss eventuell vorher eine `rename`-Operation durchgeführt werden, um die Typen anzupassen.

Vereinigung, Schnitt und kartesisches Produkt sind in der relationalen Algebra assoziativ und kommutativ, die Differenz nicht.

Als Beispiel betrachten wir die beiden Relationen:

Relation R_1:

SNR	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

Relation R_2:

SNR	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

12.1.1 Vereinigung

Die Vereinigung von zwei Relationen R_1 und R_2 bedeutet die mengentheoretische Vereinigung der beiden Tupelmengen. Das Resultat ist die Menge der Tupel, die entweder in R_1 oder in R_2 oder in beiden Relationen sind. (Duplikate gibt es nicht.)

Das Resultat der Vereinigung von zwei beliebigen Relationen ist i.a. aber keine Relation mehr. Denn eine Relation darf nur Tupel von einem Typ enthalten. Das bedeutet, dass die Vereinigung in der relationalen Algebra eine spezielle Form hat: Die Eingaberelationen müssen **typkompatibel** sein, d.h. sie müssen denselben Kopf haben. Das Resultat hat dann auch denselben Kopf.

Beispiel:

R_1 `union` R_2

$(R_1 \cup R_2)$:

SNR	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris

Der Operator `outer union` wurde als Erweiterung entwickelt, um partiell typkompatible Relationen zu vereinigen, indem die nicht kompatiblen Attribute jeweils mit NULLs aufgefüllt werden.

12.1.2 Differenz

Die Differenz von zwei typkompatiblen Relationen R_1 `minus` R_2 ist eine Relation mit demselben Kopf wie R_1 und R_2 und Tupeln, die zu R_1, nicht aber zu R_2 gehören.

Beispiel:

R_1 `minus` R_2

$(R_1 - R_2)$:

SNR	SNAME	STATUS	CITY
S4	Clark	20	London

R_2 `minus` R_1

$(R_2 - R_1)$:

SNR	SNAME	STATUS	CITY
S2	Jones	10	Paris

12.1.3 Schnitt

Der Schnitt von zwei typkompatiblen Relationen R_1 und R_2 ist eine Relation mit demselben Kopf wie R_1 und R_2 und Tupeln, die zu R_1 *und* zu R_2 gehören.

Beispiel:

R_1 `intersect` R_2

$(R_1 \cap R_2)$:

SNR	SNAME	STATUS	CITY
S1	Smith	20	London

Der Schnitt ist ausdrückbar mit Hilfe von `minus`:
R_1 `intersect` $R_2 \equiv R_1$ `minus` $(R_1$ `minus` $R_2)$

Der Schnitt ist daher keine **primitive** Operation.

12.1.4 Kartesisches Produkt

In der Mathematik besteht das kartesische Produkt von zwei Mengen R_1 und R_2 aus allen *geordneten* Paaren (r_1, r_2), wobei $r_1 \in R_1$ und $r_2 \in R_2$ gilt.

Übertragen auf Relationen würde das bedeuten, dass wir geordnete Tupelpaare erhalten würden. Das wäre aber keine Relation. Das Produkt wird also folgendermaßen modifiziert:

Jedes geordnete Tupelpaar koalesziert in ein Tupel, was zunächst einfach eine mengentheoretische Vereinigung der beiden Tupel ist. Damit aber die resultierende Tupelmenge eine Relation wird, muss auch ein Kopf gebildet werden, nämlich durch Vereinigung der beiden Input-Köpfe. Wenn zwei Attributnamen gleich sind, dann müssen diese vorher umbenannt werden.

Diese Definition macht das kartesische Produkt auch kommutativ. Das kartesiche Produkt kann nur von zwei Relationen gebildet werden, die keine gemeinsamen Attribute haben.

Konzeptionell wird das kartesische Produkt sehr oft verwendet, obwohl ein praktisches Datenbanksystem selten wirklich das Produkt ausführt.

Beispiel:

Relation R_3:

SNR
S1
S2

Relation R_4:

PNR
P1
P2
P3

R_3 `times` R_4

$(R_3 \times R_4)$:

SNR	PNR
S1	P1
S1	P2
S1	P3
S2	P1
S2	P2
S2	P3

12.2 Spezifisch relationale Operationen

12.2.1 Selektion (Restriktion)

`select` (oder `restrict`) ist eine Abkürzung für Θ-Selektion (oder Θ-Restriktion), wo Θ für einen einfachen skalaren Vergleichsoperator $(=, \neq, >, \geq, \ldots)$ steht.

Die Selektion der Relation R auf den Attributen oder Skalaren A und B

R `where` $A\Theta B$

$(\sigma_{A\Theta B} R)$

ist eine Relation mit dem gleichen Kopf wie R und mit einem Körper, der aus der Teilmenge aller Tupel von R besteht, die die Bedingung $A\Theta B$ erfüllen. A und B müssen dieselben Wertebereiche haben.

Hinter `where` können wir syntaktisch boolsche Ausdrücke von Bedingungen erlauben, da diese äquivalent zu Mengenoperationen sind:

R `where` φ_1 `and` φ_2 $\equiv$ $(R$ `where` $\varphi_1)$ `intersect` $(R$ `where` $\varphi_2)$
R `where` φ_1 `or` φ_2 $\equiv$ $(R$ `where` $\varphi_1)$ `union` $(R$ `where` $\varphi_2)$
R `where not` φ $\equiv$ R `minus` $(R$ `where` $\varphi)$

Hiermit ist auch die syntaktische Kategory *condition* (φ) definiert, nämlich als boolscher Ausdruck von Θs.

Beispiel:

S `where` CITY = 'London':

SNR	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

P `where` WEIGHT $<$ 14:

PNR	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12	London
P5	Cam	Blue	12	Paris

SP `where` SNR = 'S1' `and` PNR = 'P1':

SNR	PNR	QTY
S1	P1	300

12.2.2 Projektion

Die Projektion der Relation R auf die Attribute (von R) $A, B \ldots C$

$$R[A, B \ldots C]$$

$$(\pi_{A,B \ldots C} R)$$

oder

$$R[\{A, B \ldots C\}]$$

ist eine Relation mit dem Kopf $\{A, B \ldots C\}$ und einem Körper, der aus allen Tupeln der Form $\{A{:}a,\ B{:}b \ldots C{:}c\}$ besteht, wobei es in R ein Tupel gibt mit A-Wert a, B-Wert $b \ldots C$-Wert c.

Projektion bedeutet das Streichen von Spalten in R. Eventuell anfallende Duplikate werden eliminiert.

Die Projektion

$$R$$

ohne Kommaliste ist die identische Projektion und liefert wieder R. Die Projektion mit leerer Kommaliste

$$R\ [\,]$$

ist auch erlaubt. Sie repräsentiert die **Nilprojektion** ***(nullary projection)*** und liefert eine leere Relation.

Beispiel:

S [CITY]:

CITY
London
Paris
Athens

P [COLOR, CITY]:

COLOR	CITY
Red	London
Green	Paris
Blue	Rome
Blue	Paris

(S `where` CITY = 'Paris') [SNR]:

SNR
S2
S3

12.2.3 Join

Natural Join

Die **Join**-Operation (auch **Natürlicher Join**, ***natural join)*** ist folgendermaßen definiert:

R_1 sei eine Relation mit dem Kopf

$\{ A_1 \ldots A_m, B_1 \ldots B_n \}$

und R_2 eine Relation mit dem Kopf

$\{ C_1 \ldots C_p, B_1 \ldots B_n \}$

d.h die Attribute $B_1 \ldots B_n$ seien R_1 und R_2 gemeinsam und jeweils auf demselben Wertebereich definiert. Betrachten wir $A_1 \ldots A_m$, $B_1 \ldots B_n$ und $C_1 \ldots C_p$ als drei zusammengesetzte Attribute A, B und C. Der natürliche Join von R_1 und R_2

R_1 `natural join` R_2

$(R_1 \bowtie_{\texttt{natural}} R_2)$

ist eine Relation mit dem Kopf $\{ A, B, C \}$ und einem Körper, der aus allen Tupeln $\{ A{:}a, B{:}b, C{:}c \}$ besteht, wobei es ein Tupel in R_1 gibt mit A-Wert a und B-Wert b und ein Tupel in R_2 gibt mit C-Wert c und B-Wert b.

Der natürliche Join ist assoziativ und kommutativ. Es muss nicht sein, dass Schlüssel dabei beteiligt sind.

Beispiel:

S `natural join` P:

SNR	SNAME	STATUS	CITY	PNR	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12
S1	Smith	20	London	P4	Screw	Red	14
S1	Smith	20	London	P6	Cog	Red	19
S2	Jones	10	Paris	P2	Bolt	Green	17
S2	Jones	10	Paris	P5	Cam	Blue	12
S3	Blake	30	Paris	P2	Bolt	Green	17
S3	Blake	30	Paris	P5	Cam	Blue	12
S4	Clark	20	London	P1	Nut	Red	12
S4	Clark	20	London	P4	Screw	Red	14
S4	Clark	20	London	P6	Cog	Red	19

Wenn R_1 und R_2 keine gemeinsamen Attribute haben, dann ist der Join äquivalent zum kartesischen Produkt. In der Tat kann der Join als Produkt mit nachgeschalteter Selektion und Projektion formuliert werden. Unser Beispiel würde folgendermaßen lauten:

S `natural join` P ≡

((S `times` (P `rename` CITY `as` PCITY)) `where` CITY = PCITY)
[SNR, SNAME, STATUS, CITY, PNR, PNAME, COLOR, WEIGHT]

Allgemein:

A `natural join` B ≡

((A `times` (B `rename` Y `as` W)) `where` Y = W)[X, Y, Z]

Der Join ist daher keine primitive Operation.

Θ-Join

Der Θ-Join ist das kartesische Produkt mit nachgeschalteter Selektion.

Sei R_1 eine Relation mit dem Attribut A und R_2 eine Relation mit dem Attribut B, dann ist der Θ-Join der Relation R_1 auf Attribut A mit Relation R_2 auf Attribut B

$$(R_1 \texttt{ times } R_2) \texttt{ where } A\Theta B$$

$$(R_1 \bowtie_{A\Theta B} R_2 \equiv \sigma_{A\Theta B}(R_1 \times R_2))$$

eine Relation mit dem Kopf des kartesischen Produkts und allen Tupeln des kartesischen Produkts, die die Bedingung $A\Theta B$ erfüllen.

Der ***equijoin*** **(Gleichverbund)** ist ein Θ-Join, wobei Θ das Gleichheitszeichen ist. Häufigste Joins überhaupt sind Equijoins

$$R_1 \bowtie_K R_2,$$

wobei K Schlüssel in R_1 und Fremdschlüssel in R_2 ist ($R_1 \xleftarrow{K} R_2$).

Noch allgemeiner kann man den Join als ein kartesisches Produkt sehen, bei dem jedes resultierende Tupel r bezüglich eines Prädikats φ ausgewertet wird und nur die Tupel im Resultat erscheinen, für die $\varphi(r) = \texttt{true}$.

$$R_1 \bowtie_\varphi R_2$$

Allgemeine Joins sind noch kommutativ, aber i.a. nicht mehr assoziativ. Man betrachte dazu als Beispiel

$$(SP \bowtie_{\texttt{SNR}} S) \bowtie_{\texttt{PNR}} P$$

und

$$SP \bowtie_{\texttt{SNR}} (S \bowtie_{\texttt{PNR}} P)$$

Da S keine Spalte `PNR` hat, kann die Join-Bedingung im zweiten Fall nicht vernünftig formuliert werden.

Outer Join

Bei der bisher beschriebenen Join-Operation (auch `inner join` genannt) fallen Tupel unter den Tisch, die in der anderen Tabelle keinen passenden Partner haben ***(dangling tupels)***. Bei der Frage nach Lieferantennamen und Namen der gelieferten Teile kommt der Lieferant S5 nicht vor, da er nichts geliefert hat. Wenn man ihn trotzdem in der Tabelle mit einem leeren Teilenamen führen will, muss man einen ***outer join*** durchführen. Dabei werden Tupel erhalten, die keinen passenden Partner haben. Für den fehlenden Partner werden NULLs eingefügt.

Es gibt drei Möglichkeiten:

R_1 `left outer join` R_2 $\equiv$
: Nur Tupel des linken Partners R_1 werden erhalten.

R_1 `right outer join` R_2 $\equiv$
: Nur Tupel des rechten Partners R_2 werden erhalten.

R_1 `full outer join` R_2 $\equiv$
: Tupel des linken und rechten Partners werden erhalten.

Die Symbole dazu sind:

$$\bowtie^l \quad \bowtie^r \quad \bowtie^f$$

Betrachten wir die Relation:

S `full outer natural join` P ?

Wieviel Tupel hat diese Relation?

12.2.4 Division

Die Division ist folgendermaßen definiert:

R_1 sei eine Relation mit dem Kopf

$\{ A_1 \ldots A_m, B_1 \ldots B_n \}$

und R_2 eine Relation mit dem Kopf

$\{ B_1 \ldots B_n \}$

d.h die Attribute $B_1 \ldots B_n$ seien R_1 und R_2 gemeinsam und jeweils auf demselben Wertebereich definiert. Nur A hat noch zusätzliche Attribute. Betrachten wir $\{ A_1 \ldots A_m \}$ und $\{ B_1 \ldots B_n \}$ als zwei zusammengesetzte Attribute A und B . Die Division von R_1 und R_2

R_1 `divideby` R_2
(R_1/R_2)

ist eine Relation mit dem Kopf $\{ A \}$ und einem Körper, der aus allen Tupeln $\{ A{:}a \}$ besteht, wobei für *jedes* Tupel $\{ B{:}b \}$ in R_2 mit B-Wert b es ein Tupel $\{ A{:}a, B{:}b \}$ in R_1 gibt mit A-Wert a und B-Wert b.

Es gilt:

(R_1 `times` R_2) `divideby` R_2 ergibt wieder R_1.

aber:

(R_1 `divideby` R_2) `times` $R_2 \subseteq R_1$

Beispiel:

V:

SNR	PNR
S1	P1
S1	P2
S1	P3
S1	P4
S2	P2
S3	P1
S3	P2
S3	P4
S4	P3
S4	P4

W:

PNR
P2
P4

V `divideby` W:

SNR
S1
S3

Als Resultat erhalten wir die Lieferanten, die P2 *und* P4 geliefert haben.

Die Division ist nützlich bei Anfragen, die das Wort "alle" enthalten. Z.B. "Gib mir die Lieferanten, die *alle* Teile in W geliefert haben." Hier muss einfach durch die Tabelle mit allen Teilenummern dividiert werden.

Die Division kann durch andere Operationen ausgedrückt werden und ist daher keine primitive Operation.

V `divideby` W $\equiv$ V[SNR] `minus` ((V[SNR] `times` W) `minus` V)[SNR]

allgemein: R_1 `divideby` $R_2 \equiv R_1[A]$ `minus` ($(R_1[A]$ `times` $R_2)$ `minus` $R_1)[A]$, wobei A die Menge der Attribute von R_1 minus der Menge der Attribute von R_2 ist.

12.3 Beispiele

Im folgenden formulieren wir verschiedene Datenbankanfragen und geben dazu den algebraischen relationalen Ausdruck, dessen Resultat die Antwort auf die Anfrage ist. Manche Anfragen werden durch eine Erläuterung ergänzt.

Es gibt häufig mehrere Möglichkeiten der algebraischen Formulierung einer Anfrage, deren Unterschiede in einem späteren Kapitel diskutiert werden. Bei einem guten Optimierer sollte es keine Performanzunterschiede geben.

1. *Namen der Lieferanten, die Teil* P2 *liefern:*

   ((SP `join` S) `where` PNR = 'P2')[SNAME]

   Die Wirkung des Joins ist eine Expansion der Tabelle SP um die Lieferanteninformation. Diese Tabelle wird auf die P2-Teile eingeschränkt und dann auf die Lieferantennamen projeziert.

2. *Namen der Lieferanten, die mindestens ein rotes Teil liefern:*

   (((P `where` COLOR = 'Red') `join` SP)[SNR] `join` S)[SNAME]

   oder:

   ((P `join` SP `join` S[SNR, SNAME]) `where` COLOR = 'Red')[SNAME]

   Die erste Projektion bei S ist nötig, um zu verhindern, dass auch über CITY gejoined wird.

3. *Namen der Lieferanten, die alle Teile liefern:*

   (S `join` (SP `divideby` (P[PNR])))[SNAME]

   Wenn P leer ist, dann wird jeder Lieferant aufgeführt !

4. *Lieferantennummern der Lieferanten, die mindestens alle die Teile liefern, die Lieferant* S2 *liefert:*

   (SP[SNR, PNR] `divideby` (SP `where` SNR = 'S2')[PNR])[SNR]

5. *Alle Paare von Lieferantennummern der Lieferanten, die in derselben Stadt sitzen:*

   ```
   ( ( (S rename SNR as ERSTER) [ERSTER, CITY] join
         (S rename SNR as ZWEITER) [ZWEITER, CITY])
         where ERSTER < ZWEITER ) [ERSTER, ZWEITER]
   ```

   Es hätte genügt, nur in einer S-Relation SNR umzubenennen.

6. *Namen der Lieferanten, die Teil* P2 *nicht liefern:*

   ((S[SNR] `minus` (SP `where` PNR = 'P2')[SNR]) `join` S)[SNAME]

   (Vergleiche mit 1.)

12.4 Sinn der Algebra

Codds acht Operationen bilden keine **minimale** Menge von Operationen, da einige nicht **primitiv** sind, d.h. einige Operationen können durch andere Operationen ausgedrückt werden. Join, Schnitt und Division können durch die anderen Operationen ausgedrückt werden. Diese drei Operationen – insbesondere der Join – sind so nützlich, dass sie mit unterstützt werden.

Der eigentliche Sinn der Algebra ist es, **Ausdrücke** zu formulieren, die Relationen repräsentieren. Diese Relationen werden angewendet zur Definition von Daten in der DB, nämlich Daten,

- die aus der DB geholt werden sollen ***(retrieval)***,
- die manipuliert werden sollen ***(update)***,
- die als View sichtbar gemacht werden sollen ***(named virtual relation)***,
- die Schnappschüsse ***(snapshot)*** ergeben,
- die Bereiche zur Definition von Zugriffsrechten ***(security rules)*** sind,
- die Bereiche von Stabilitätsanforderungen ***(stability requirements)*** im Zusammenhang mit parallelen Aktivitäten sind,
- die zur Definition von DB-spezifischen Integritätsregeln ***(integrity rules)*** verwendet werden.

Unter Anwendung von **Transformationsregeln** ***(transformation rules)*** (Assoziativität und Kommutativität bestimmter Operationen) können Ausdrücke in effizientere Ausdrücke umgewandelt werden, so dass die Algebra eine Basis für den Optimierer darstellt.

Der Ausdruck

((SP `join` S) `where` PNR = 'P2')[SNAME]

ist vor Einsatz eines Optimierers weniger effizient als der äquivalente Ausdruck:

((SP `where` PNR = 'P2') `join` S)[SNAME]

Wegen ihrer fundamentalen Bedeutung dient die relationale Algebra als Maßstab für die Mächtigkeit von Anfragesprachen (z.B. SQL). Eine Anfragesprache ist **relational vollständig** ***(relational complete)***, wenn sie mindestens so mächtig ist, wie die relationale Algebra, d.h., wenn sie die Definition aller Relationen erlaubt, die auch mit der relationalen Algebra definiert werden können. SQL ist relational vollständig.

12.5 `extend` und `summarize`

Es wurden seit Codds acht Operatoren viele neue algebraische Operatoren vorgeschlagen. Zwei sehr nützliche werden im folgenden vorgestellt.

12.5.1 `extend`

Bisher bietet die Algebra keine Möglichkeiten, mit Skalaren Berechnungen durchzuführen. Z.B. ist es nicht möglich, einen View zu definieren, bei dem das WEIGHT als WEIGHT * 454 dargestellt ist (Gewicht in Gramm anstatt amerikanische Pfund).

Mit `extend` kann eine Relation um ein Attribut erweitert werden, dessen Werte Resultate eines **Berechnungsausdrucks** ***(scalar computational expression)*** sind. Z.B.

`extend` P `add` (WEIGHT * 454) `as` GMWT

Dieser Ausdruck ergibt eine Relation, deren Kopf um das Attribut GMWT erweitert ist.

Das neue Attribut kann natürlich in Selektionen und Projektionen verwendet werden.

Allgemein lautet die Syntax:

`extend` *term* `add` *scalar-expression* `as` *attribute*

`extend` R `add` Ausdruck `as` C

C darf kein Attribut von R sein. "Ausdruck" darf C nicht verwenden. `extend` verändert *term* nicht, sondern erzeugt eine nicht benannte Relation.

Beispiele:

1. `extend` S `add` 'Lieferant' `as` BEZEICHNUNG
2. `extend` (P `join` SP) `add` (WEIGHT * QTY) `as` SHIPWT
3. (`extend` S `add` CITY `as` SCITY)[SNR, SNAME, STATUS, SCITY]

   ist äquivalent zu

   S `rename` CITY `as` SCITY
4. `extend` S `add count` ((SP `rename` SNR `as` X) `where` X = SNR) `as` NP

   Das Resultat ist:

SNR	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

   Die Umbenennung von SNR in X ist nötig, weil das zweite SNR sich auf das S bezieht.

   Hier wurde die Aggregat-Funktion `count` verwendet, die – auf eine Relation angewendet – deren Cardinalität (Anzahl Tupel) zurückgibt. NP ist also die Anzahl der Teile, die ein Lieferant liefert.

   Andere Aggregat-Funktionen sind `sum, avg, max, min`
5. Multiples `extend`:
   `extend` P `add` City `as` PCITY, (WEIGHT * 454) `as` GMWT

12.5.2 summarize

Das `extend` bietet reihenweise (horizontale) Berechnungen an. `summarize` ermöglicht spaltenweise oder vertikale Berechnungen.

`summarize` SP `by` (PNR) `add sum` (QTY) `as` TOTQTY

ergibt eine Relation mit dem Kopf {PNR, TOTQTY} mit einem Tupel pro unterschiedlichem PNR-Wert in SP. PNR enthält den PNR-Wert und TOTQTY die Summe aller QTY, die zu dem PNR-Wert gehören.

Syntax:

`summarize` *term* `by` (*attribute-commalist*)
`add` *aggregate-expression* `as` *attribute*

`summarize` R `by` $(A_1, A_2 \ldots A_n)$ `add` Ausdruck `as` C

$A_1, A_2 \ldots A_n$ sind unterschiedliche Attribute von R. Das Resultat ist eine Relation mit dem Kopf $\{A_1, A_2 \ldots A_n, C\}$ und mit einem Körper, der aus allen Tupeln t besteht, so dass t ein Tupel der Projektion von R über $A_1, A_2 \ldots A_n$ erweitert um einen Wert für C ist. Der C-Wert ergibt sich durch Auswertung von "Ausdruck" über alle Tupel von R, die dieselben Werte für $A_1, A_2 \ldots A_n$ haben wie das Tupel t. "Ausdruck" darf C nicht verwenden. Die *attribute-commalist* darf C nicht enthalten.

Die Kardinalität des Resultats ist gleich der Kardinalität der Projektion von R über $A_1, A_2 \ldots A_n$.

`summarize` (P `join` SP) `by` (CITY) `add count as` NSP

ergibt :

CITY	NSP
London	5
Paris	6
Rome	1

Beispiele:

1. `summarize` SP `by` ( ) `add sum` (QTY) `as` GRANDTOTAL

   Enthält SP mindestens ein Tupel, dann haben alle Tupel von SP denselben Wert für "keine Attribute" und es gibt daher nur eine Gruppe, über die summiert wird. Das Resultat ist eine Relation mit einer Spalte und einer Reihe und enthält die Summe aller QTY-Werte.

   Wenn aber SP leer ist, dann gibt es auch keine Gruppe, über die summiert werden kann. Das Resultat ist eine leere Relation, und leider nicht eine Relation, die den Wert 0 enthält.

2. `summarize` SP `by` (SNR) `add count as` NP

   ergibt :

SNR	NP
S1	6
S2	2
S3	1
S4	3

S5 erscheint in dieser Tabelle nicht, weil S5 garnicht in SP vorkommt (Vergleiche Beispiel bei `extend`).

`count` ist äquivalent zu `sum` (1)

3. *Alle Städte mit mehr als fünf roten Teilen:*

   ((`summarize` (P `where` COLOR = 'Red')
   `by` (CITY) `add count as` N) `where` N > 5)[CITY]

   Alle Städte mit fünf oder weniger roten Teilen:

   ((`summarize` (P `where` COLOR = 'Red')
   `by` (CITY) `add count as` N) `where` N $\leq$ 5)[CITY]

   ergibt nicht das korrekte Resultat, da die Städte ohne rote Teile fehlen. Korrekt ist

   P[CITY] `minus`
   ((`summarize` (P `where` COLOR = 'Red')
   `by` (CITY) `add count as` N) `where` N > 5)[CITY]

4. Multiples `summarize`:

   (`summarize` SP `by` (PNR) `add sum` (QTY) `as` TOTQTY,
   `avg` (QTY) `as` AVGQTY

`summarize` ist nicht primitiv. Es kann mit `extend` simuliert werden.

12.6 Update-Operationen

Zur relationalen Algebra gehört auch der **Zuweisungsoperator *(assignment operator)***, den wir bisher noch nicht benötigt haben.

Syntax:

target `:=` *source*;

Linke und rechte Seite sind relationale Ausdrücke mit typ-kompatiblen Resultaten. Links steht gewöhnlich eine benannte Relation, typischerweise eine Basisrelation, deren Inhalt durch das Ergebnis des rechten Ausdrucks ersetzt wird.

Der Zuweisungsoperator ermöglicht, das Ergebnis eines algebraischen Ausdrucks festzuhalten und den Zustand der DB zu ändern. Allerdings – für kleine Änderungen an einer Tabelle – ist die Zuweisung ein unverhältnismäßig starkes Mittel. Als Beispiel wollen wir ein Tupel addieren:

S `:=` S `union` { {
⟨SNR:'S6'⟩,
⟨SNAME:'Baker'⟩,
⟨STATUS:50⟩,
⟨CITY:'Madrid'⟩ } };

Oder ein Tupel entfernen:

SP `:=` SP `minus` { {
⟨SNR:'S1'⟩,

⟨PNR:'P1'⟩,
⟨QTY:300⟩ } };

Man kann im Prinzip alle Update-Operationen auf diese Art durchführen. Aber `union` oder `minus` behandeln Fehlersituationen nicht erwartungsgemäß. Z.B. sollte der Versuch, ein Duplikat einzufügen oder ein nicht existentes Tupel zu löschen, zurückgewiesen werden.

Daher wird jedes DBMS explizite `insert`-, `delete`- und `update`-Operationen zur Verfügung stellen.

`insert`-Statement :

`insert` *source* `into` *target*;

source und *target* sind typ-kompatible Ausdrücke. *source* wird ausgewertet und alle Tupel der resultierenden Relation werden in *target* eingefügt.

`insert` (S `where` CITY = 'London') `into` TEMP;

Dabei ist TEMP irgendeine zu S typ-kompatible Relation.

`update`-Statement :

`update` *target assignment-commalist*;

wobei jede Zuweisung *assigment* von der Form

attribute `:=` *scalar-expression*

ist. *target* ist ein relationaler Ausdruck und jedes *attribute* ist ein Attribut der resultierenden Relation. Jedes Tupel der resultierenden Relation wird entsprechend den Zuweisungen verändert. Diese Änderungen wirken sich auf die entsprechenden Basisrelationen aus.

`update` P `where` COLOR = 'Red', CITY = 'Paris';

`delete`-Statement :

`delete` *target*;

Das *target* ist ein relationaler Ausdruck. Alle Tupel in der resultierenden Relation werden gelöscht. Dieses Löschen schlägt auf die entsprechenden Basisrelationen durch.

`delete` S `where` STATUS < 20;

Alle diese Operationen sind Mengenoperationen, d.h. sie wirken auf eine Menge von Tupeln, nicht auf einzelne Tupel. Folgendes Beispiel soll das illustrieren (Der Status von S1 und S4 muss immer gleich sein und wird hier bei beiden gleichermaßen verändert.):

`update` S `where` SNR = 'S1' `or` SNR = 'S4' STATUS `:=` STATUS + 5;

Prädikate sind das Kriterium für die Akzeptierbarkeit eines Updates. Die Prädikate für die Basisrelationen seien bekannt. Wie wirken diese sich auf abgeleitete Relationen aus? Dazu müssen einfache Regeln aufgestellt werden.

Seien R_1 und R_2 Relationen und PR_1 bzw. PR_2 Prädikate von R_1 bzw. R_2. Das Prädikat PR_3 für

R_3 `:=` R_1 `intersect` R_2

lautet dann

PR_3 `:=` PR_1 `and` PR_2

Für jedes Tupel t in R_3 muss also $PR_1(t)$ *und* $PR_2(t)$ wahr sein.

Für die anderen Operationen können entsprechende Regeln aufgestellt werden.

12.7 Relationale Vergleiche

Die ursprünglich definierte relationale Algebra enthielt keinen direkten Vergleich zweier Relationen. Dadurch wurden manche Ausdrücke sehr kompliziert.

Wir definieren daher eine neue Art von Bedingung:

condition `:=` *expression* Θ *expression*

Die relationalen Ausdrücke *expression* ergeben typ-kompatible Relationen, und Θ ist einer der folgenden Operatoren:

$=$: ist gleich

$\neq$: ist ungleich

$\leq$: ist Teilmenge von

$<$: ist echte Teilmenge von

$\geq$: ist Obermenge von

$>$: ist echte Obermenge von

Beispiele:

1. S[CITY] = P[CITY]
   Ist die Projektion der Lieferanten über CITY *gleich der Projektion der Teile über* CITY?
2. S[SNR] > SP[SNR]
   Gibt es Lieferanten, die keine Teile liefern?
3. S `where` ((SP `rename` SNR `as` X) `where` X = SNR)[PNR] = P[PNR]
   Dieser Ausdruck enthält alle Lieferanten, die alle Teile liefern.

Noch zwei Definitionen:

`is_empty` **(*expression*)** : Prüft ab, ob ein Ausdruck leer, d.h. eine leere Relation ist.

t `in` R : Prüft ab, ob ein Tupel t in der Relation R ist.

12.8 Syntax der relationalen Algebra

Wir wollen hier für die relationale Algebra ausschnittsweise eine Syntax als BNF-Grammatik darstellen:

expression
::= *monadic-expression* | *dyadic-expression*

monadic-expression
::= *renaming* | *restriction* | *projection*

renaming
::= *term* `rename` *rename-commalist*

rename-commalist
::= *rename* | *rename*, *rename-commalist*

rename
::= *attribute* `as` *attribute*

term
::= *relation* | (*expression*)

restriction
::= *term* `where` *condition*

projection
::= *term* | *term*
`[` *attribute-commalist* | *set-of-attributes-commalist* `]`

attribute-commalist
::= *attribute* | *attribute*, *attribute-commalist*

dyadic-expression
::= *projection dyadic-operation expression*

dyadic-operation
::= `union` | `intersect` | `minus` | `times` | `join` | `divideby`

relation und *attribute* sind Identifikatoren (eine terminale Kategorie bezüglich dieser Grammatik) und repräsentieren einen Relationsnamen bzw. einen Attributnamen.

12.9 Symbolische Notation

In vielen Veröffentlichungen wird eine symbolische Notation[30] verwendet, mit der sich oft etwas praktischer rechnen lässt.

Die Syntax als BNF-Grammatik lautet in *linearer* Darstellung:

expression
::= *monadic-expression* | *dyadic-expression*

monadic-expression
::= *renaming* | *selection* | *projection* | *groupby*

renaming
::= β[*rename-commalist*] *term*

rename-commalist
::= *rename* | *rename*, *rename-commalist*

rename
::= *attribute*$\leftarrow$*attribute*

term
::= *relation* | (*expression*)

selection
::= σ[*condition*] *term*

projection
::= π[*attribute-commalist* | *set-of-attributes-commalist*] *term*

groupby
::= γ[*function-commalist* ; *attribute-commalist* | *set-of-attributes-commalist*] *term*

orderby
::= ω[*attribute-commalist* | *set-of-attributes-commalist*] *term*

attribute-commalist
::= *attribute* | *attribute*, *attribute-commalist*

function-commalist
::= *function* | *function*, *function-commalist*

function
::= *function-name* (*attribute-commalist*)

function-name
::= `count` | `sum` | `avg` | `min` | `max` | `+` | `-` | `*` | `/`

dyadic-expression
::= *term dyadic-operation term*

dyadic-operation
::= $\cup$ | $\cap$ | $-$ | $\times$ | $\bowtie$ | $\bowtie$[`natural` | *condition-commalist* | *attribute-commalist*] | /

condition-commalist
::= *condition* | *condition*, *condition-commalist*

relation und *attribute* sind Identifikatoren (eine terminale Kategorie bezüglich dieser Grammatik) und repräsentieren einen Relationsnamen bzw. einen Attributnamen.

function-name: Die Operatoren "`+ - * /`" sind in prefix-Notation anzuwenden. Sie können auch verschachtelt werden.

Wenn bei $\bowtie$ eine *attribute-commalist* angegeben wird, so sind damit die "Join-Attribute" gemeint, die in beiden Termen denselben Namen haben müssen und auf Gleichheit der Werte geprüft werden. Wenn nichts angegeben wird, ist das Kreuzprodukt gemeint. (Es gibt allerdings auch Autoren, die darunter den natürlichen Join verstehen.)

In der *nicht-linearen (non-inline)* Notation werden die Ausdrücke in eckigen Klammern *tief* gestellt (mit oder ohne Klammern).

Beispiele:

β[SCITY←CITY] S

$\beta_{\text{SCITY←CITY}}$ S

σ[CITY='London'] S

$\sigma_{\text{CITY='London'}}$ S

π[COLOR, CITY] P

$\pi_{\text{COLOR, CITY}}$ P

S $\bowtie$ P

SP $\bowtie$[PNR] P

SP $\bowtie_{\text{PNR}}$ P

SP $\bowtie$[QTY>WEIGHT] P

SP $\bowtie_{\text{QTY>WEIGHT}}$ P

SP $\bowtie_{\text{natural}}$ P

γ[sum(QTY);SNR] SP

$\gamma_{\text{sum(QTY);SNR}}$ SP

ω[QTY] SP

ω_{QTY} SP

Übung: Schreiben Sie weitere in diesem Kapitel vorkommende Beispiele in dieser Notation auf.

Kapitel 13

Relationale Operatoren II: Relationales Kalkül

Relationale Algebra und **relationales Kalkül** ***(relational calculus)*** sind alternative Grundlagen des rationalen Modells. Die Algebra ist **prozedural**, indem sie die Operationen angibt, mit denen eine gewünschte Relation gebaut werden kann. Das Kalkül ist **deklarativ**, indem dort nur die Eigenschaften der gewünschten Relation beschrieben werden.

Die Anfrage

"Lieferantennummern und Städte von Lieferanten, die Teil P2 liefern"

lautet algebraisch etwa:

"Bilde den natürlichen Join von S und SP über SNR. Schränke dann das Resultat auf die Tupel für Teil P2 ein. Projeziere schließlich auf SNR und CITY."

Die Kalkülformulierung sieht etwa folgendermaßen aus:

"Bestimme SNR und CITY für diejenigen Lieferanten, für die es eine Lieferung in SP gibt mit dem gleichen SNR-Wert und dem PNR-Wert von P2."

Algebra und Kalkül sind äquivalent zueinander, indem es für jeden algebraischen Ausdruck einen entsprechenden Kalkül-Ausdruck gibt und umgekehrt. Das Kalkül ist aber etwas näher an der natürlichen Sprache.

Das relationale Kalkül beruht auf einem Teilgebiet der mathematischen Logik, dem Prädikatenkalkül oder der Prädikatenlogik und wurde zuerst von Kuhns, dann Codd (Data Sublanguage ALPHA) als Grundlage für DB-Sprachen eingeführt. ALPHA wurde nie implemetiert, aber eine sehr ähnliche Sprache QUEL wurde unter dem DBS INGRES implementiert.

Die **Tupelvariable** ***(tuple variable, range variable)*** ist ein fundamentales Konzept des Kalküls. Ihr Wertebereich sind Tupel einer Relation. "Die Tupelvariable T **durchläuft** ***(ranges over)*** die Relation R." Man spricht auch von **Tupelkalkül** ***(tuple calculus)***.

Die Anfrage "Bestimme Lieferantennummern für Lieferanten in London" lautet in QUEL:

```
RANGE OF SX IS S
```

```
RETRIEVE (SX.SNR) WHERE SX.CITY = "London"
```

Die Tupelvariable ist hier `SX` und läuft über die Relation `S`. Das `RETRIEVE`-Statement kann gelesen werden als: "Gib für jeden möglichen Wert der Variablen `SX` die `SNR`-Komponente genau dann, wenn die `CITY`-Komponente den Wert `London` hat."

Auf den **Wertebereichkalkül** ***(domain calculus, domain variable)*** gehen wir hier nicht ein (QBE).

Beweise für die oben genannten Behauptungen werden nicht geführt.

13.1 Tupelorientiertes Kalkül

13.1.1 BNF-Grammatik

Es folgt eine BNF-Grammatik für das relationale Kalkül.

range-variable-definition
::= `range of` *variable* `is` *range-item-commalist*;

range-item
::= *relation* `|` *expression*

expression
::= (*target-item-commalist*) [`where` *wff*]

target-item
::= *variable* `|` *variable.attribute* [`as` *attribute*]

wff
::= *condition*
`|` `not` *wff*
`|` *condition* `and` *wff*
`|` *condition* `or` *wff*
`|` `if` *condition* `then` *wff*
`|` `exists` *variable* (*wff*)
`|` `forall` *variable* (*wff*)
`|` (*wff*)

Die terminalen Kategorien *relation, variable* und *attribute* sind jeweils Identifikatoren. Die Kategorie *condition* repräsentiert eine einfachen skalaren Vergleich der Form

comparand Θ *comparand*

Dabei ist *comparand* entweder ein einfacher Skalar oder ein Attributwert einer Tupelvariablen (*attribute.variable*, qualifizierter Attributname).

Die Kategorie *wff* repräsentiert eine "well-formed-formula" (WFF), deren Wert ein Wahrheitswert *true* oder *false* ist. In einer WFF kommen Tupelvariable vor. D.h. wir können ein WFF f als Funktion einer oder mehrerer Tupelvariablen T betrachten ($f(T)$).

13.1.2 Tupelvariable

Eine Tupelvariable T wird durch ein Statement der Form

`range of` T `is` $X_1, X_2 \ldots X_n$;

definiert, wobei die $X_i (i = 1, 2 \ldots n)$ entweder Relationsnamen oder Kalkülausdrücke sind. Sie müssen typ-kompatibel sein. Die Tupelvariable T läuft über die Vereinigung der X_i.

Beispiele:

```
range of SX is S;
range of SPX is SP;
range of SY is (SX) where SX.CITY = 'London',
               (SX) where exists SPX (SPX.SNR = SX.SNR
                                      and SPX.PNR = 'P1');
```

Die Tupelvariable SY läuft über die Lieferantentupel, wo der Lieferant in London sitzt oder das Teil P1 liefert.

Für die weiteren Beispiele in diesem Kapitel gilt:

```
range of SX is S;
range of SY is S;
range of SZ is S;
range of PX is P;
range of PY is P;
range of PZ is P;
range of SPX is SP;
range of SPY is SP;
range of SPZ is SP;
```

13.1.3 Freie und gebundene Variable

Eine Tupelvariable in einem WFF kann frei (free) oder gebunden (bound) sein. Ein WFF kann wahr oder falsch sein, jenachdem wie die darin vorkommenden Tupelvariablen besetzt werden. Eine gebundene Tupelvariable kann in einem WFF nicht "willkürlich" besetzt werden, d.h. der Wahrheitswert einer WFF hängt nicht von einer gebundenen Tupelvariablen ab.

Tupelvariable werden durch die Quantoren `exists` und `forall` gebunden, ansonsten sind sie frei. Sie behalten den Zustand – gebunden oder frei – über logische und `if-then-` Verknüpfungen von WFFs bei.

Beispiel von WFFs:

1. Alle Vorkommen von SX, PX, SPX sind frei:

   SX.SNR = 'S1'
   SX.SNR = SPX.SNR
   SPX.PNR $\neq$ PX.PNR
   PX.WEIGHT $<$ 15 `or` PX.WEIGHT $>$ 25
   `not` (SX.CITY = 'London')

SX.SNR = SPX.SNR `and` SPX.PNR ≠ PX.PNR
`if` PX.COLOR = 'Red' `then` PX.CITY = 'London'

2. SPX und PX sind gebunden, SX ist frei:
   `exists` SPX (SPX.SNR = SX.SNR `and` SPX.PNR = 'P2')
   `forall` PX (PX.COLOR = 'Red')

13.1.4 Quantoren

Sei f eine WFF mit der freien Tupelvariablen T, die über die Tupel $\{t_1, t_2 \ldots t_n\}$ läuft, d.h.

`range of` T `is` $\{t_1, t_2 \ldots t_n\}$;

Der Existenzquantor `exists` (existential quantifier) ist durch folgende Äquivalenzen definiert:

`exists` T (f)
$\equiv$ `exists` T $(f(T))$
$\equiv$ *false* `or` $(f(t_1))$ `or` $\ldots$ `or` $(f(t_n))$

Der Allquantor `forall` (universal quantifier) ist durch folgende Äquivalenzen definiert:

`forall` T (f)
$\equiv$ `forall` T $(f(T))$
$\equiv$ *true* `and` $(f(t_1))$ `and` $\ldots$ `and` $(f(t_n))$
$\equiv$ `not exists` T (`not` f)

Die Definitionen der Quantoren zeigen, daß die Variable T insofern gebunden wird, als sie nach außen nicht mehr erscheint. (Die Definition enthält nicht mehr die Tupelvariable T, sondern nur die bestimmten Tupelwerte t_i.)

Eine WFF, bei der alle Tupelvariablen gebunden sind, heißt geschlossen (closed). Bei einer offenen WFF (open) gibt es mindestens eine freie Tupelvariable.

13.1.5 Ausdrücke

Ein *Ausdruck (expression)* hat die Form

(*target-item-commalist*) [`where` f]

Jede in f freie Tupelvariable muß in *target-item-commalist* erwähnt sein.

Beispiele:

(SX.SNR)
(SX.SNR) `where` SX.CITY = 'London'
(SX.SNR `as` SNO) `where` SX.CITY = 'London'
(SX.SNR, SX.CITY) `where exists` SPX (SPX.SNR = SX.SNR `and` SPX.PNR = 'P2')
(SX)
(SX.SNR, PX.PNR) `where` SX.CITY ≠ PX.CITY

13.2 Beispiele

Einige der folgenden Beispiele sind identisch mit Beispielen aus dem Algebra-Kapitel.

1. *Lieferantennummern der Lieferanten in Paris mit Status > 20:*

   (SX.SNR) `where` SX.CITY = 'Paris' `and` SX.STATUS > 20

2. *Alle Paare von Lieferantennummern der Lieferanten, die in derselben Stadt sitzen:*

   (SX.SNR `as` ERSTER, SY.SNR `as` ZWEITER)
   `where` SX.CITY = SY.CITY `and` SX.SNR < SY.SNR

3. *Namen der Lieferanten, die Teil* P2 *liefern:*

   (SX.SNAME) `where exists` SPX (SPX.SNR = SX.SNR
   `and` SPX.PNR = 'P2')

4. *Namen der Lieferanten, die mindestens ein rotes Teil liefern:*

   (SX.SNAME) `where exists` SPX (SPX.SNR = SX.SNR
   `and exists` PX (SPX.PNR = PX.PNR
   `and` PX.COLOR = 'Red'))

   oder (prenex normal form):

   (SX.SNAME) `where exists` SPX (`exists` PX (SPX.SNR = SX.SNR
   `and` SPX.PNR = PX.PNR `and` PX.COLOR = 'Red'))

   oder mit eine Klammer weniger

   (SX.SNAME) `where exists` SPX `exists` PX (SPX.SNR = SX.SNR
   `and` SPX.PNR = SPX.PNR `and` PX.COLOR = 'Red')

   Wir werden aber weiterhin diese Klammern angeben.

5. *Lieferantennamen der Lieferanten, die mindestens ein Teil liefern, das auch Lieferant* S2 *liefert:*

   (SX.SNAME) `where exists` SPX (`exists` SPY (SX.SNR = SPX.SNR
   `and` SPX.PNR = SPY.PNR `and` SPY.PNR = 'S2'))

6. *Namen der Lieferanten, die alle Teile liefern:*

   (SX.SNAME) `where forall` PX (`exists` SPX
   (SX.SNR = SPX.SNR `and` SPX.PNR = PX.PNR))

   Im Gegensatz zum entsprechenden algebraischen Ausdruck, ist dieser Ausdruck für alle Fälle richtig.

7. *Namen der Lieferanten, die Teil* P2 *nicht liefern:*

   (SX.SNAME) `where not exists` SPX
   (SX.SNR = SPX.SNR `and` SPX.PNR = 'P2')

8. *Lieferantennummern der Lieferanten, die mindestens alle die Teile liefern, die Lieferant* S2 *liefert:*

   (SX.SNR) `where forall` SPY (SPY.SNR ≠ 'S2' `or exists` SPZ
   (SPZ.SNR = SX.SNR `and` SPZ.PNR = SPY.PNR))

Da

`if` f `then` g

äquivalent zu

(`not` f) `or` g

ist, ist die obige Formulierung äquivalent zu:

(SX.SNR) `where forall` SPY (`if` SPY.SNR = 'S2' `then`
`exists` SPZ (SPZ.SNR = SX.SNR `and` SPZ.PNR = SPY.PNR))

9. *Teilenummern von Teilen, die entweder mehr als 16 Pfund wiegen oder von Lieferant* S2 *geliefert werden (oder beides):*

   `range of` PU `is` PX.PNR `where` PX.WEIGHT > 16,
   SPX.PNR `where` SPX.SNR = 'S2';
   (PU.PNR)

   oder

   (PX.PNR) `where` PX.WEIGHT > 16 `or exists` SPX
   (SPX.PNR = PX.PNR `and` SPX.SNR = 'S2')

   Die zweite Lösung verläßt sich auf die Tatsache, daß jede in SP vorkommende Teilenummer auch in P vorkommt.

13.3 Berechnungsmöglichkeiten

Um Berechnungen im Kalkül einzuführen müssen wir die Definitionen von *comparand* und *target-item* um eine neue Kategorie *scalar-expression* erweitern. Die Operanden eines solchen skalaren Ausdrucks können Literale, Attributreferenzen, Aggregatfunktionsaufrufe *aggregate-function-reference* sein.

Erklärungsbedürftig ist nur der Aggregatsfunktionsaufruf:

aggregate-function-reference
::= *aggregate-function* (*expression* [, *attribute*])

aggregate-function kann sein `count, sum, avg, min, max` und andere.

expression ergibt eine Relation. Für `count` ist das *attribute* bedeutungslos und muß weggelassen werden. Die anderen Funktionen benötigen genau ein Attribut. Nur wenn die Relation vom Grad 1 ist, kann dieses Attribut weggelassen werden.

Beispiele:

1. *Gib Teilenummern und Gewichte in Gramm für jedes Teil schwerer als 10000 Gramm an:*

   (PX.PNR, PX.WEIGHT * 454 `as` GMWT)
   `where` PX.WEIGHT * 454 > 10000

2. *Alle Lieferanten sollen die Bezeichnung 'Lieferant' bekommen:*

   (SX, 'Lieferant' `as` BEZ)

3. *Jede Lieferung soll mit Teiledaten und dem Liefergewicht ausgegeben werden:*

   (SPX.SNR, APX.QTY, PX, PX.WEIGHT * SPX.QTY `as` SHIPWT)
   `where` PX.PNR = SPX.PNR

4. *Gib Teilenummer und Gesamtlieferumfang für jedes Teil an:*

   (PX.PNR, `sum` (SPX `where` SPX.PNR = PX.PNR, QTY)
   `as` TOTQTY)

   Das wird auch richtig für Teile, die nicht in SP vorkommen, nämlich 0.

5. *Bestimme die Anzahl aller gelieferten Teile:*

   (`sum` (SPX, QTY) `as` GRANDTOTAL)

   Das wird auch richtig, wenn SP leer ist.

6. *Gib für jeden Lieferanten die Lieferantennummer und die Anzahl der gelieferten Teile an:*

   (SX.SNR, `count` (SPX `where` SPX.SNR = SX.SNR) `as` ANZTEILE)

   Auch für S5 bekommen wir ein Resultat (0).

7. *Gib Städte an, wo mehr als fünf rote Teile gelagert sind:*

   (PX.CITY) `where count`
   (PY `where` PY.CITY = PX.CITY `and` PY.COLOR = 'Red') > 5

8. *Gib Städte an, wo weniger als sechs rote Teile gelagert sind:*

   (PX.CITY) `where count`
   (PY `where` PY.CITY = PX.CITY `and` PY.COLOR = 'Red') $\leq$ 5

Kapitel 14

Die Anfragesprache SQL2

Die DSL SQL – ***structured query language*** – hat sich zur standard relationalen Sprache entwickelt, obwohl sie weit davon ist, eine treue Implementation des relationalen Modells zu sein. SQL wird von jedem relationalen Produkt unterstützt. Wenn man sich mit Datenbanken beschäftigt, muss man daher etwas über SQL wissen.

Das Dokument über den SQL-Standard ist etwa 600 Seiten lang. Es werden hier nur Grundzüge vermittelt. Wir beziehen uns auf "SQL/92" oder "SQL-92" oder "SQL2" oder die "International Standard Database Language SQL (1992)". Bei detailierten Syntaxfragen sei auf die Literatur verwiesen [38]. [14] [5] [31]

Kein SQL-Podukt implementiert den vollen Standard. Die meisten kommerziellen Produkte bieten eine "Obermenge einer Teilmenge" von SQL *("superset of a subset")* an.

Wir schließen SQL-Statements mit einem Semikolon ab, obwohl das im Standard nicht vorgeschrieben ist.

14.1 Datendefinition

In diesem Abschnitt behandeln wir den Datendefinitionsteil (DDL) von SQL. Die wichtigsten Statements sind:

```
CREATE DOMAIN
ALTER DOMAIN
DROP DOMAIN

CREATE TABLE
ALTER TABLE
DROP TABLE
```

View-Statements werden in einem späteren Kapitel besprochen.

14.1.1 Wertebereiche

SQL-Wertebereiche haben leider wenig zu tun mit den Wertebereichen des relationalen Modells. Im folgenden werden die Mängel herausgestellt:

- SQL-Wertebereiche sind eigentlich nur syntaktische Kürzel und stehen nicht für vom Benutzer definierte Datentypen. Wertebereiche müssen nicht verwendet werden. Man kann die Spalten direkt mit den eingebauten Typen definieren.
- Man kann Wertebereiche nicht mit schon definierten Wertebereichen definieren. Jeder Wertebereich setzt direkt auf den eingebauten Typen auf.
- Es gibt keine starke Typenbindung *(strong typing)*. Es gibt keine Wertebereichsüberprüfung. Bei Vergleichen beschränkt sich z.B. der Typencheck auf die eingebauten Typen. Man kann also Dollar mit Mark vergleichen.
- Man kann keine Operationen spezifisch für einen Wertebereich definieren.
- Es gibt kein Konzept von Ober- und Untertypen oder Vererbung.
- Es gibt keinen boolschen Wertebereich.

Die Syntax zur Erzeugung eines Wertebereichs lautet:

`CREATE DOMAIN` *domain data-type* [*default-definition*] [*domain-constraint-definition-list*]

Erklärung:

1. *data-type* kann sein:

   `CHARACTER [VARYING]` **(*n*)** :
   Mit der Spezifikation `VARYING` kann der Datentyp bis zu n Zeichen enthalten. Ohne `VARYING` werden immer n Speicherplätze angelegt, was zwar speicherintensiv, aber schneller ist. Die Abkürzung `CHAR` ist möglich.

   `BIT [VARYING]` **(*n*)**

   `INTEGER` :
   Binärer Integer mit Wortlänge. Die Abkürzung `INT` ist möglich.

   `SMALLINT` :
   Binärer Integer mit Halbwortlänge.

   `NUMERIC` **(*p, q*)** :
   Exakte Darstellung einer Gleitkommazahl mit p Dezimalstellen insgesamt und q Stellen hinter dem Komma. Genauigkeit genau p Stellen. `NUMERIC(2,1)` bedeutet der Bereich `-9.9` bis `+9.9` .

   `DECIMAL` **(*p, q*)** :
   Exakte Darstellung einer Gleitkommazahl mit p Dezimalstellen insgesamt und q Stellen hinter dem Komma. Genauigkeit mindestens p Stellen. Es können auch – implementationsabhängig – mehr Stellen gespeichert werden. `DECIMAL(2,1)` bedeutet mindestens der Bereich `-9.9` bis `+9.9` , kann aber auch z.B. von `-99.9` bis `+99.9` gehen.

`FLOAT (p)` :
Binäre (approximative) Darstellung einer Gleitkommazahl mit p Dezimalstellen.

`DATE`

`TIME`

`TIMESTAMP`

`INTERVAL`

2. Mit *default-definition*

   `DEFAULT` *defaultvalue*

   wird ein Defaultwert für den Wertebereich definiert. (Eine Spalte kann aber darüberhinaus einen eigenen Defaultwert haben.) *defaultvalue* kann ein Literal, eine eingebaute Funktion ohne Argument *(niladic function)* (z.B. `CURRENT_DATE`) oder `NULL` sein.

3. *domain-constraint-definition-list* ist eine Liste von *domain-constraint-definition* der Form

   `CONSTRAINT` *constraint-name* `CHECK` (*truth-valued-expression*)

   truth-valued-expression kann `VALUE` – den Wert, der überprüft werden soll – verwenden.

Beispiel:

```
CREATE DOMAIN COLOR CHAR (6)
   DEFAULT '???'
   CONSTRAINT VALID_COLORS
      CHECK (VALUE IN ('Red', 'Yellow', 'Blue', 'Green', '???'));
```

Mit `ALTER DOMAIN` kann die Defintion eines Wertebereichs verändert werden. Auf die teilweise komplexe Syntax kann hier nicht eingegangen werden.

Ein Wertebereich kann gelöscht werden mit

`DROP DOMAIN` *domain* `RESTRICT | CASCADE`

Bei Option `RESTRICT` schlägt die Operation fehl, wenn der zu löschende Wertebereich irgendwo referenziert wird.

Bei Option `CASCADE` wird die Löschoperation durchgeführt. Wenn der Wertebereich irgendwo referenziert wurde, dann wird u.U. stattdessen der zugrundeliegende eingebaute Datentyp verwendet. Auf die verschiedenen, teilweise komplizierten Reaktionsmöglichkeiten wird hier nicht eingegangen.

14.1.2 Basistabellen

SQL-Tabellen können Tupelduplikate haben. Basistabellen werden mit folgender Syntax definiert

`CREATE TABLE` *base-table* (*base-table-element-commalist*);

base-table-element
::= *column-definition* | *base-table-constraint-definition*

column-definition
::= *column representation* [`DEFAULT` *default*]

representation ist entweder ein eingebauter Datentyp oder ein vom Benutzer definierter Wertebereich.

`DEFAULT` *default* spezifiziert einen für die Spalte spezifischen Defaultwert, der den eventuell für den Wertebereich definierten Defaultwert überschreibt. Gibt es keinerlei Default, dann ist `NULL` der Defaultwert.

base-table-constraint-definition ist entweder die Definition eines Schlüssels oder eines Fremdschlüssels oder eine "check constraint"-Definition. Diese Randbedingungen kann man durch den Vorsatz

`CONSTRAINT` *constraint-name*

benennen.

base-table-constraint-definition
::= [`CONSTRAINT` *constraint-name*]
`UNIQUE` (*column-commalist*)
| `PRIMARY KEY` (*column-commalist*)
| `FOREIGN KEY` (*column-commalist*)
`REFERENCES` *base-table* [(*column-commalist*)]
`[ON DELETE NO ACTION | CASCADE | SET DEFAULT | SET NULL]`
`[ON UPDATE NO ACTION | CASCADE | SET DEFAULT | SET NULL]`
| `CHECK` (*conditional-expression*)

Eine Basistabelle kann höchstens einen Primärschlüssel, aber beliebig viele sonstige Schlüssel (`UNIQUE`) haben. Beim Primärschlüssel wird angenommen, dass jede zum Primärschlüssel gehörige Spalte implizit die Randbedingung `NOT NULL` hat.

Beim Fremdschlüssel ist die Option `NO ACTION` Default, d.h. eine Änderung oder Löschung des Schlüsselwerts wird zurückgewiesen. Die anderen Optionen sind selbsterklärend.

Mit `CHECK` wird eine Randbedingung definiert. Eine Datenmanipulation wird zurückgewiesen, wenn der Bedingungsausdruck für die Manipulation falsch wird. Die spezielle Randbegingung

`CHECK` (*column* `IS NOT NULL`)

kann ersetzt werden durch die Angabe des Defaultwertes `NOT NULL` bei der Spaltendefinition.

Unter der Voraussetzung, dass die Wertebereiche `SNR, PNR, QTY` und die Tabellen `S` und `P` mit den Primärschlüsseln `SNR` und `PNR` definiert seien, wird als Beispiel die Tabelle `SP` erzeugt.

```
CREATE TABLE SP
   (
   SNR SNR NOT NULL, PNR PNR NOT NULL, QTY QTY NOT NULL,
   PRIMARY KEY (SNR, PNR),
   FOREIGN KEY (SNR)
      REFERENCES S
      ON DELETE CASCADE
      ON UPDATE CASCADE,
```

```
FOREIGN KEY (PNR)
   REFERENCES P
   ON DELETE CASCADE
   ON UPDATE CASCADE,
CHECK (QTY > 0 AND QTY < 5001)
);
```

Bemerkung zu `SNR SNR`: Die zweite Nennung von `SNR` ist als Domainname zu verstehen.

Die Definition einer Basistabelle kann verändert werden (`ALTER TABLE`):

- Eine neue Spalte kann angefügt werden.
- Ein neuer Defaultwert kann für eine Spalte definiert werden.
- Ein Defaultwert für eine Spalte kann gelöscht werden.
- Eine Spalte kann gelöscht werden.
- Eine zusätzliche Randbedingung kann spezifiziert werden.
- Eine Randbedingung kann gelöscht werden.

Ein Beispiel für den ersten Fall ist:

```
ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1;
```

oder Änderung eines Spaltennamens und seines Typs:

```
ALTER TABLE S CHANGE CITY STADT CHAR(20);
```

Eine Basistabelle kann gelöscht werden durch

`DROP TABLE` *base-table* `RESTRICT | CASCADE`;

Wird die Tabelle von anderen Tabellen referenziert, dann wird der Löschversuch bei der Option `RESTRICT` zurückgewiesen. Bei der Option `CASCADE` werden alle referenzierenden Elemente in der DB auch gelöscht.

14.2 Informationsschema

Das Datenlexikon (oder der Datenkatalog) einer DB heißt in SQL **Katalog** ***(catalog)*** und enthält die komplette Beschreibung einer individuellen DB.

Ein **Schema** ***(schema)*** enthält die Beschreibung des Teils der DB, die einem speziellen Benutzer zugänglich ist.

Eine DB hat daher einen Katalog und beliebig viele Schemata. Es gibt aber immer ein Schema, nämlich das *Informationsschema* (`INFORMATION_SCHEMA`), das die ganze DB-Beschreibung zeigt und die normale Katalog-Funktion hat. Es enthält auch die Definitionen aller anderen Schemata (*definition schema*).

Das Informationsschema führt z.B. folgende Tabellen:

```
SCHEMATA
DOMAINS
TABLES
VIEWS
COLUMNS
TABLE_PRIVILEGES
COLUMN_PRIVILEGES
USAGE_PRIVILEGES
DOMAIN_CONSTRAINTS
TABLE_CONSTRAINTS
REFERENTIAL_CONSTRAINTS
CHECK_CONSTRAINTS
KEY_COLUMN_USAGE
ASSERTIONS
VIEW_TABLE_USAGE
VIEW_COLUMN_USAGE
CONSTRAINT_TABLE_USAGE
CONSTRAINT_COLUMN_USAGE
CONSTRAINT_DOMAIN_USAGE
```

Für MySQL sei in diesem Zusammenhang auf die beiden Anweisungen

```
show tables;
describe Tabellenname;
```

hingewiesen.

14.3 Aktualisierende Operationen

In diesem Abschnitt beschäftigen wir uns mit dem DML-Teil von SQL. Die wichtigsten Statements sind hier `INSERT`, `UPDATE`, `DELETE` und `SELECT`.

Die Syntax der aktualisierenden Operationen `INSERT`, `UPDATE` und `DELETE` soll an einigen selbsterklärenden Beispielen gezeigt werden.

14.3.1 `INSERT` von Zeilen

```
INSERT
   INTO  P (PNR, PNAME, COLOR, WEIGHT, CITY)
   VALUES   ('P8', 'Sprocket', 'Pink', 14, 'Nice'),
            ('P7', 'Cam', 'Red', 9, 'London'),
            ('P3', 'Cock', 'Blue', 21, 'Paris');
```

14.3.2 INSERT einer Relation

```
INSERT
   INTO  TEMP (SNR, CITY)
   SELECT   S.SNR, S.CITY
      FROM  S
      WHERE S.STATUS > 15;
```

14.3.3 UPDATE

In den beiden folgenden Beispielen werden mehrere Zeilen verändert.

```
UPDATE   P
   SET   COLOR = 'Yellow',
         WEIGHT = WEIGHT + 5
   WHERE CITY = 'Paris';
```

```
UPDATE   P
   SET   CITY =   (SELECT  S.CITY
                     FROM  S
                     WHERE S.SNR = 'S5'
                  )
   WHERE COLOR = 'Red';
```

14.3.4 DELETE

```
DELETE
   FROM  SP
   WHERE 'London' =  (SELECT  S.CITY
                        FROM  S
                        WHERE S.SNR = SP.SNR
                     );
```

Statements der Art

```
DELETE FROM SP;
```

sind sehr gefährlich, da sie den Inhalt einer ganzen Tabelle löschen.

14.4 Such-Operationen

In diesem Abschnitt werden wir `SELECT` behandeln, indem wir eine Reihe von Beispielen vorstellen. Eine formale Behandlung erfolgt in einem späteren Abschnitt.

1. *Farbe und Stadt für Teile nicht in Paris und Gewicht größer als 14:*

```
SELECT   P.COLOR, P.CITY
   FROM  P
   WHERE    P.CITY <> 'Paris'
      AND   P.WEIGHT > 14;
```

   Bemerkungen:

   - Skalare Vergleichsoperatoren sind: `=  <>  <  >  <=  >=`
   - Resultat ist eine Tabelle mit *vier* Zeilen. SQL eliminiert Duplikate nicht, wenn das nicht explizit angegeben wird wie hier:

```
SELECT DISTINCT   P.COLOR, P.CITY
   FROM  P
   WHERE    P.CITY <> 'Paris'
      AND   P.WEIGHT > 14;
```

   - In SQL sind nicht-qualifizierte Namen solange zugelassen, als es keine Mehrdeutigkeiten gibt. Das "`P.`" ist daher im oben genannten Beispiel überflüssig:

```
SELECT DISTINCT   COLOR, CITY
   FROM  P
   WHERE    CITY <> 'Paris'
      AND   WEIGHT > 14;
```

     Qualifizierung der Namen macht die Statements lesbarer. Allerdings müssen in der `ORDER BY`-Klausel nicht-qualifizierte Namen verwendet werden:

```
SELECT DISTINCT   P.COLOR, P.CITY
   FROM  P
   WHERE    P.CITY <> 'Paris'
      AND   P.WEIGHT > 14
   ORDER BY CITY DESC;
```

   - Die `ORDER BY`-Klausel hat die Form:

     `ORDER BY` *order-item-commalist*

     Ein *order-item* besteht aus einem nicht-qualifizierten Spaltennamen und einem optionalen `ASC` (*ascending*, aufsteigend (Default)) oder `DESC` (*descending*, absteigend).

2. *Alle Teilenummern und Gewichte in Gramm:*

```
SELECT   P.PNR, P.WEIGHT * 454 AS GMWT
   FROM  P;
```

Die Spezifikation `AS GMWT` sorgt dafür, dass eine sonst namenlose (!) Spalte des Resultats einen Namen bekommt.

3. *Alle Informationen über die Lieferanten:*

```
SELECT   *  -- Kommentar: Alle Spalten von S
   FROM  S;
```

oder

```
SELECT   S.*   -- Kommentar: Alle Spalten von S
   FROM  S;
```

oder

```
TABLE S;
```

   - Kommentare in SQL werden mit `--` eingeleitet und gelten bis zum Ende der Zeile.
   - Für embedded Anwendungen ist der Stern nicht zu empfehlen, da sich seine Bedeutung ändern kann.

4. *Alle Kombinationen von Lieferanten und Teilen, die in einer Stadt sind:*

```
SELECT DISTINCT
         S.SNR, S.SNAME, S.STATUS, S.CITY,
         P.PNR, P.PNAME, P.COLOR, P.WEIGHT
   FROM  S, P
   WHERE S.CITY = P.CITY;
```

oder

```
SELECT DISTINCT
         S.SNR, S.SNAME, S.STATUS, S.CITY,
         P.PNR, P.PNAME, P.COLOR, P.WEIGHT
   FROM  S JOIN P ON (S.CITY = P.CITY);
```

oder

```
SELECT DISTINCT *
   FROM  S JOIN P USING (CITY);
```

oder

```
SELECT DISTINCT *
   FROM  S NATURAL JOIN P;
```

   - In jedem Fall ergibt das den natürlichen Join der Tabellen `S` und `P`.
   - Die erste Formulierung repräsentiert die Auflösung des Joins in ein kartesisches Produkt (`FROM S, P`), eine Restriktion (`WHERE ...`) und eine Projektion (`SELECT ...`).

5. *Alle Städtepaare, wobei der Lieferant in der ersten Stadt das Teil in der zweiten Stadt liefert:*

```
SELECT DISTINCT   S.CITY AS SCITY, P.CITY AS PCITY
   FROM  S JOIN SP USING (SNR) JOIN P USING (PNR);
```

oder

```
SELECT DISTINCT   S.CITY AS SCITY, P.CITY AS PCITY
   FROM  S NATURAL JOIN SP JOIN P USING (PNR);
```

nicht aber

```
SELECT DISTINCT   S.CITY AS SCITY, P.CITY AS PCITY
   FROM  S NATURAL JOIN SP NATURAL JOIN P;
```

Denn hier wird auch `CITY` für den Join verwendet.

6. *Alle Paare von Lieferantennummern, die in derselben Stadt sitzen:*

```
SELECT   ERST.SNR AS ERSTER, ZWEIT.SNR AS ZWEITER
   FROM  S AS ERST, S AS ZWEIT
   WHERE ERST.CITY = ZWEIT.CITY
      AND ERST.SNR < ZWEIT.SNR;
```

   - Explizite Bereichsvariable müssen hier verwendet werden.
   - `ERSTER` und `ZWEITER` können nicht in der `WHERE`-Klausel verwendet werden.

7. *Anzahl der Lieferanten:*

```
SELECT   COUNT(*) AS N
   FROM  S;
```

   - Resultat ist eine Tabelle mit einer Zeile und einer Spalte.
   - Es gibt die üblichen Aggregat-Funktionen `COUNT`, `SUM`, `AVG`, `MAX` und `MIN`.
   - `DISTINCT` vor dem Argument der Funktion eliminiert alle Duplikate.
   - `NULL`-Werte im Argument werden immer eliminiert.
   - Wenn das Argument eine leere Menge ist, ist das Resultat immer `NULL`, nicht Null (0) außer bei `COUNT`, wo bei leeren Menge Null (0) resultiert!
   - `COUNT` mit dem Argument `*` (`COUNT(*)`) ist besonders: `DISTINCT` ist nicht erlaubt. `NULL`-Werte werden immer mitgezählt.

8. *Bestimme maximal und minimal gelieferte Menge für Teil* `P2`*:*

```
SELECT   MAX(SP.QTY) AS MAXQTY, MIN(SP.QTY) AS MINQTY
   FROM  SP
   WHERE SP.PNR = 'P2';
```

   Die `FROM`- und `WHERE`-Klauseln gehören effektiv zu den Funktionsargumenten, müssen aber außerhalb formuliert werden. Diese Syntaxeigenschaft von SQL hat negative Auswirkungen auf die Brauchbarkeit, Struktur und Orthogonalität (Unabhängigkeit der Sprachkonzepte) der Sprache.

   Eine Konsequenz davon ist, dass Aggregatfunktionen nicht verschachtelt werden können. "Bestimme Mittelwert der totalen Teile-Liefermengen" kann nicht als `AVG(SUM(QTY))` formuliert werden, sondern muss lauten:

```
SELECT   AVG(TOTQTY) AS AVGTOTQTY
   FROM
      (SELECT  SP.PNR, SUM(QTY) AS TOTQTY
         FROM  SP
         GROUP BY SP.PNR
      );
```

9. *Teilenummer und Gesamtliefermenge:*

```
SELECT   SP.PNR, SUM(SP.QTY) AS TOTQTY
   FROM  SP
```

```
   GROUP BY SP.PNR;
```

Dies ist ein Analogon zum `summarize`-Statement der relationalen Algebra. Es ist wichtig, dass die Ausdrücke in der `SELECT`-Klausel pro Gruppe einen einzigen Wert liefern.

Eine alternative Formulierung ist:

```
SELECT   P.PNR,   (SELECT SUM(QTY)
                     FROM  SP
                     WHERE SP.PNR = P.PNR
                  ) AS TOTQTY
   FROM  P;
```

Diese Formulierung ergibt auch Zeilen für Teile, die nicht geliefert werden. Dort steht als Wert allerdings der `NULL`-Wert, nicht Null (0) drin. Verschachteltes `SELECT` zur Repräsentation skalarer Werte ist seit SQL2 erlaubt.

10. *Teilenummern von Teilen, die von mehr als einem Lieferanten geliefert werden:*

```
SELECT   SP.PNR
   FROM  SP
   GROUP BY SP.PNR
   HAVING   COUNT(SP.SNR) > 1;
```

Die `HAVING`-Klausel spielt für Gruppen von Zeilen dieselbe Rolle wie die `WHERE`-Klausel für einzelne Zeilen und hat die Funktion, Gruppen zu eliminieren.

11. *Namen der Lieferanten, die Teil* `P2` *liefern:*

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE S.SNR IN
      (SELECT  SP.SNR
         FROM  SP
         WHERE SP.PNR = 'P2'
      );
```

Dieses Beispiel benutzt eine Zwischenanfrage *(subquery)* und die `IN`-Bedingung. Eine äquivalente Formulierung ist:

```
SELECT DISTINCT   S.SNAME
   FROM  S, SP
   WHERE S.SNR = SP.SNR
      AND   SP.PNR = 'P2';
```

oder

```
SELECT DISTINCT   S.SNAME
   FROM  S NATURAL JOIN SP
   WHERE PNR = 'P2';
```

12. *Namen von Lieferanten, die mindestens ein rotes Teil liefern:*

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE S.SNR IN
      (SELECT  SP.SNR
         FROM  SP
         WHERE SP.PNR IN
            (SELECT  P.PNR
```

```
                FROM  P
                WHERE P.COLOR = 'Red'
             )
      );
```

oder

```
SELECT DISTINCT   S.SNAME
   FROM S NATURAL JOIN SP JOIN P USING (PNR)
   WHERE COLOR = 'Red';
```

13. *Nummern der Lieferanten mit Status kleiner als dem Maximalstatus in der Tabelle* `S` *:*

```
SELECT   S.SNR
   FROM  S
   WHERE S.STATUS <
      (SELECT  MAX(S.STATUS)
         FROM  S
      );
```

Dieses Beispiel verwendet zwei implizite Bereichsvariable, die beide über die Tabelle `S` laufen.

14. *Namen der Lieferanten, die Teil* `P2` *liefern:*

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE EXISTS
      (SELECT  *
         FROM  SP
         WHERE SP.SNR = S.SNR
            AND   SP.PNR = 'P2'
      );
```

Der `EXISTS()`-Bedingungs-Ausdruck wird wahr, wenn der Tabellenausdruck in der Klammer eine nicht-leere Tabelle liefert.

15. *Namen von Lieferanten, die nicht Teil* `P2` *liefern:*

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE NOT EXISTS
      (SELECT  *
         FROM  SP
         WHERE SP.SNR = S.SNR
            AND   SP.PNR = 'P2'
      );
```

Alternativ:

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE S.SNR NOT IN
      (SELECT  SP.SNR
         FROM  SP
         WHERE SP.SNR = S.SNR
            AND   SP.PNR = 'P2'
```

```
);
```

16. *Namen von Lieferanten, die alle Teile liefern:*

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE NOT EXISTS
      (SELECT  *
         FROM  P
         WHERE NOT EXISTS
            (SELECT  *
               FROM  SP
               WHERE SP.SNR = S.SNR
                  AND   SP.PNR = P.PNR
            )
      );
```

SQL hat keinen `forall`-Quantor. Daher müssen alle derartigen Anfragen mit dem negierten Existenz-Quantor formuliert werden.

Eine alternative Formulierung für die obige Anfrage ist:

```
SELECT DISTINCT   S.SNAME
   FROM  S
   WHERE (SELECT  COUNT(SP.PNR)
            FROM  SP
            WHERE SP.SNR = S.SNR
         )
         =
         (SELECT  COUNT(P.PNR)
            FROM  P
         );
```

- Die zweite Formulierung beruht auf der Tatsache, dass jede in `SP` vorkommende Teilenummer auch in `P` vorkommt, beruht also auf dem Einhalten einer Integritätsbedingung.
- Die zweite Formulierung geht nur in SQL2.
- SQL unterstützt nicht den Vergleich von zwei Tabellen. Aber das ist es, was wir eigentlich hier tun wollen.

  ```
  SELECT DISTINCT   S.SNAME
     FROM  S
     WHERE (SELECT  SP.PNR
              FROM  SP
              WHERE SP.SNR = S.SNR
           )
           =
           (SELECT  P.PNR
              FROM  P
           );
  ```

  Stattdessen mussten wir den Umweg über den Vergleich von Kardinalitäten gehen.

17. *Nummern von Teilen, die entweder weniger als 16 Pfund wiegen oder von Lieferant* `S2` *geliefert werden:*

```
SELECT   P.PNR
   FROM  P
   WHERE P.WEIGHT < 16
UNION
SELECT   SP.PNR
   FROM  SP
   WHERE SP.SNR = 'S2';
```

Duplikate werden immer eliminiert bei den Operationen `UNION`, `INTERSECT` und `EXCEPT`. `EXCEPT` ist das SQL-Analogon zur Operation `minus`. Bei den Varianten `UNION ALL`, `INTERSECT ALL` und `EXCEPT ALL` werden Duplikate beibehalten. (Anstatt `EXCEPT` funktioniert bei manchen DBS `MINUS`.)

Wenn das DBS `EXCEPT` nicht anbietet, dann kann man das wahrscheinlich durch sein einen Konstrukt

```
SELECT A, B, C FROM ... WHERE (A, B, C) NOT IN (SELECT A, B, C ...) T;
```

ersetzen. Das `INTERSECT` müsste dann durch zwei solche Konstrukte ersetzt werden.

SQL ist eine sehr redundante Sprache, indem es oft viele verschiedene Möglichkeiten gibt, eine Anfrage zu formulieren. In den genannten Beispielen sind die wichtigsten, aber bei weitem nicht alle Möglichkeiten gezeigt worden.

Aufwärmübungen:

1. Lieferanten mit Status größer 15.
2. Lieferungen mit QTY kleiner 250.
3. Alle blauen Teile.
4. Alle Teile mit Grammgewicht größer 8000 Gramm.
5. Namen der Lieferanten, die Teil P5 liefern, mit Angabe der QTY.
6. Welche Lieferanten haben ein Teil in Paris geliefert?
7. Erstellen Sie einen View, der Lieferantenname, Teilname und QTY zeigt.
8. Zeigen Sie, ob Status und Gesamtliefermenge korreliert sind.
9. Welcher Lieferant liefert die meisten Teile P4?
10. Welcher Lieferant liefert die meisten Teile?
11. Welche Lieferanten liefern weniger Teile als der Lieferant S2?
12. Erhöhen Sie den Status der Lieferanten um neun, die ein rotes Teil geliefert haben.
13. Erhöhen Sie den Status der Lieferanten um sieben, die mindestens eine Lieferung haben, die über dem Durchschnitt liegt.
14. Lieferantenpaare (Namen, Status, Gesamtliefermengen) und Differenzen der Gesamtliefermengen und Stati.

14.5 Tabellen-Ausdrücke

Eine erschöpfende Behandlung von Ausdrücken ist hier nicht möglich. Stattdessen geben wir eine einigermaßen vollständige BNF-Grammatik und gehen ausführlich nur auf die `SELECT`-Klausel ein.

14.5.1 BNF-Grammatik

table-expression
::= *join-table-expression* | *nonjoin-table-expression*

join-table-expression
::= *table-reference* [`NATURAL`] [*join-type*] `JOIN`
table-reference [`ON` *conditional-expression*
| `USING` (*column-commalist*)]
| *table-reference* `CROSS JOIN` *table-reference*
| (*join-table-expression*)

join-type
::= `INNER`
| `LEFT` [`OUTER`]
| `RIGHT` [`OUTER`]
| `FULL` [`OUTER`]
| `UNION`

table-reference
::= *table* [[`AS`] *range-variable* [(*column-commalist*)]]
| (*table-expression*) [`AS`] *range-variable*
[(*column-commalist*)]
| *join-table-expression*

nonjoin-table-expression
::= *nonjoin-table-term*
| *table-expression* `UNION` [`ALL`] [`CORRESPONDING` [`BY`
(*column-commalist*)]] *table-term*
| *table-expression* `EXCEPT` [`ALL`] [`CORRESPONDING` [`BY`
(*column-commalist*)]] *table-term*

nonjoin-table-term
::= *nonjoin-table-primary*
| *table-term* `INTERSECT` [`ALL`] [`CORRESPONDING` [`BY`
(*column-commalist*)]] *table-primary*

table-term
::= *nonjoin-table-term* | *join-table-expression*

table-primary
::= *nonjoin-table-primary* | *join-table-expression*

nonjoin-table-primary
::= `TABLE` *table*
| *table-constructor*
| *select-expression*
| `(` *nonjoin-table-expression* `)`

table-constructor
::= `VALUES` *row-constructor-commalist*

row-constructor
::= *scalar-expression*
| `(` *scalar-expression-commalist* `)`
| `(` *table-expression* `)`

select-expression
::= `SELECT [ ALL | DISTINCT ]` *select-item-commalist*
`FROM` *table-reference-commalist*
`[ WHERE` *conditional-expression* `]`
`[ GROUP BY` *column-commalist* `]`
`[ HAVING` *conditional-expression* `]`

select-item
::= *scalar-expression* `[ [ AS ]` *column* `] | [` *range-variable* `. ] *`

14.5.2 SELECT

Der `SELECT`-Ausdruck besteht aus fünf Komponenten: `SELECT`-Klausel, `FROM`-Klausel, `WHERE`-Klausel, `GROUP BY`-Klausel und `HAVING`-Klausel, wobei die letzen drei Klauseln optional sind.

`SELECT`-Klausel

Die `SELECT`-Klausel hat die Form:

`SELECT [ ALL | DISTINCT ]` *select-item-commalist*

`ALL` ist Default. Die *select-item-commalist* darf nicht leer sein.

Wie nehmen an, dass die Auswertung aller anderen Klauseln eine Tabelle R_1 ergibt. Aus R_1 wird eine neue Tabelle R_2 mit soviel Spalten gebildet, als *select-item-commalist* Elemente enthält, indem die skalaren Ausdrücke der *select-item*s für jede Zeile von R_1 ausgewertet werden. Die ausgewerteten Skalarausdrücke bilden eine Zeile von R_2. Eine Spalte von R_2 bekommt als Name entweder das, was in der `AS`-Klausel steht, oder den Skalarausdruck selbst, falls dieser ein einfachen Spaltenname ist, oder bleibt namenlos.

`FROM`-Klausel

Die `FROM`-Klausel hat die Form:

`FROM` *table-reference-commalist*

Die *table-reference-commalist* darf nicht leer sein. Die `FROM`-Klausel wird ausgewertet, indem das kartesische Produkt aller Elemente der *table-reference-commalist* gebildet wird.

`NATURAL` und `UNION` können bei einem `JOIN` nicht gleichzeitig spezifiziert werden. Beide vertragen sich nicht mit der `ON` und der `USING` Klausel. `INNER` ist Default.

`WHERE`-Klausel

Die `WHERE`-Klausel hat die Form:

`WHERE` *conditional-expression*

Die Tabelle R sei das Resultat der Auswertung der `FROM`-Klausel. Die `WHERE`-Klausel wird ausgewertet, indem in R alle Zeilen gestrichen werden, bei denen der *conditional-expression* falsch wird.

`GROUP BY`-Klausel

Die `GROUP BY`-Klausel hat die Form:

`GROUP BY` *column-commalist*

column-commalist darf nicht leer sein. Die Tabelle R sei das Resultat der Auswertung der `FROM`-Klausel und einer eventuellen `WHERE`-Klausel. Jede in der `GROUP BY`-Klausel genannte Spalte muss eine Spalte von R sein. Konzeptionell ist das Resultat der Auswertung der `GROUP BY`-Klausel eine *gruppierte* Tabelle, die aus Zeilengruppen besteht. Innerhalb einer Zeilengruppe sind die Werte für die in *column-commalist* genannten Spalten gleich. Das ist natürlich keine echte relationale Tabelle.

Aber durch die `SELECT`-Klausel wird daraus wieder eine Relation, wenn die `SELECT`-Klausel folgende Einschränkung befolgt:

Jedes *select-item* muss genau *einen* Wert pro Gruppe haben. Es darf also in einem *select-item* keine Spalte vorkommen, die nicht in *column-commalist* erwähnt ist, es sei denn, sie kommt in einer Aggregat-Funktion (`COUNT, SUM, AVG, MAX, MIN`) vor. Das Vorkommen in einer Aggregat-Funktion hat den Effekt, dass die Werte einer Gruppe auf einen Wert – das Resultat der Auswertung der Aggregat-Funktion – reduziert werden.

`HAVING`-Klausel

Die `HAVING`-Klausel hat die Form:

`HAVING` *conditional-expression*

Die Tabelle G sei die aus der Auswertung einer `GROUP BY`-Klausel resultierende gruppierte Tabelle. (Falls es keine `GROUP BY`-Klausel gibt, ist G das Resultat der Auswertung der `FROM`-Klausel und der eventuellen `WHERE`-Klausel aufgefasst als eine Tabelle, die aus genau *einer* Gruppe besteht.)

Das Resultat der `HAVING`-Klausel ist eine gruppierte Tabelle, die durch Streichung aller Zeilen aus G entsteht, bei denen *conditional-expression* falsch wird. Skalar-Ausdrücke in der `HAVING`-Klausel müssen pro Gruppe genau ein Resultat liefern. D.h. die `HAVING`-Klausel streicht immer ganze Gruppen.

Die HAVING-Klausel ist redundant.

14.5.3 Beispiel

Anfrage: *Bestimme Teilenummer, Gewicht in Gramm, Farbe und maximale Liefermenge für alle Teile, die rot oder blau sind und wovon mindestens insgesamt 350 geliefert wurden, wobei Lieferungen mit weniger als 201 Teilen unberücksichtigt bleiben.*

```
SELECT   P.PNR,
         'Grammgewicht = ' AS TEXT1,
         P.WEIGHT * 454 AS GMWT,
         P.COLOR,
         'Max Liefermenge = ' AS TEXT2,
         MAX(SP.QTY) AS MQY
   FROM  P, SP
   WHERE P.PNR = SP.PNR
         AND (P.COLOR = 'Red' OR P.COLOR = 'Blue')
         AND SP.QTY > 200
   GROUP BY P.PNR, P.WEIGHT, P.COLOR
   HAVING   SUM(SP.QTY) >= 350;
```

Mit Ausnahme der SELECT-Klausel selbst werden die Klauseln des SELECT-Ausdrucks konzeptionell in der Reihenfolge ausgewertet, in der sie notiert sind. Zuletzt wird die SELECT-Klausel selbst ausgewertet.

Bei der GROUP BY-Klausel würde eigentlich die Angabe P.PNR genügen, da Gewicht und Farbe eines Teils eindeutig davon abhängen. SQL weiß aber davon nichts und würde eine Fehlermeldung liefern, weil Gewicht und Farbe in der SELECT-Klausel vorkommen.

Das Resultat lautet:

PNR	TEXT1	GMWT	COLOR	TEXT2	MQY
P1	Grammgewicht =	5448	Red	Max Liefermenge =	300
P5	Grammgewicht =	5448	Blue	Max Liefermenge =	400
P3	Grammgewicht =	7718	Blue	Max Liefermenge =	400

14.5.4 JOIN-Ausdrücke

A und B seien zwei Tabellen-Referenzen *(table-reference)*, z.B.

A:

AA	AB	XY
a	aa	x
a	ab	xy
a	NULL	x
b	NULL	y
c	cc	NULL

B:

BB	AB	XY
b	bb	x
b	ab	xy
b	NULL	x
b	NULL	z
c	bb	NULL

Dann können wir folgende JOIN-Ausdrücke bilden:

- `A CROSS JOIN B`
  ergibt das kartesische Produkt von `A` und `B` und ist äquvalent zu

  ```
  SELECT * FROM A, B
  ```

- `A NATURAL [INNER] JOIN B`
  Aus dem kartesischen Produkt von `A` und `B` bleiben nur die Zeilen übrig, wo Spalten mit gleichem Namen in `A` und `B` überall gleiche Werte haben. NULLs werden nicht berücksichtigt. Von den Spalten mit gleichem Namen wird nur eine Spalte behalten. `ON` oder `USING` kann bei `NATURAL` nicht verwendet werden. Resultat des Beispiels:

AA	BB	AB	XY
a	b	ab	xy

- `A [INNER] JOIN B USING (XY)`
  Aus dem kartesischen Produkt von `A` und `B` bleiben nur die Zeilen übrig, wo die im `USING` genannten Spalten mit gleichem Namen in `A` und `B` überall gleiche Werte haben. NULLs werden nicht berücksichtigt. Von den Spalten mit gleichem Namen wird nur eine Spalte behalten. `ON` oder `USING` muss spezifiziert werden. Resultat des Beispiels:

AA	AB	BB	AB	XY
a	aa	b	bb	x
a	aa	b	NULL	x
a	NULL	b	bb	x
a	NULL	b	NULL	x
a	ab	b	ab	xy

- `A [INNER] JOIN B ON (A.XY = B.XY)`
  Ist äquvalent zu dem oben genannten JOIN-Ausdruck mit `USING`, außer dass beide XY-Spalten behalten werden. Resultat des Beispiels:

AA	AB	XY	BB	AB	XY
a	aa	x	b	bb	x
a	aa	x	b	NULL	x
a	NULL	x	b	bb	x
a	NULL	x	b	NULL	x
a	ab	xy	b	ab	xy

- `A UNION JOIN B`
  Das Resultat enthält alle Spalten und alle Zeilen von `A` und `B`, wobei jeweils mit `NULL` aufgefüllt wird. Duplikate werden nicht eliminiert. Zeilen von `A` und `B` werden nicht wie im kartesischen Produkt kombiniert. `ON` oder `USING` kann bei `UNION` nicht verwendet werden. Resultat des Beispiels:

AA	AB	XY	BB	AB	XY
a	aa	x	NULL	NULL	NULL
a	ab	xy	NULL	NULL	NULL
a	NULL	x	NULL	NULL	NULL
b	NULL	y	NULL	NULL	NULL
c	cc	NULL	NULL	NULL	NULL
NULL	NULL	NULL	b	bb	x
NULL	NULL	NULL	b	ab	xy
NULL	NULL	NULL	b	NULL	x
NULL	NULL	NULL	b	NULL	z
NULL	NULL	NULL	c	bb	NULL

- `A LEFT [OUTER] JOIN B USING (XY)`
  Das Resultat ist ein Join, wobei alle Zeilen von `A` auf jeden Fall erhalten bleiben. Resultat des Beispiels:

AA	AB	BB	AB	XY
a	aa	b	bb	x
a	aa	b	NULL	x
a	NULL	b	bb	x
a	NULL	b	NULL	x
a	ab	b	ab	xy
b	NULL	NULL	NULL	y
c	cc	NULL	NULL	NULL

- `A RIGHT [OUTER] JOIN B USING (XY)`
  Das Resultat ist ein Join, wobei alle Zeilen von `B` auf jeden Fall erhalten bleiben. Resultat des Beispiels:

AA	AB	BB	AB	XY
a	aa	b	bb	x
a	aa	b	NULL	x
a	NULL	b	bb	x
a	NULL	b	NULL	x
a	ab	b	ab	xy
NULL	NULL	b	NULL	z
NULL	NULL	c	bb	NULL

- `A FULL [OUTER] JOIN B USING (XY)`
  Das Resultat ist ein Join, wobei alle Zeilen von `A` und `B` auf jeden Fall erhalten bleiben. NULLs werden beim Join nicht berücksichtigt. Resultat des Beispiels:

AA	AB	BB	AB	XY
a	aa	b	bb	x
a	aa	b	NULL	x
a	NULL	b	bb	x
a	NULL	b	NULL	x
a	ab	b	ab	xy
b	NULL	NULL	NULL	y
c	cc	NULL	NULL	NULL
NULL	NULL	b	NULL	z
NULL	NULL	c	bb	NULL

14.6 Bedingungsausdrücke

Zunächst eine BNF-Grammatik für Bedingungsausdrücke:

conditional-expression
::= *conditional-term*
| *conditional-expression* `OR` *conditional-term*

conditional-term
::= *conditional-factor*
| *conditional-term* `AND` *conditional-factor*

conditional-factor
::= `[NOT]` *conditional-primary*

simple-condition
::= *comparison-condition*
| *in-condition*
| *match-condition*
| *all-or-any-condition*
| *exists-condition*

comparison-condition
::= *row-constructor comparison-operator row-constructor*

comparison-operator
::= `=` | `<` | `<=` | `>` | `>=` | `<>`

in-condition
::= *row-constructor* `[NOT] IN (` *table-expression* `)`
| *scalar-expression* `[NOT] IN`
`(` *scalar-expression-commalist* `)`

match-condition
::= *row-constructor* `MATCH UNIQUE (` *table-expression* `)`

all-or-any-condition
::= *row-constructor*
comparison-operator `ALL (` *table-expression* `)`
| *row-constructor*
comparison-operator `ANY (` *table-expression* `)`

exists-condition
::= `EXISTS (` *table-expression* `)`

Auf die Komplexität der Bedingungsausdrücke, die durch Nullen entstehen, wurde hier verzichtet.

14.6.1 MATCH-Bedingung

Die MATCH-Bedingung bringen wir hier nur in einer Form. Wegen einer ausführlichen Diskussion verweisen wir auf die Literatur.

row-constructor MATCH UNIQUE (*table-expression*)

z_1 sei die Zeile, die *row-constructor* liefert, und R sei die Tabelle, die aus *table-expression* resultiert. Dann wird die MATCH-Bedingung wahr, wenn es genau *eine* Zeile z in R gibt mit $z_1 = z$.

Anfrage: *Bestimme alle Lieferanten, die genau eine Lieferung haben.*

```
SELECT   S.*
   FROM  S
   WHERE S.SNR MATCH UNIQUE (SELECT SP.SNR FROM SP);
```

14.6.2 *All-or-Any*-Bedingung

Die *All-or-Any*-Bedingung hat die Form

row-constructor comparison-operator qualifier (*table-expression*)

z_1 sei die Zeile, die *row-constructor* liefert, und R sei die Tabelle, die aus *table-expression* resultiert. Dann wird die *All-or-Any*-Bedingung wahr, wenn

z_1 *comparison-operator* z

für *alle* (ALL) bzw *eine* (ANY) Zeile z von R gilt. (Ist R leer, dann wird die ALL-Bedingung wahr, die ANY-Bedingung falsch.)

Anfrage: *Bestimme die Namen der Teile, deren Gewicht größer ist als das Gewicht jeden blauen Teils.*

```
SELECT DISTINCT   PX.PNAME
   FROM  P AS PX
   WHERE PX.WEIGHT > ALL ( SELECT   PY.WEIGHT
                           FROM  P AS PY
                           WHERE PY.COLOR = 'Blue' );
```

Resultat ist:

PNAME
Cog

14.7 Skalarausdrücke

Ein Tabellen-Ausdruck in runden Klammern kann als Skalarausdruck verwendet werden, wenn die resultierende Tabelle aus einer Zeile und einer Spalte besteht (SQL2).

Folgende Operatoren können zur Bildung von Ausdrücken verwendet werden:

`+, -, *, /`
`||` (Aneinanderhängen von Zeichenfolgen)

```
BIT_LENGTH
CASE
CAST
CHARACTER_LENGTH
CURRENT_USER
LOWER
OCTET_LENGTH
POSITION
SESSION_USER
SUBSTRING
SYSTEM_USER
TRIM
UPPER
USER
```

Wir geben hier nur ein Beispiel für den `CASE`-Operator:

```
CASE
   WHEN S.STATUS <  5 THEN 'Muss bald eliminiert werden'
   WHEN S.STATUS < 10 THEN 'Geht gerade noch'
   WHEN S.STATUS < 15 THEN 'Geht'
   WHEN S.STATUS < 20 THEN 'Mittelmaessig'
   WHEN S.STATUS < 25 THEN 'Akzeptabel'
   ELSE                    'In Ordnung'
END
```

14.8 Embedded SQL

SQL-Statements können interaktiv oder als Teil eines Applikationsprogramms gegeben werden. In letzterem Fall spricht man von ***embedded*** SQL, "eingebettet" in eine ***Host***-Sprache, in der das Applikationsprogramm geschrieben ist. Jedes interaktiv verwendbare SQL-Statement kann mit kleineren Anpassungen auch in einem Applikationsprogramm benutzt werden ***(dual mode principal)***. Das Umgekehrte ist nicht wahr.

Insbesondere können nicht nur DML-Statements, sondern auch DDL-Statements verwendet werden.

Embedded SQL bietet syntaktische Konstrukte, um

- SQL-Statements abzusetzen,
- Host-Sprachen-Variable zu verwenden,

- Anfrageresultate zu verarbeiten,
- Fehler zu behandeln.

Zunächst ein C++ Programmfragment als Beispiel:

```
EXEC SQL BEGIN DECLARE SECTION;
   char  SQLSTATE[5];
   char  pnr[2];
   int   weight;
EXEC SQL END DECLARE SECTION;

pnr[0] = 'P';
pnr[1] = '2';

EXEC SQL SELECT   P.WEIGHT
            INTO  :weight
            FROM  P
            WHERE P.PNR = :pnr;

if (SQLSTATE[0] == '0'
   && SQLSTATE[1] == '0'
   && SQLSTATE[2] == '0'
   && SQLSTATE[3] == '0'
   && SQLSTATE[4] == '0')
   {
   cout << "SELECT hat funktioniert.\n";
   }
else
   {
   cout << "SELECT hat nicht funktioniert.\n";
   }
```

Bemerkungen:

1. Embedded SQL-Statements beginnen mit `EXEC SQL` und werden in den meisten Sprachen durch Semikolon abgeschlossen.
2. Ausführbare SQL-Statements können überall dort auftreten, wo ausführbare Statements der Host-Sprache auftreten können. Statements, die das Wort `DECLARE` enthalten, und das `WHENEVER`-Statement sind nicht ausführbar.
3. SQL-Statements können Variable der Host-Sprache verwenden. Diese haben dann einen Doppelpunkt als Prefix. Sie dürfen keine Felder und Strukturen sein.
4. Die `INTO`-Klausel spezifiziert die Host-Sprachen-Variablen, die mit den aus der Datenbankanfrage ermittelten Werten belegt werden sollen. Die i-te Variable korrespondiert zur i-ten Spalte einer ermittelten Zeile.

5. Jede in SQL-Statements benutzte Host-Variable muss in einer `DECLARE SECTION` definiert werden.

6. Jedes SQL-Applikationsprogramm muss eine Host-Variable mit dem Namen `SQLSTATE` enthalten. Nach jeder Ausführung eines SQL-Statements enthält `SQLSTATE` einen Statuscode. Der Code '00000' bedeutet erfolgreiche Durchführung.

7. Host-Variable müssen einen Datentyp haben, der zum SQL-Datentyp kompatibel ist.

8. Host-Variable dürfen denselben Namen wie Tabellenspalten haben.

9. Die `WHENEVER`-Deklaration gibt es in den vier Formen:

   ```
   EXEC SQL WHENEVER NOT FOUND   GO TO label;
   EXEC SQL WHENEVER NOT FOUND   CONTINUE;
   EXEC SQL WHENEVER SQLERROR    GO TO label;
   EXEC SQL WHENEVER SQLERROR    CONTINUE;
   ```

   `NOT FOUND` bedeutet, dass keine Daten gefunden wurden. `SQLERROR` bedeutet, dass ein Fehler aufgetreten ist.

   Die Wirkung der Deklaration ist so, dass nach jedem ausführbaren SQL-Statement automatisch in Abhängigkeit vom Fehlerstatus die deklarierte Aktion durchgeführt wird.

   Eine erneute Deklaration für `NOT FOUND` bzw `SQLERROR` überschreibt die alte Deklaration.

14.8.1 Operationen ohne Kursor

Beispiele:

```
EXEC SQL SELECT   STATUS, CITY
            INTO  :rank, :city
            FROM  S
            WHERE SNR = :snr;
```

Das funktioniert, weil die resultierende Tabelle nur aus einer Zeile besteht.

```
EXEC SQL INSERT
            INTO  P (PNR, PNAME, COLOR, WEIGHT, CITY)
            VALUES   (:pnr, :pname, DEFAULT, :weight, DEFAULT);
```

```
EXEC SQL UPDATE   S
            SET   STATUS = STATUS + :mehr
            WHERE CITY = 'London';
```

```
EXEC SQL DELETE
            FROM  SP
            WHERE :city = (SELECT   CITY
```

```
                    FROM  S
                    WHERE S.SNR = SP.SNR);
```

14.8.2 Operationen mit Kursor

Das Problem beim `SELECT`-Statement oder generell Tabellen-Ausdrücken ist, dass das Resultat eine Tabelle mit beliebig vielen Zeilen ist. Dieses Problem wird durch die Deklaration von *Kursoren* behandelt.

Zunächst geben wir ein Programmfragment, an dem wir die Vorgehensweise mit Kursoren erläutern.

```
EXEC SQL DECLARE X CURSOR FOR
   SELECT   S.SNR, S.SNAME, S.STATUS
      FROM  S
      WHERE S.CITY = :city;

EXEC SQL OPEN X;
   while (SQLSTATE[1] != '2')
      {
      EXEC SQL FETCH X INTO :snr, :sname, :status;
      cout << "SNR = " << snr << " SNAME = " << sname;
      cout << " STATUS = " << status << endl;
      }
EXEC SQL CLOSE X;
```

Bemerkungen:

1. Zunächst wird mit `DECLARE X CURSOR FOR` ein Kursor `X` deklariert. Die allgemeine Form ist:

   `EXEC SQL DECLARE` *cursor* `[SCROLL] CURSOR FOR`
   table-expression
   `[ ORDER BY` *order-item-commalist* `]`

   Das ist nur eine Deklaration. Der Tabellen-Ausdruck wird nicht ausgewertet.

2. Das Statement

   `EXEC SQL OPEN` *cursor*;

   aktiviert den deklarierten Kursor mit der Wirkung, dass der Tabellenausdruck mit den aktuellen Werten von eventuellen Host-Variablen ausgewertet wird. Das Resultat der Auswertung sei die Tabelle T. Der Kursor wird auf die erste Zeile der Tabelle T gesetzt. Wenn es keine erste Zeile gibt, d.h. wenn T leer ist, dann ist `SQLSTATE[1]` gleich `'2'`.

3. Das Statement

   `EXEC SQL FETCH [[`*row-selector*`] FROM]` *cursor* `INTO` *host-variable-commalist*;

weist den Inhalt der laufenden bzw selektierten Zeile den Variablen in der *host-variable-commalist* zu und setzt den Kursor auf die nächste Zeile. Wenn es keine nächste Zeile gibt, dann wird `SQLSTATE[1]` auf '2' gesetzt.

row-selector
    `NEXT`
  | `PRIOR`
  | `FIRST`
  | `LAST`
  | `ABSOLUTE` *n*
  | `RELATIVE` *n*

Für alle Optionen – ausgenommen `NEXT` – muss bei der Kursordeklaration `SCROLL` angegeben werden. `NEXT` bedeutet Selektion der laufenden Zeile.

4. Das Statement

   `EXEC SQL CLOSE` *cursor*;

   deaktiviert wieder den Kursor. Er kann wieder geöffnet werden, wobei der Tabellenausdruck erneut ausgewertet wird.

5. Die `UPDATE`- und `DELETE`-Statements können sich auch auf einen Kursor beziehen (`CURRENT OF` *cursor*). Zum Beispiel:

```
EXEC SQL UPDATE   S
            SET   STATUS = STATUS + :mehr
            WHERE CURRENT OF X;
```

14.8.3 SQLJ

SQLJ wurde von verschiedenen Firmen (Oracle, IBM, Tandem, Sybase, JavaSoft) als sogenannte "high-level"-Schnittstelle entwickelt, um von Java-Programmen aus auf Datenbanken – ebenso wie in den vorigen Abschnitten behandelt – zugreifen zu können.

(Allerdings steht Java-Entwicklern von Haus aus die "low-level"-Programmierschnittstelle JDBC zur Verfügung.)

Mit SQLJ werden die SQL-Befehle mit der besonderen Kennzeichnung

```
#sql { <SQL-Befehl> };
```

wie zum Beispiel

```
#sql { DELETE * FROM SP };
```

in ein Java-Programm geschrieben. Die Datei hat die Erweiterung `.sqlj` und wird von einem Präprozessor in gewöhnlichen Java-Quellcode mit JDBC-Anweisungen umgewandelt.

Der Präprozessor wird gestartet mit

```
$ sqlj dateiname.sqlj
```

Die Syntax von SQLJ ist sehr ähnlich dem normalen embedded SQL. Kursoren werden allerdings über einen Iterator verwendet. Im folgenden wird das Vorgehen beispielhaft für eine Anfrage gezeigt, die mehrere Tupel als Resultat liefert. Die Tupel bestehen aus einer Zeichenkette, Integer und Datum.

```
#sql iterator ResultatTyp (String, int, Date);
   // ResultatTyp wird als Iterator definiert.

ResultatTyp resultat;
   // Variable resultat wird definiert.

#sql resultat = { SELECT LNAME, QTY, DATUM
                     FROM LIEFERUNG
                     WHERE QTY > :qtySchranke };
   // Definition oder Ausführung der Anfrage

while (!resultat.endFetch ())
   {
   #sql { FETCH :resultat INTO :lname, :qty, :datum };
   // Holt einzelne Tupel.
   // ---
   }
```

Vergleich SQLJ und JDBC:

- Vorteil von SQLJ: Verwendung von SQL, als ob es zur Sprache gehört.
- Nachteil von SQLJ: Zwingend zu verwendender Präprozessor, der schlecht lesbaren Java-Code erzeugt.
- Vorteil von SQLJ: Typ- und Syntax-Überprüfung zur Übersetzungszeit mit Option `-online` . Bei JDBC treten SQL-Syntax-Fehler erst zur Laufzeit auf.
- Nachteil von SQLJ: SQL-Befehle können nicht zur Laufzeit erzeugt werden.
- SQLJ und JDBC können gemischt werden.

Kapitel 15

Normalformen

Redundanz von Daten ist schwer zu verwalten. Redundanz bedeutet, dass dasselbe Datum mehrfach in der Datenbank gespeichert ist. Dies führt zu verschiedenen Problemen:

- Eine Aktualisierung der Daten muss an mehreren Stellen durchgeführt werden ***(duplication effort)***.
- Speicherplatz wird verschwendet.
- Immer lauert die Gefahr der Inkonsistenz der Daten.

Aus Effizienzgründen mag es manchmal sinnvoll sein, Daten mehrfach zu halten. Das muss aber streng überwacht werden ***(controlled redundancy)***.

Ein gutes Entwurfsprinzip heißt daher Vermeidung von Redundanz (***"one fact in one place"***). Das Thema dieses Kapitels – Normalformen, Normalisierung – ist im wesentlichen eine Formalisierung eines derartigen Entwurfsprinzips oder allgemein anerkannter Grundsätze des Datenbankentwurfs.

Die Forderung der Redundanzfreiheit an eine DB gilt unabhängig vom relationalen Modell.

William Kent hat das auf folgenden Nenner gebracht: ***Each attribute must represent a fact about the key, the whole key, and nothing but the key.*** Der "key" repräsentiert eindeutig eine Entität, ein Objekt der realen Welt. Ein Attribut soll daher nur über genau *eine* Entität etwas aussagen.

Wir ergänzen das Zitat von Kent noch um: ***And the key should not tell us anything about the object.*** Der Schlüssel sollte also **keine** Information über das Objekt enthalten. Er dient nur der Indentifizierung.

Wir werden hier sechs Normalformen besprechen: 1NF, 2NF, 3NF, BCNF, 4NF und 5NF. Diese Formen sind hierarchisch zu verstehen, d.h. wenn eine Relation in 4NF ist, dann ist sie auch in 1NF, 2NF, 3NF und BCNF. Wir sprechen von niederen und höheren Normalformen.

Die Normalisierungsprozedur überführt *eine* Relation in eine *Menge* von Relationen, die meistens – nicht immer – in der nächst höheren Normalform sind. Dieser Vorgang ist insofern reversibel,

als die ursprüngliche Relation aus den neuen Relationen wiederhergestellt werden kann. Es geht bei der Normalisierung keine Information verloren.

Die ursprüngliche Relation wird zerlegt. Der Zerlegungsoperator in der Normalisierungsprozedur ist die Projektion. Die Umkehrung ist ein Join. Reversibilität heißt also, dass die ursprüngliche Relation gleich dem Join ihrer Projektionen ist.

Ziel eines Datenbankentwurfs sollte 5NF sein. Aber es kann gute Gründe geben, warum man bei früheren Normalformen stehen bleibt. Die einzig harte Forderung an eine relationale DB ist 1NF.

Jede Relation hat **Bedeutung** oder **Semantik** oder noch genauer: Die Tupel der Relation machen ein Prädikat wahr. Je einfacher die Bedeutungen oder die Prädikate der Relationen einer Datenbank zu formulieren sind, desto besser ist das Design der Datenbank.

15.1 Funktionale Abhängigkeit

Normalisierung läuft auf eine Zerlegung von Relationen hinaus. Diese Zerlegung muss **verlustfrei** erfolgen ***(nonloss/lossless decomposition)***. Keine Information darf verloren gehen. Die Frage, ob eine Zerlegung verlustfrei ist, hängt eng mit dem Begriff der **funktionalen Abhängigkeit** ***(functional dependency/dependence) FD*** zusammen.

Beispiel:

Tabelle SSTC :

SNR	STATUS	CITY
S3	30	Paris
S5	30	Athens

Zerlegung a):

Tabelle SST :

SNR	STATUS
S3	30
S5	30

Tabelle SC :

SNR	CITY
S3	Paris
S5	Athens

Zerlegung b):

Tabelle SST :

SNR	STATUS
S3	30
S5	30

Tabelle STC :

STATUS	CITY
30	Paris
30	Athens

Im Fall a) wird durch die Zerlegung keine Information verloren. Bei Zerlegung b) gibt es einen Informationsverlust, bzw es entsteht andere Information. Das ist leicht durch Bildung der Joins zu zeigen:

SST $\bowtie_{SNR}$ SC = SSTC

SST $\bowtie_{STATUS}$ STC $\neq$ SSTC

Definition: Sei R eine Relation mit X und Y Teilmengen der Attribute von R. Y heißt funktional abhängig von X

symbolisch: $X \longrightarrow Y$

("X bestimmt Y funktional") genau dann, wenn für alle Werte von R zu jedem X-Wert in R genau ein Y-Wert in R gehört.
Eine solche Beziehung heißt **funktionale Abhängigkeit** oder kurz **FD**. X heißt **Determinante** oder **linke Seite**, Y **Abhängige** oder **rechte Seite** einer FD.

In der Tabelle SP ist {QTY} funktional abhängig von {SNR, PNR}:

{SNR, PNR} $\longrightarrow$ {QTY}

In der Tabelle SCPQ : gibt es folgende FDs:

SNR	CITY	PNR	QTY
S1	London	P1	100
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

{SNR, PNR} $\longrightarrow$ {QTY}
{SNR, PNR} $\longrightarrow$ {CITY}
{SNR, PNR} $\longrightarrow$ {CITY, QTY}
{SNR, PNR} $\longrightarrow$ {SNR}
{SNR, PNR} $\longrightarrow$ {SNR, PNR, CITY, QTY}
{SNR} $\longrightarrow$ {QTY}
{QTY} $\longrightarrow$ {SNR}
{SNR} $\longrightarrow$ {CITY}

Aber

{CITY} $\longrightarrow$ {SNR}
{PNR} $\longrightarrow$ {QTY}

sind *keine* funktionalen Abhängigkeiten in SCPQ. Bei einzelnen Attributen werden die Mengensymbole {} oft weggelassen:

SNR $\longrightarrow$ CITY

Die funktionalen Abhängigkeiten

{SNR} $\longrightarrow$ {QTY}
{QTY} $\longrightarrow$ {SNR}

sind für den momentanen Wert von SCPQ gültig. Aber wenn wir an die Bedeutung von QTY denken, dann wird sich das im Laufe der Zeit wahrscheinlich ändern. Wir haben die Definition der FD so formuliert, dass sie für alle Werte von SCPQ gültig sein soll. Daher zählen diese beiden Beziehungen *nicht* zu den FDs von SCPQ.

Die Forderung der Gültigkeit einer FD ergibt eine **Integritätsbedingung**.

Wenn X ein Schlüssel von R ist, dann sind alle Attributteilmengen von R funktional von X abhängig. Gibt es in R eine FD: $A \rightarrow B$ und ist A kein Schlüssel von R, dann ist das ein Hinweis darauf, dass R irgendeine Art von Redundanz enthält. Die FD: SNR $\rightarrow$ CITY in Relation SCPQ ist solch ein Fall.

Es gibt in einer Relation sehr viele FDs. Aber viele dieser FDs sind **Implikationen** anderer FDs oder sie sind **trivial**. FDs können von anderen FDs nach den Regeln oder "Axiomen" von Armstrong abgeleitet werden, die aus der Definition der FD folgen:

Inferenzregeln von Armstrong: Seien A, B und C Attributteilmengen einer Relation R. Dann gilt:

1. Reflexivität: $B \subseteq A \Longrightarrow A \longrightarrow B$ (triviale FD)
2. Augmentation: $A \longrightarrow B \Longrightarrow A \cup C \longrightarrow B \cup C$
3. Transitivität: $A \longrightarrow B \quad \wedge \quad B \longrightarrow C \Longrightarrow A \longrightarrow C$

Die Regeln sind insofern vollständig, als alle FDs, die von einem Satz von FDs ableitbar sind, mit diesen Regeln abgeleitet werden können.

Folgende Regeln folgen aus den Armstrongschen Regeln und erleichtern den Umgang mit FDs:

1. Selbstbestimmung: $A \longrightarrow A$ (triviale FD)
2. Zerlegung: $A \longrightarrow B \cup C \Longrightarrow A \longrightarrow B \quad \wedge \quad A \longrightarrow C$
3. Vereinigung: $A \longrightarrow B \quad \wedge \quad A \longrightarrow C \Longrightarrow A \longrightarrow B \cup C$
4. Komposition: $A \longrightarrow B \quad \wedge \quad C \longrightarrow D \Longrightarrow A \cup C \longrightarrow B \cup D$
5. **Allgemeines Unifikationstheorem** ***(general unification theorem)***: $A \longrightarrow B \quad \wedge \quad C \longrightarrow D \Longrightarrow A \cup (C - B) \longrightarrow B \cup D$

Beispielhaft beweisen wir das Unifikationstheorem: Aus

$$A \longrightarrow B \quad \wedge \quad C \longrightarrow D$$

folgt durch Augmentation mit $(C - B)$:

$$A \cup (C - B) \longrightarrow B \cup (C - B) \quad \wedge \quad C \longrightarrow D$$

und durch Augmentation mit B folgt:

$$A \cup (C - B) \longrightarrow B \cup (C - B) \quad \wedge \quad C \cup B \longrightarrow D \cup B$$

Wegen $B \cup (C - B) = B \cup C = C \cup B$ erhalten wir:

$$A \cup (C - B) \longrightarrow C \cup B \quad \wedge \quad C \cup B \longrightarrow D \cup B$$

Darauf wenden wir die Transitivitätsregel an und erhalten:

$$A \cup (C - B) \longrightarrow D \cup B$$

Definition: Eine FD: $A \longrightarrow B$ in R heißt **irreduzierbar** genau dann, wenn kein Attribut der linken Seite (d.h. von A) entfernt werden kann. Man sagt auch: B ist voll oder irreduzierbar funktional von A abhängig.

Definition: Die (abgeschlossene) **Hülle** ***(closure)*** F^+ einer Menge F von FDs in R ist die Menge aller FDs, die aus den FDs in F abgeleitet werden können.

Definition: Eine Menge F von FDs in R ist **irreduzierbar** oder **minimal** genau dann, wenn sie folgende Eigenschaften erfüllt:

1. Jede FD in F ist irreduzierbar.
2. Die rechte Seite jeder FD in F enthält nur ein Attribut.
3. Keine FD in F kann entfernt werden ohne die Hülle von F zu ändern.

In der Relation SSTC bilden die FDs

SNR $\longrightarrow$ STATUS
SNR $\longrightarrow$ CITY

eine irreduzierbare Menge von FDs. Es ist kein Zufall, dass der Join der Projektionen SSTC [SNR, STATUS] und SSTC [SNR, CITY] wieder SSTC ergibt. Denn es gilt das folgende Theorem:

Theorem von Heath: Sei R eine Relation auf den Attributen $A \cup B \cup C$, wobei A, B und C Mengen von Attributen mit jeweils leerem Durchschnitt sind.
Wenn R die FD: $A \longrightarrow B$ erfüllt, dann ist R gleich dem Join (über A) ihrer Projektionen auf $A \cup B$ und $A \cup C$.

Bemerkungen:

1. Ein Beispiel für das Theorem ist R = SSTC, A = {SNR}, B = {STATUS} und C = {CITY}. SSTC kann verlustfrei in die Projektionen auf {SNR, STATUS} und {SNR, CITY} zerlegt werden.
2. Es genügt tatsächlich nur die Gültigkeit von $A \longrightarrow B$ zu fordern, wie an folgendem modifizierten Beispiel zu sehen ist, bei dem $A \not\longrightarrow C$:

   Tabelle MSSTC :

SNR	STATUS	CITY
S3	30	Paris
S3	30	London
S5	30	Athens

   Zerlegung:

   Tabelle MSST :

SNR	STATUS
S3	30
S5	30

   Tabelle MSC :

SNR	CITY
S3	Paris
S3	London
S5	Athens

Die Normalisierungsprozedur eliminiert alle FDs, die *nicht* einen Schlüssel als Determinante haben.

FDs gehören zur Semantik einer DB. {SNR} $\longrightarrow$ {CITY} bedeutet ja, dass ein Lieferant nur in einer Stadt sitzen kann. Eine FD repräsentiert eine Integritätsbedingung, die vom DBMS eingehalten werden muss. Normalisierung ist eine Möglichkeit, dem DBMS eine solche Integritätsbedingung mitzuteilen.

15.2 Erste, zweite und dritte Normalform

15.2.1 Erste Normalform

Definition 1NF: Eine Relation R liegt in der **ersten Normalform** vor ("ist in 1NF") genau dann, wenn jeder Attributwert **atomar** ***(atomic, simple, indivisible)*** ist. (Der zugrundeliegende Wertebereich enthält nur skalare Werte.)

Diese Eigenschaft ist bereits in der Definition einer Relation enthalten. Das relationale Modell kennt keine "repeating groups". Es gilt das Entwurfsprinzip, dass alle Relationen einer DB in 1NF sein müssen.

Die Relation

PNR	CITY
P1, P4, P6	London
P2, P5	Paris
P3	Rome

ist nicht in 1NF. Die 1NF dieser Relation sieht folgendermaßen aus:

PNR	CITY
P1	London
P4	London
P6	London
P2	Paris
P5	Paris
P3	Rome

15.2.2 Zweite Normalform

Definition 2NF: Eine Relation R liegt in der **zweiten Normalform** vor ("ist in 2NF") genau dann, wenn sie in 1NF ist und jedes Nichtschlüsselattribut voll (irreduzierbar) funktional abhängig von einem Schlüssel

ist. Irreduzierbare FDs, bei denen die linke Seite ein Schlüssel

ist, heißen **2NF-konform**.

oder anders: Eine Relation R liegt **nicht** in der **zweiten Normalform** vor, wenn es ein Nichtschlüsselattribut gibt, das nur von einem Teil des Schlüssels abhängt.

Die Erweiterung der Relation SP um das Attribut STATUS ist nicht in 2NF, da es die FD: {SNR} $\longrightarrow$ {STATUS} gibt und {SNR} nur ein Teil des Schlüssels {SNR, PNR} ist.
Oder die Abhängigkeit FD: {SNR, PNR} $\longrightarrow$ {STATUS} ist auf die Abhängigkeit FD: {SNR} $\longrightarrow$ {STATUS} reduzierbar.

SNR	STATUS	PNR	QTY
S1	20	P1	300
S1	20	P2	200
S1	20	P3	400
S1	20	P4	200
S1	20	P5	100
S1	20	P6	100
S2	10	P1	300
S2	10	P2	400
S3	30	P2	200
S4	20	P2	200
S4	20	P4	300
S4	20	P5	400

Das hat folgende **Update-Anomalien** ***(update anomalies)*** zur Folge:

1. Insert: Den STATUS-Wert für einen neuen Lieferanten kann man erst eintragen, wenn es eine Lieferung des Lieferanten gibt.
2. Delete: Wenn eine Lieferung gelöscht wird (z.B. S3...), dann wird damit auch die von der Lieferung unabhängige STATUS-Information über S3 gelöscht.
3. Update: Wenn man den STATUS-Wert von S1 ändern will, dann muss man jedes Tupel mit S1 suchen und dort den Wert ändern.

Der tiefere Grund für diese Anomalien ist, dass wir versucht haben in dieser Relation Informationen über zwei Entitäten zu verwalten, nämlich über Lieferungen und Lieferanten.

Normalisierung 1NF $\rightarrow$ 2NF: Jede Relation R, die nicht in 2NF ist, wird folgendermaßen zerlegt:
Die Relation R[A, B, C] (auf den schnittfreien Attributmengen A, B, C) habe die irreduzierbare FD: $A \longrightarrow B$, wobei A echter Teil eines Schlüssels

ist und B ein Nichtschlüsselattribut (auch nicht partiell) ist, dann wird durch die Zerlegung
R1 = R[$\overline{A}$, C]
R2 = R[$\underline{A}$, B]
die FD: $A \longrightarrow B$ 2NF-konform. A ist Fremdschlüssel in R1.

15.2.3 Dritte Normalform

Definition 3NF: Eine Relation R liegt in der **dritten Normalform** vor ("ist in 3NF") genau dann, wenn sie in 1NF ist und jedes Nichtschlüsselattribut *nur* (oder *nicht transitiv* (Date)) voll (irreduzierbar) funktional abhängig von einem Schlüssel

ist. Funktionale Abhängigkeiten, die der dritten Normalform widersprechen, sind nicht **3NF-konform**.

3NF schließt 2NF ein.

Die folgende Relation SCST hat die FD: {CITY} $\longrightarrow$ {STATUS} und ist daher nicht in 3NF. Wegen der FD: {SNR} $\longrightarrow$ {CITY} ist STATUS auch transitiv vom Schlüssel SNR abhängig. Die Verwendung des Begriffes Transitivität ist hier nicht glücklich. Der wesentliche Punkt hier ist, dass es außer der Abhängigkeit vom Schlüssel auch noch eine andere Abhängigkeit gibt, nämlich die Abhängigkeit von einem Nichtschlüsselattribut, das auch nicht Teil eines Schlüssels ist. Die Transitivität ist etwas rein Semantisches, indem die abzubildende Welt eben nur die Abhängigkeiten {SNR} $\rightarrow$ {CITY} und {CITY} $\rightarrow$ {STATUS} hat.

Tabelle SCST:

SNR	CITY	STATUS
S1	London	30
S2	Paris	10
S3	Paris	10
S4	London	30
S5	Athens	30

Das hat folgende Update-Anomalien zur Folge:

1. Insert: Wenn es in Rom keinen Lieferanten gibt, können wir nicht die Tatsache verwalten, dass Rom den STATUS-Wert 50 hat.
2. Delete: Wenn wir den Lieferanten S5 löschen, dann verlieren wir die Information über Athen.
3. Update: Der STATUS-Wert für eine Stadt kommt öfters vor (Redundanz). Bei einem Update müssen wir die Relation nach allen Tupeln mit einer bestimmten Stadt absuchen.

Der tiefere Grund für diese Anomalien ist wieder, dass wir versucht haben, in dieser Relation Tatsachen über verschiedene Entitäten, nämlich Lieferanten und Städte zu verwalten.

Normalisierung 2NF $\rightarrow$ 3NF: Jede Relation R, die nicht in 3NF ist, wird folgendermaßen zerlegt:
Die Relation R[A, B, C] (auf den schnittfreien Attributmengen A, B, C) habe die irreduzierbare FD: $A \longrightarrow B$, wobei A *nicht* Schlüssel

ist, dann wird durch die Zerlegung
R1 = R[$\overline{A}$, C]
R2 = R[$\underline{A}$, B]
die FD: $A \longrightarrow B$ 3NF-konform. A ist Fremdschlüssel in R1.

Beispiel: $A = \{CITY\}, B = \{STATUS\}, C = \{SNR\}$

Die Normalisierungen 1NF $\rightarrow$ 3NF und 2NF $\rightarrow$ 3NF können zusammengefasst werden:

Normalisierung 1,2NF $\rightarrow$ 3NF: Jede Relation R, die nicht in 3NF ist, wird folgendermaßen zerlegt:
Die Relation R[A, B, C] (auf den schnittfreien Attributmengen A, B, C) habe die irreduzierbare FD: $A \longrightarrow B$, wobei A nicht ein Schlüssel

von R ist und B ein Nichtschlüsselattribut (auch nicht partiell) ist, dann wird durch die Zerlegung
R1 = R[$\overline{A}$, C]
R2 = R[$\underline{A}$, B]
die FD: $A \longrightarrow B$ 3NF-konform und 2NF-konform. A ist Fremdschlüssel in R1.

In unserem Beispiel führt das zur Zerlegung:

R1 = SCST[SNR, CITY] :

SNR	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

R2 = SCST[CITY, STATUS] :

CITY	STATUS
London	30
Paris	10
Athens	30

15.2.4 Unabhängige Projektionen

Die angegebene Normalisierungsprozedur hat verhindert, dass wir die Zerlegung

R1 = SCST[SNR, CITY] und R2' = SCST[SNR, STATUS]

gewählt haben. Auch diese Zerlegung ist verlustfrei. Aber sie hat folgende Nachteile:

1. Insert: Wir können eine CITY-STATUS-Beziehung nicht ohne Lieferanten eintragen.
2. Inter-relationale Integritätsregel: Die beiden Projektionen sind voneinander abhängig, insofern als das DBMS bei einem Update zur Erhaltung der FD: {CITY} → {STATUS} beide Projektionen berücksichtigen muss. Diese FD ist durch die Zerlegung eine **inter-relationale** Integritätsregel geworden. Eigentlich ist diese FD verlorengegangen.

Normalisierung und *Erhaltung von Abhängigkeiten*: Die Projektionen R1 und R2 einer Zerlegung von R sollten **unabhängig** sein im Sinne von Rissanen:

Definition Rissanen: R1 und R2 sind unabhängige Projektionen von R genau dann, wenn

1. jede FD in R eine logische Konsequenz der FDs in R1 und R2 ist und
2. die gemeinsamen Attribute von R1 und R2 einen Schlüssel für mindestens eine der beiden Projektionen bilden.

D.h. alle FDs von R sollten in den Zerlegungen R1 und R2 repräsentiert sein.

In unserem Beispiel kann man aus {SNR} → {CITY} (gültig in R1) und {SNR} → {STATUS} (gültig in R2') *nicht* {CITY} → {STATUS} (gültig in R) herleiten.

Aber mit Hilfe der Transitivitätsregel kann aus {SNR} → {CITY} (gültig in R1) und {CITY} → {STATUS} (gültig in R2) die Beziehung {CITY} → {STATUS} (gültig in R) hergeleitet werden.

Die Forderung bezüglich des Schlüssels erfüllen beide Zerlegungen.

Eine Relation, die nicht in unabhängige Relationen zerlegbar ist, heißt **atomar**. Das bedeutet aber nicht, dass man jede Relation in unabhängige Relationen zerlegen sollte. Beispiel ist die Relation P.

15.3 Boyce-Codd-Normalform – BCNF

Die dritte Normalform behandelt Relationen mit mehreren Schlüsseln nicht befriedigend, wenn solche Schlüssel zusammengesetzt sind und sich überlappen. Die Boyce-Codd-Normalform BCNF löst diese Probleme. Obwohl die meisten Beispiele dazu eher pathologisch sind, hat die BCNF den Vorteil, dass sie ohne Referenz auf 2NF und 3NF auskommt und diese Konzepte damit überflüssig macht.

Definition BCNF: Eine Relation ist in BCNF genau dann, wenn sie in 1NF ist und jede irreduzierbare Determinante (linke Seite) einer nicht trivialen FD ein Schlüssel ist (bzw wenn jede nicht-triviale irreduzierbare FD einen Schlüssel als Determinante hat).

Bemerkungen:

1. Weniger formal heißt das: Linke Seiten von FDs müssen Schlüssel sein.
2. Wir sagen auch: Eine Relation ist in BCNF genau dann, wenn jede irreduzierbare FD in R BCNF-*konform* ist, d.h. entweder trivial ist, oder die Determinante ein Schlüssel ist.
3. BCNF ist stärker als 3NF. D.h. wenn eine Relation in BCNF ist, ist sie auch in 3NF.
4. BCNF ist konzeptionell einfacher als 3NF.
5. Obwohl wir die Einführung der zweiten und dritten Normalform für überflüssig halten, sei hier noch mal zusammengestellt, wie die Normalformen sich unterscheiden. Die folgende Tabelle ist so zu verstehen: Die angegebene funktionale Abhängigkeit (FD) verletzt die links stehenden Normalformen.

FD:	A	$\longrightarrow$	B
2NF 3NF BCNF	echter Schlüsselteil	$\longrightarrow$	Nichtschlüssel
3NF BCNF	Nichtschlüssel oder echter Schlüsselteil	$\longrightarrow$	Nichtschlüssel
BCNF	Nichtschlüssel oder echter Schlüsselteil	$\longrightarrow$	beliebige Attributmenge

Dabei bedeutet "Nichtschlüssel" eine Attributmenge, die weder Schlüssel ist noch Schlüsselteile enthält. "Nichtschlüssel oder echter Schlüsselteil" bedeutet dann "beliebige Attributmenge, die nicht Schlüssel ist".

Wenn in der Relation S SNAME auch ein Schlüssel ist, dann ist die Relation

SSP:

SNR	SNAME	PNR	QTY
S1	Smith	P1	300
S1	Smith	P2	200
S1	Smith	P3	400
S1	Smith	P4	200
S1	Smith	P5	100
S1	Smith	P6	100
S2	Jones	P1	300
S2	Jones	P2	400
S3	Blake	P2	200
S4	Clark	P2	200
S4	Clark	P4	300
S4	Clark	P5	400

in 3NF, nicht aber in BCNF. 3NF hat sich nur um Nichtschlüsselattribute gekümmert. SNAME ist hier eine Komponente des Schlüssels {SNAME, PNR}. Daher verletzt die FD: {SNR} → {SNAME} nicht 3NF. Aber BCNF wird durch die Existenz der FD: {SNR} → {SNAME} verletzt. Es gibt in der Tat auch Update-Anomalien: Wenn ein Name geändert wird, muss man die Tabelle nach diesem Namen absuchen.

Normalisierung auf BCNF: Jede Relation R, die nicht in BCNF ist, wird in 1NF überführt und dann eventuell folgendermaßen zerlegt:
Die Relation R[A, B, C] (auf den schnittfreien Attributmengen A, B, C) habe die nicht BCNF-konforme FD: $A \longrightarrow B$, dann wird durch die Zerlegung
R1 = R[$\overline{A}$, C]
R2 = R[$\underline{A}$, B]
die FD: $A \longrightarrow B$ BCNF-konform. A ist Fremdschlüssel in R1.

15.3.1 Problematisches Beispiel 1

Als weiteres Beispiel betrachten wir die Relation SFD, die Studenten (ST), Fächer (FC) und Dozenten (DZ) verwaltet:

SFD:

ST	FC	DZ
Smith	Math	White
Smith	Phys	Green
Jones	Math	White
Jones	Phys	Brown

Die Bedeutung der Tabelle ist, dass Student ST von Dozent DZ im Fach FC unterrichtet wird. Es gibt folgende FDs:

{DZ} → {FC} (Ein Dozent unterrichtet genau ein Fach.)

{ST, FC} → {DZ} (Ein Student wird in einem Fach nur von einem Dozenten unterrichtet.)

Ein Fach kann von mehreren Dozenten unterrichtet werden ({FC} $\not\rightarrow$ {DZ}).

Wir haben hier überlappende Schlüssel {ST, FC} und {ST, DZ}. Die Relation ist in 3NF, nicht aber in BCNF. Die Tatsache, dass Jones Physik studiert, können wir nicht löschen, ohne die Tatsache, dass Brown Physik lehrt, auch zu löschen. Es liegt daran, dass das Attribut DZ eine Determinante, aber kein Schlüssel ist.

Zerlegung in die Projektionen

SD = SFD[ST, DZ]:

ST	DZ
Smith	White
Smith	Green
Jones	White
Jones	Brown

und

DF = SFD[DZ, FC]:

DZ	FC
White	Math
Green	Phys
Brown	Phys

beseitigt diese Anomalien. SD und DF sind beide in BCNF. Allerdings sind die Projektionen nicht unabhängig im Rissanen'schen Sinn: Denn die FD

{ST, FC} → {DZ}

kann nicht von der einzigen, übriggebliebenen FD

{DZ} → {FC}

unter Verwendung der Armstrong'schen Regeln abgeleitet werden. Die beiden Projektionen können nicht unabhängig verändert werden. Der Versuch, das Tupel {Smith, Brown} in SD einzufügen, müsste zurückgewiesen werden, weil Brown auch Physik lehrt und Smith aber schon bei Green Physik hat.

Die Zerlegung in BCNF-Komponenten *und* die Zerlegung in zwei unabhängige Komponenten ist leider nicht immer gleichzeitig möglich. SFD ist atomar, aber nicht in BCNF.

15.3.2 Problematisches Beispiel 2

Für die drei Entitäten Lager (L), Mitarbeiter (M) und Teil (T) sollen folgende Regeln gelten:

1. Jeder Mitarbeiter gehört zu genau einem Lager.
2. In jedem Lager gibt es für jedes Teil genau einen Mitarbeiter.
3. Ein Teil kann in mehr als einem Lager sein.

15.4 Vierte und fünfte Normalform

Funktionale Abhängigkeit (FD) von Schlüsseln und triviale FDs sind unproblematisch. Alle anderen funktionalen Abhängigkeiten führen zu Anomalien. Eine Relation durch Zerlegung in BCNF zu bringen, bedeutet Elimination dieser anderen FDs.

Außer FDs gibt es noch andere Abhängigkeiten:

- **Mehrwertige Abhängigkeit *(multivalued dependency)*** – MVD

- **Verbund-Abhängigkeit, Join-Abhängigkeit *(join dependency)*** – JD

Elimination von nicht trivialen MVDs führt auf die 4NF (vierte Normalform). Elimination von nicht trivialen JDs führt auf die 5NF oder PJ/NF (fünfte Normalform, *projection-join normalform).*

MVD ist eine Verallgemeinerung von FD, und JD eine Verallgemeinerung von MVD. D.h. eine FD ist auch eine MVD und eine MVD ist eine JD. Das wiederum bedeutet, dass eine Relation in 4NF auch in BCNF ist und eine Relation in 5NF auch in 4NF.

Mehrwertige Attribute einer Entität sind oft der Grund für eine MVD. Ansonsten kommen nicht triviale MVDs und JDs, die nicht FDs sind, in der Praxis selten vor. Es ist aber mindestens wichtig zu wissen, dass es so etwas gibt, um gegebenenfalls, d.h. wenn "Merkwürdigkeiten" bei der Datenmodellierung auftreten, in der richtigen Richtung suchen zu können.

15.4.1 Vierte Normalform

Als Beispiel betrachten wir eine Relation STPLKO, die Daten über die Mitglieder einer Studienplankommission enthält. Für jedes Mitglied MG werden seine Fachkenntnisse FK und die Unterkommissionen UK, in denen es mitarbeitet aufgeführt. Wir präsentieren die Daten zunächst einmal in einer Relation, die nicht einmal in 1NF ist.

STPLKO:

MG	FK	UK
Smith	Informatik, Mathematik	Grundstudium, Hauptstudium
Jones	Informatik, Software,	Grundstudium, Hauptstudium, Multimedia
Clark	Elektronik, Physik, Mathematik	Grundstudium, Technik

Diese Tabelle zeigt, dass jedes Kommissionsmitglied mehrere Fachkenntnisse hat, die es alle in jeder bei ihm genannten Unterkommission einbringt.

Wir bringen diese Relation jetzt in 1NF:

STPLKO:

MG	FK	UK
Smith	Informatik	Grundstudium
Smith	Informatik	Hauptstudium
Smith	Mathematik	Grundstudium
Smith	Mathematik	Hauptstudium
Jones	Informatik	Grundstudium
Jones	Informatik	Hauptstudium
Jones	Informatik	Multimedia
Jones	Software	Grundstudium
Jones	Software	Hauptstudium
Jones	Software	Multimedia
Clark	Elektronik	Grundstudium
Clark	Elektronik	Technik
Clark	Physik	Grundstudium
Clark	Physik	Technik
Clark	Mathematik	Grundstudium
Clark	Mathematik	Technik

Diese Relation ist schon in BCNF. Trotzdem enthält die normalisierte Form ziemlich viel Redundanz, die zu Update-Anomalien führt. Angenommen, das sei eine Tabelle für den Kommissionsleiter, der z.B Herrn Clark aus der UK Grundstudium in die UK Multimedia versetzen möchte, dann muss er drei Tupel verändern. Oder wenn festgestellt wird, dass Herr Jones auch BWL kann, dann müssen drei Tupel ergänzt werden.

Der Grund für dieses Verhalten ist, dass FK und UK völlig unabhängig voneinander sind. Das Redundanzproblem können wir lösen durch folgende Zerlegung:

MGFK:

MG	FK
Smith	Informatik
Smith	Mathematik
Jones	Informatik
Jones	Software
Clark	Elektronik
Clark	Physik
Clark	Mathematik

und MGUK:

MG	UK
Smith	Grundstudium
Smith	Hauptstudium
Jones	Grundstudium
Jones	Hauptstudium
Jones	Multimedia
Clark	Grundstudium
Clark	Technik

MGFK und MGUK sind beide in BCNF und 4NF und es gilt:

STPLKO = MGFK `join` MGUK

Die Theorie dazu wurde 1977 von Fagin entwickelt und beruht auf mehrwertigen Abhängigkeiten (MVD). Die vorige Zerlegung kann nicht auf der Basis von FDs gemacht werden, denn außer trivialen FDs gibt es in STPLKO keine FDs. Es gibt in STPLKO allerdings zwei MVDs:

$\{MG\} \twoheadrightarrow \{FK\}$

$\{MG\} \twoheadrightarrow \{UK\}$

D.h. alle Fachkenntnisse, die Smith hat, sind eindeutig von Smith abhängig, oder alle Unterkommissionen, in denen Smith ist, sind eindeutig von Smith abhängig. Oder noch anders ausgedrückt: Es genügt den Namen Smith zu kennen, und die Datenbank kann mir alle Fachkenntnisse und

alle Unterkommissionen von Smith nennen. FK und UK sind insofern unabhängig als es nicht möglich ist, von einer Teilnahme in einer Unterkommission oder einer Kombination von Unterkommissionen auf die Fachkenntnisse oder umgekehrt zu schließen.

Der Doppelpfeil steht für eine MVD:

$A \twoheadrightarrow B$ (A *multidetermines* B; B *is multidependent on* A)

Es wird nicht die einzelne Fachkenntnis eindeutig durch das Mitglied bestimmt, wie das bei einer FD der Fall wäre, sondern es wird die *Menge* der zu einem Mitglied gehörenden Fachkenntnisse eindeutig bestimmt. Die Interpretation der zweiten MVD ist analog.

Definition MVD: Seien R eine Relation und A, B und C beliebige Teilmengen der Menge der Attribute von R. Dann hängt B mehrwertig von A ab

$A \twoheadrightarrow B$

genau dann, wenn die Menge der B-Werte, die zu einem gegebenen Paar (A-Wert, C-Wert) in R gehören, nur vom A-Wert abhängt und nicht vom C-Wert abhängt, d.h. gleich der Menge der B-Werte ist, für die es ein Tupel mit dem vorgegebenen A-Wert gibt.

Bemerkung: Eine FD ist eine MVD, bei der die Menge der B-Werte genau ein Element hat.

Definition triviale MVD: Eine MVD: $A \twoheadrightarrow B$ ist trivial, wenn entweder $B \subseteq A$ oder die Relation nur aus den Attributen A und B besteht ($\mathrm{R} = \mathrm{R}[A \cup B]$).

Definition 4NF: Eine Relation R ist in 4NF genau dann, wenn aus der Existenz einer *nicht trivialen* MVD: $A \twoheadrightarrow B$ in R folgt, dass alle Attribute von R auch funktional von A abhängig sind. D.h., dass jede nicht triviale MVD eine BCNF-konforme FD ist.

Bemerkung: In 4NF sind die einzigen nicht trivialen Abhängigkeiten FDs mit Schlüsseln als Determinanten.

Satz: R sei eine Relation auf den schnittfreien Attributmengen A, B und C, dann folgt aus der MVD: $A \twoheadrightarrow B$ die MVD: $A \twoheadrightarrow C$. Man schreibt auch:

$A \twoheadrightarrow B|C$

Theorem von Fagin: R sei eine Relation auf den schnittfreien Attributmengen A, B und C. Analog zum Heath'schen Theorem ist dann R gleich dem Join seiner Projektionen auf $A \cup B$ und $A \cup C$ genau dann, wenn in R die MVD: $A \twoheadrightarrow B \mid C$ erfüllt ist.

Rissanen gilt analog: R sei eine Relation auf den schnittfreien Attributmengen A, B und C und erfüllen die MVDs

$A \twoheadrightarrow B$ und $B \twoheadrightarrow C$ (und dann auch $A \twoheadrightarrow C$).

Dann ist die Zerlegung in $\mathrm{R}[A \cup B]$ und $\mathrm{R}[B \cup C]$ der Zerlegung in $\mathrm{R}[A \cup B]$ und $\mathrm{R}[A \cup C]$ vorzuziehen (Erhaltung der Abhängigkeiten).

15.4.2 Fünfte Normalform

Bisher konnten wir eine Relation verlustfrei in zwei Projektionen zerlegen. Es gibt aber Relationen, die nicht verlustfrei in zwei Projektionen zerlegbar sind, aber in drei oder mindestens n Projektionen. Solche Relationen heißen **n-zerlegbar** ***(n-decomposable)***.

Als Beispiel betrachten wir die Relation:

STPLKO:

MG	FK	UK
Smith	Informatik	Grundstudium
Smith	Mathematik	Hauptstudium
Jones	Informatik	Hauptstudium
Smith	Informatik	Hauptstudium

Diese Relation hat die drei Projektionen:

MGFK:

MG	FK
Smith	Informatik
Smith	Mathematik
Jones	Informatik

MGUK:

MG	UK
Smith	Grundstudium
Smith	Hauptstudium
Jones	Hauptstudium

FKUK:

FK	UK
Informatik	Grundstudium
Mathematik	Hauptstudium
Informatik	Hauptstudium

MGFK `join` MGUK ergibt:

MG	FK	UK
Smith	Informatik	Grundstudium
Smith	Informatik	Hauptstudium
Smith	Mathematik	Grundstudium ←
Smith	Mathematik	Hauptstudium
Jones	Informatik	Hauptstudium

Das ist eine Relation, die das dritte Tupel mehr hat als STPLKO. Wenn wir allerdings diese Relation noch mit FKUK über die Attribute {FK, UK} joinen, erhalten wir STPLKO zurück. Also

STPLKO = MGFK `join` MGUK `join` FKUK

Es hängt übrigens nicht davon ab, welches Projektionenpaar wir für den ersten Join wählen.

Für das weitere Vorgehen schreiben wir das Beispiel etwas abstrakter auf:

SPJ:

SNR	PNR	JNR
S1	P1	J2
S1	P2	J1
S2	P1	J1
S1	P1	J1

Die drei Projektionen heißen SP, SJ und PJ.

Die Tatsache, dass SPJ gleich dem Join der drei Projektionen ist, ist äquivalent zu

Wenn	*das Paar*	*(s1, p1)*	*in SP erscheint*
und	*das Paar*	*(s1, j1)*	*in SJ erscheint*
und	*das Paar*	*(p1, j1)*	*in PJ erscheint,*
dann	*muss das Tripel*	*(s1, p1, j1)*	*in SPJ erscheinen.*

Die Umkehrung dieses Statements ist auch wahr. Ferner gilt: Wenn das Tripel (s1, p1, x) in SPJ erscheint, dann erscheint das Paar (s1,p1) in SP. Entsprechendes gilt für SJ und PJ. Daher können wir das Statement folgendermaßen umschreiben:

Wenn	*(s1, p1, j2), (s1, p2, j1) und (s2, p1, j1) in SPJ erscheinen,*
dann	*erscheint auch (s1, p1, j1) in SPJ.*

Wenn das für alle Werte von SPJ gelten soll, dann stellt das eine Integritätsbedingung für SPJ oder *Abhängigkeit gewisser Tupel von anderen* dar. Sie ist äquivalent zu dem Statement, dass SPJ 3-zerlegbar ist, d.h. gleich dem Join gewisser Projektionen, und heißt daher **Verbund-Abhängigkeit** oder verständlicher **Join-Abhängigkeit *(join dependency) JD***. Man beachte die zyklische Natur der Bedingung (s1 ist mit p1, p1 mit j1 verknüpft). *Eine Relation ist n-zerlegbar genau dann, wenn sie solch eine zyklische Bedingung erfüllt.*

Diese Abhängigkeit bedeutet in unserem Beispiel, dass wenn Smith die Fachkenntnis Informatik hat und Informatik irgendwo im Hauptstudium eingebracht wird und Smith in der UK Hauptstudium ist, dann muss Smith auch die Informatik ins Hauptstudium einbringen.

Definition der Join-Abhängigkeit JD: Seien R eine Relation und $A, B \cdots Z$ beliebige Teilmengen der Attribute von R. Dann erfüllt R eine Join-Abhängigkeit

JD*$(A, B \cdots Z)$

genau dann, wenn R gleich dem Join seiner Projektionen auf $A, B \cdots Z$ ist.

Zum Beispiel erfüllt SPJ die JD*({S, P}, {S, J}, {P, J}).

Fagins Theorem über MVDs kann folgendermaßen umformuliert werden:

Theorem von Fagin: R sei eine Relation auf den schnittfreien Attributmengen A, B und C. R erfüllt die JD*$(\{A, B\}, \{A, C\})$ genau dann, wenn in R die MVD: $A \twoheadrightarrow B \mid C$ erfüllt ist.

Da dieses Theorem als Definition für mehrwertige Abhängigkeit verstanden werden kann, folgt, dass MVDs Spezialfälle von JDs sind. JDs sind die *allgemeinste* From der Abhängigkeit, solange wir uns auf Abhängigkeiten beschränken, die auf der Zerlegung von Relationen durch Projektionen und Zusammensetzung von Relationen mit Joins beruhen. Erlauben wir andere Operatoren, dann kommen andere Abhängigkeiten ins Spiel.

Es ist in der Tat wünschenswert, bei vorhandener JD eine Zerlegung durchzuführen und das solange zu tun, bis keine JD vorhanden ist, d.h. jede Relation in 5NF ist.

Satz über Implikation: Eine JD*$(A, B \cdots Z)$ wird von Schlüsseln *impliziert*, wenn jede Attributmenge $A, B \cdots Z$ einen Schlüssel enthält. Diese Implikation folgt aus dem Theorem von Heath. Eine allgemeinere Formulierung der Implikation führt hier zu weit.

Definition 5NF oder PJ/NF: Eine Relation R ist in 5NF genau dann, wenn jede JD in R von Schlüsseln in R impliziert wird.

Bemerkungen:

1. 5NF ist immer erreichbar.
2. PJ/NF steht für projection-join normal form.
3. Eine Relation in 5NF ist automatisch in 4NF, weil MVDs spezielle JDs sind.
4. SPJ ist nicht in 5NF, aber die Zerlegungen SP, SJ, PJ sind es.

15.5 Zusammenfassung der Normalisierung

Eine Relation R ist in Normalform, wenn sie in 1NF ist und wenn es kein Tupel in R gibt, das von anderen Tupeln in R logisch abhängt.

Eine Datenbank DB ist in Normalform, wenn jede Relation in 1NF ist und wenn es kein Tupel in der DB gibt, das von anderen Tupeln in DB logisch abhängt.

Gegeben sei eine Relation R in 1NF und eine Liste von Abhängigkeiten (FDs, MVDs, JDs), dann wird R durch Normalisierung systematisch reduziert zu einer Ansammlung von kleinen Relationen, die äquivalent zu R ist, aber eine wünschenswertere – anomaliefreie – Form hat.

Folgende Schritte sind durchzuführen:

1. Bilde Projektionen der Original-Relation, bis alle FDs eliminiert sind, deren Determinante *nicht* ein Schlüssel ist. Alle Relationen sind dann in BCNF.
2. Bilde weiter Projektionen, bis alle MVDs eliminiert sind, die *nicht* FDs sind. Alle Relationen sind dann in 4NF.
3. Bilde weiter Projektionen, bis alle JDs eliminiert sind, die *nicht* von Schlüsseln impliziert werden. Alle Relationen sind dann in 5NF.

Bemerkungen:

1. Die Projektionen müssen verlustfrei sein und nach Möglichkeit Abhängigkeiten bewahren.
2. Folgender Parallelismus gibt äquivalente Definitionen der Normalformen:
   - Eine Relation ist in BCNF genau dann, wenn jede FD von einem Schlüssel impliziert wird.
   - Eine Relation ist in 4NF genau dann, wenn jede MVD von einem Schlüssel impliziert wird.
   - Eine Relation ist in 5NF genau dann, wenn jede JD von einem Schlüssel impliziert wird.
3. Ziele der Normalisierung sind:
   - Elimination von Redundanz.
   - Vermeidung von Update-Anomalien.

- Erstellung eines Datenbankdesigns, das eine vernünftige Repräsentation der realen Welt ist und intuitiv leicht zu verstehen ist.
- Erleichterung der Einhaltung von Integritätsbedingungen.

4. Es kann gute Gründe geben den Normalisierungsprozess an einem Punkt abzubrechen.
5. Abhängigkeiten sind semantischer Natur. Die *Bedeutung* der Daten spielt eine Rolle. Relationale Algebra, Kalkül und Sprachen wie SQL kennen nur Daten*werte*. Das hat zur Folge, dass solche Formalismen die Normalisierung *nicht* fordern.
6. Gute Top-Down Entwurfsmethoden erzeugen im allgemeinen weitgehend normalisierte Entwürfe.
7. Ein etwas extremer Entwurf ist die sogenannte **strenge Zerlegung** ***(strict decomposition)***, wobei man ausgehend von einer **Universalrelation** ***(universal relation)***, die alle Attribute der gesamten Datenbank enthält, immer weiter zerlegt, bis alle Relationen in der gewünschten Normalform sind.

15.6 Andere Normalformen

Die Forschung über Normalisierung und ähnliche Themen geht weiter und ist heutzutage unter dem Namen *Abhängigkeitstheorie (dependency theory)* bekannt. Hier wird noch ein Beispiel vorgestellt:

Definition Domain-Key Normal Form (DK/NF) (Fagin): Eine Relation R ist in DK/NF genau dann, wenn jede Abhängigkeit in R eine logische Konsequenz einer **Wertebereichsabhängigkeit** ***(domain constraint)*** oder einer **Schlüsselabhängigkeit** ***(key constraint)*** in R ist.

- ***domain constraint*** oder ***attribute constraint:*** Jeder Wert eines Attributs muss aus einem bestimmten Wertebereich (Domäne) genommen werden.
- ***key constraint:*** Gewisse Attributkombinationen bilden einen Schlüssel.

DK/NF ist konzeptionell einfach: Es genügt alle Wertebereichsbedingungen und Schlüsselbedingungen einzuhalten und alle anderen Bedingungen sind damit automatisch eingehalten. Jede Relation in DK/NF ist auch in 5NF! Jedoch kann DK/NF nicht immer erreicht werden.

Kapitel 16

Relationaler DB-Entwurf mit E/R-Modell

Im folgenden beschreiben wir ein Rezept, wie die Elemente des E/R-Modells auf Relationen einer relationalen Datenbank abzubilden sind.

16.1 Reguläre Entitäten

Für jede reguläre Entität wird eine Basis-Tabelle angelegt. Jede Basis-Tabelle bekommt einen Primärschlüssel, der entweder schon im E/R-Model angegeben ist oder zusätzlich definiert wird.

Bemerkung: Schlüssel oder ID-Nummern, die im E/R-Model erscheinen, sind meistens Attribute, die vom Auftraggeber gewünscht werden bzw. aus dem UoD kommen und deren Eindeutigkeit daher im Allgemeinen nicht garantiert werden kann. Oft repräsentieren diese Attribute auch Datenwerte. Datenwerte sollten niemals zu Schlüsseln gemacht werden. Deshalb sollte in der Regel *immer* ein künstlicher Primärschlüssel ***(surrogate key)*** definiert werden. Z.B. ist man geneigt, die Matrikelnummer eines Studenten zum Schlüssel zu machen. Die Praxis zeigt aber, dass diese Nummern bezüglich Eindeutigkeit oft fehlerhaft sind.

Wir empfehlen, den Schlüssel in jeder Tabelle mit demselben Namen zu versehen, z.B. `PK` (primary key) oder `ID`.

Für die Fremdschlüssel empfehlen wir als Namen eine Kombination aus `FK` (foreign key) und der referenzierten Tabelle (`FKTabelle`). Da eine Tabelle öfter referenziert werden kann, muss man die Namen eventuell noch qualifizieren (`FK2Tabelle` oder `FKChefTabelle`).

Ferner gibt es den Fall, dass der Fremdschlüssel selbst Schlüssel oder Teil des Schlüssels ist. Wenn der Schlüssel einfach (d.h. nicht zusammengesetzt) ist, dann verwenden wir wieder `PK` als Bezeichnung des Schlüssels. Wenn der Schlüssel zusammengesetzt ist, dann verwenden wir die Fremdschlüsselbezeichnungen.

Die regulären Entitäten unseres Beispiels ergeben daher folgende Basis-Tabellen:

Tabelle Abteilung mit Primärschlüssel PK:

PK	...
...	...

In der folgenden Tabelle ist ANR ein vom Auftraggeber gefordertes eindeutiges Attribut. Wir verwenden es nicht als Schlüssel, sondern definieren einen künstlichen Schlüssel.

Tabelle Angestellter mit Primärschlüssel PK:

PK	ANR	...
...	...	...

Tabelle Projekt mit Primärschlüssel PK:

PK	...
...	...

Tabelle Lieferant mit Primärschlüssel PK:

PK	...
...	...

Tabelle Teil mit Primärschlüssel PK:

PK	...
...	...

16.2 Many-to-Many-Beziehungen

Jede Many-to-Many-Beziehung wird eine eigene Basis-Tabelle **(Fremdschlüsseltabelle)**. Die an der Beziehung teilnehmenden Entitäten werden Fremdschlüssel dieser Basis-Tabelle, wobei Regeln für Update- und Delete-Operationen zu definieren sind.

Für den Primärschlüssel gibt es zwei Möglichkeiten:

1. Kombination der Fremdschlüssel. Das ist nur möglich, wenn die Kombination für jede Instanz der Beziehung eindeutig ist. (Das ist im allgemeinen so.)

2. Einführung eines neuen Primärschlüssels.

Manche Autoren meinen, dass man zusammengesetzte Primärschlüssel vermeiden sollte. Bei Beziehungen sehen wir das nicht so und werden im folgenden zusammengesetzte Schlüssel verwenden. Die Bezeichnung dieser Schlüssel wurde im vorhergehenden Abschnitt diskutiert.

Die Many-to-Many-Beziehungen unseres Beispiels ergeben die Tabellen:

Tabelle ArbeitetAn mit Primärschlüssel (FKAngestellter, FKProjekt):

FKAngestellter	FKProjekt	...
...	...	...

Tabelle LiefertTeil mit Primärschlüssel (FKLieferant, FKTeil):

FKLieferant	FKTeil	...
...	...	...

Tabelle LiefertTeilFürProjekt mit Primärschlüssel (FKLieferant, FKTeil, FKProjekt):

FKLieferant	FKTeil	FKProjekt	...
...	...	...	...

Im folgenden Beispiel haben wir den Fall, dass die Tabelle "Teil" zweimal referenziert wird. Wir qualifizieren die Namen durch "E" (enthält) und "I" (ist enthalten).

Tabelle Struktur mit Primärschlüssel (FKETeil, FKITeil):

FKETeil	FKITeil	...
...	...	...

Die Tabelle Struktur hat folgende Interpretation: FKETeil und FKITeil enthalten Werte von TNR. Das Teil mit der Nummer FKETeil enthält das Teil mit der Nummer FKITeil. FKITeil ist Teil von FKETeil.

16.3 One-to-Many-Beziehungen

Die Beteiligung schwacher Entitäten wird unten behandelt. Falls keine schwache Entität beteiligt ist, dann gibt es prinzipiell zwei Möglichkeiten:

1. In der Tabelle auf der "Many-Seite" wird ein Fremdschlüssel-Attribut angelegt, der die Tabelle auf der "One-Seite" referenziert. Attribute der Beziehung werden mit in der Tabelle auf der "Many-Seite" aufgenommen.
2. Die Beziehung wird wie eine Many-to-Many-Beziehung behandelt. D.h. es wird eine eigene Tabelle angelegt. Das macht man insbesondere dann gern,
   - wenn man die Tabelle der Many-Seite nicht verändern will oder darf.
   - wenn die Beziehung Attribute hat.
   - wenn die Beziehung *nicht* zwingend ist. Das ist insbesondere dann zu empfehlen, wenn die Beziehung selten eingegangen wird. In unserem Beispiel haben wir keinen solchen Fall. Nehmen wir das Beispiel mit den Entitäten ENTLEIHER und BUCH und der Beziehung ENTLEIHT. Das ist sicherlich eine nicht zwingende One-to-Many-Beziehung, die von den meisten (oder sehr vielen) Büchern nicht eingegangen wird. Daher wird man hier eine eigene Tabelle für die Beziehung ENTLEIHT anlegen, die als Fremdschlüssel die Schlüssel von ENTLEIHER und BUCH enthält. Legt man keine eigene Tabelle an, muss man mit NULL-Werten im Fremdschlüssel arbeiten, was verschiedene Autoren als problematisch ansehen.

Die Beziehung "AbtAngest" verändert die Basis-Tabelle "Angestellter" folgendermaßen:

Tabelle Angestellter mit Primärschlüssel PK und Fremdschlüssel FKAbteilung:

PK	ANR	...	FKAbteilung
...	...	...	...

Die Beziehung "Leitet" wird durch den Fremdschlüssel FKAngestellter in der Tabelle Projekt mit Primärschlüssel PK ausgedrückt:

PK	...	FKAngestellter
...	...	...

Die Totalität einer Beziehung ergibt eine NOT-NULL-Spezifikation für den Fremdschlüssel.

16.4 One-to-One-Beziehungen

Diese Beziehungen kommen selten vor und können wie One-to-Many-Beziehungen behandelt werden, obwohl es dort spezielle Probleme geben kann (Beispiel Ehe).

16.5 Dreierbeziehungen

Dreier- oder höherwertige Beziehungen werden immer in eigene Relationen umgesetzt.

16.6 Schwache Entitäten

Schwache Entitäten werden wie Eigenschaften behandelt. D.h., wenn sie mehrwertig sind, werden sie auch zu Basis-Tabellen, wobei die Beziehung zur regulären Entität durch einen Fremdschlüssel ausgedrückt wird. Für den Primärschlüssel der schwachen Entitäten haben wir wieder zwei Möglichkeiten:

1. Kombination des Fremdschlüssels mit einem oder mehreren Attributen der schwachen Entität. Das ist nur möglich, wenn die Kombination für jede Instanz der Beziehung eindeutig ist, d.h. wenn diese Attribute einen sogenannten **partiellen Schlüssel** bilden.
2. Einführung eines neuen Primärschlüssels.

Bei schwachen Entitäten hat der Schlüssel oft keine Bedeutung.

Tabelle Angehöriger mit Fremdschlüssel FKAngestellter und Primärschlüssel (Vorname, FKAngestellter):

Vorname	FKAngestellter	...
...	...	...

Bezüglich des Fremdschlüssels müssen die Regeln für Update und Delete "Kaskadieren" lauten.

16.7 Eigenschaften

Jede Eigenschaft wird ein Attribut der Tabelle der zugehörigen Entität oder Beziehung.

Bei zusammengesetzten Eigenschaften erscheinen nur die Teileigenschaften, oder man legt eine eigene Tabelle an, wo die Teileigenschaften Attribute werden, die Eigenschaft selbst ein Primärschlüssel wird, der als Fremdschlüssel in der Tabelle der Entität oder Beziehung referenziert wird.

Mehrwertige Eigenschaften ergeben gemäß den Normalisierungsregeln weitere Tabellen, in denen durch Fremdschlüssel auf die ursprüngliche Tabelle Bezug genommen wird.

Tabelle Angestellter:

PK	ANR	VorN	MiN	NaN	Gehalt	FKAbteilung
...	...	...	...	...	...	...

Tabelle Adressen:

FKAngestellter	Adresse
...	...

Die Definition des Schlüssels ist hier unwichtig, da er wahrscheinlich keine praktische Bedeutung hat. Er wäre die Kombination aus "FKAngestellter" und "Adresse".

Abgeleitete Eigenschaften dürfen nur in Views auftauchen, oder es wird eine Basis-Relation mit der abgeleiteten Eigenschaft als Attribut definiert, falls die Funktion, mit der die abgeleitete Eigenschaft berechnet wird, als Tabelle darstellbar ist. Der Primärschlüssel dieser Tabelle wird aus allen Attributen gebildet, von denen die Eigenschaft abgeleitet wird.

Schwachbesetzte Attribute, die nur für wenige Tupel relevant sind, d.h. viele `NULL`s enthalten, sollten in eine eigene Relation ausgelagert werden.

16.8 Untertyp, Erweiterung, Vererbung

Die Begriffe "Untertyp, Erweiterung, Vererbung" werden synonym verwendet.

16.8.1 Vertikale Partitionierung

Die **vertikale Partitionierung** entspricht einer relativ treuen Realisierung der Vererbungsbeziehung. Für jeden Untertyp einer Entität wird eine Basis-Tabelle angelegt, die denselben Primärschlüssel hat wie die Tabelle für den Obertyp. Zusätzlich ist der Primärschlüssel auch den Obertyp referenzierender Fremdschlüssel mit kaskadierenden Update- und Delete-Regeln. In einer Hierarchie wird immer nur der unmittelbare Obertyp referenziert. Die Eigenschaften des Obertyps werden nicht wiederholt. Im Untertyp werden nur die neu hinzukommenden Eigenschaften aufgeführt.

Wenn die Entität in der Obertyptabelle gelöscht wird, dann wird sie auch in der Untertyptabelle gelöscht. Das Umgekehrte gilt leider nicht und ist mit relationalen Mitteln nicht implementierbar.

Mehrfachvererbung ist auch implementierbar, indem für jeden geerbten Obertyp ein Fremdschlüssel geführt wird, wobei jeder Fremdschlüssel auch ein Schlüssel der Untertyp-Tabelle sein sollte.

Tabelle Projekt mit Primärschlüssel PK:

PK	PName	...
...	...	...

Tabelle Tagesprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Projekt:

PK ref. Projekt	Tag	...
...	...	...

Tabelle Langzeitprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Projekt:

PK ref. Projekt	Start	Ende	...
...	...	...	...

Tabelle Entwicklungsprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Langzeitprojekt:

PK ref. Langzeitprojekt	...
...	...

16.8.2 Horizontale Partitionierung

Die vertikale Partitionierung hat den Nachteil, dass der Zugriff auf alle Attribute eines Untertyps aufgrund der notwendigen Verbundberechnung eventuell nicht besonders effizient ist.

Bei der **horizontalen Partitionierung** werden in den Tabellen für die Untertypen immer *alle* (auch die geerbten) Attribute aufgenommen. Das ist dann vorteilhaft, wenn der Obertyp abstrakt ist, so dass eigentlich immer nur auf die Instanzen der Untertypen zugegriffen wird.

Tabelle Projekt mit Primärschlüssel PK:

PK	PName
...	...

Tabelle Tagesprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Projekt:

PK ref. Projekt	PName	Tag
...	...	...

Tabelle Langzeitprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Projekt:

PK ref. Projekt	PName	Start	Ende
...	...	...	...

Tabelle Entwicklungsprojekt mit Primärschlüssel PK und Fremdschlüssel PK, Fremdschlüssel referenziert Tabelle Langzeitprojekt:

PK ref. Projekt	PName	Start	Ende	Farbe
...	...	...	...	...

Um Joins zu vermeiden werden alle Daten eines Objekts in der Tabelle seines Typs gehalten. Wie sehen aber die Tabellen der geerbten Typen aus? Dort muss man entweder NULLs verwenden oder die Daten redundant halten. Beides ist nicht befriedigend.

16.8.3 Typisierte Partitionierung

Bei der **typisierten Partitionierung** werden alle Typen einer Vererbungshierarchie in *einer* Tabelle abgebildet. Die Unterscheidung von Instanzen nach verschiedenen Typen erfolgt anhand eines speziellen Typ-Attributs. Anfragen über die gesamte Vererbungshierarchie werden dadurch zwar sehr effizient, aber viele Attribute enthalten NULL-Werte. Das Einfügen eines neuen Typs hat die Erweiterung der Tabelle um neue Attribute zur Folge.

Tabelle Projekt mit Primärschlüssel PK:

PK	Typ	PName	Tag	Start	Ende	Farbe
...	...	...	...	...	...	...
78	Langzeitprojekt	...	...	...	...	...
...	...	...	...	...	...	...
54	Entwicklungsprojekt	...	...	...	...	...
...	...	...	...	...	...	...
91	Projekt	...	...	...	...	...
...	...	...	...	...	...	...
13	Tagesprojekt	...	...	...	...	...
...	...	...	...	...	...	...

Bei der typisierten Partitionierung ist die Vererbungsstruktur nicht mehr ohne weiteres erkennbar.

16.8.4 Andere Varianten

Man sieht gelegentlich eine Art von **umgekehrt vertikaler** Partionierung, wo die Relation der Superklasse Referenzen auf die Relationen der sie erbenden Klassen hat.

Das sähe in unserem Beispiel folgendermaßen aus:

Tabelle Projekt mit Primärschlüssel PK und Fremdschlüsseln, die die erbenden Tabellen referenzieren:

PK	FKTagesprojekt	FKLangzeitprojekt	PName	...
...	...	...	...	...

Tabelle Tagesprojekt mit Primärschlüssel PK:

PK	Tag	...
...	...	...

Tabelle Langzeitprojekt mit Primärschlüssel PK und Fremdschlüssel, der die erbende Tabelle Entwicklungsprojekt referenziert:

PK	FKEntwicklungsprojekt	Start	Ende	...
...	...	...	...	...

Tabelle Entwicklungsprojekt mit Primärschlüssel PK:

PK	...
...	...

Dies hat natürlich den wesentlichen Nachteil, dass der Obertyp jedesmal geändert werden muss, wenn ein Untertyp hinzukommt. Außerdem enthält der Obertyp viele NULLs in den Fremdschlüsseln.

16.9 Kategorien

Eine Kategorie ist ein Untertyp von mehreren sich ausschließenden Obertypen. Die Obertypen haben i.a. verschiedene Schlüssel.

Es gibt drei Möglichkeiten, Kategorien in einer relationalen DB zu implementieren:

1. Für jede Kategorie wird eine Tabelle angelegt. Die Tabelle bekommt einen künstlichen Schlüssel ***(surrogate key)***, der in jeder Obertyptabelle als Fremdschlüssel geführt wird. Dieser Fremdschlüssel kann NULL-Werte annehmen.

2. Für jede Kategorie wird eine Tabelle angelegt. Die Schlüssel der Obertypen werden als Fremdschlüssel geführt, wobei NULL-Werte erlaubt sind. Entweder bildet die Kombination der Fremdschlüssel den Schlüssel für die Tabelle der Kategorie, was das DBMS wegen der NULL-Werte möglicherweise nicht erlaubt, oder es wird ein künstlicher Schlüssel eingeführt.

3. Die sauberste Lösung ist, dass man die Gemeinsamkeiten der Obertypen in eine eigene Tabelle auslagert, von der alle Obertypen *und* die Kategorie erben. Die Kategorie wird natürlich auch eine eigene Tabelle.

16.10 Spezielle Probleme aus der Praxis

16.10.1 Many-to-Many-Beziehung mit History

Am Ende des Kapitels "Entity-Relationship-Modell in UML- Notation" wurde die Many-to-Many-Beziehung zwischen Kunde und Lieferant diskutiert. Wenn man dieses Beispiel in Relationen übersetzt, dann würde die zweite Version folgende Tabellen ergeben:

Tabelle **Kunde** mit Primärschlüssel PK:

PK	...
...	...

Tabelle **Lieferant** mit Primärschlüssel PK:

PK	...
...	...

Tabelle **BestellGeschichte** mit zusammengesetztem Schlüssel:

FKKunde	FKLieferant	KundenNr	...
...	...	...	...

Tabelle **Bestellung** mit einem dreifach zusammengesetzten Schlüssel:

FKKunde ref. BestellGeschichte	FKLieferant	LaufendeNr	Datum	Bestelltext	...
...	...	...	...	...	...

Wenn wir die Entität BestellGeschichte nicht benötigen, weil wir z.B. die KundenNr woanders verwalten, dann hätten wir formal eine ternäre Beziehung zwischen Kunde, Lieferant und Bestellung, wobei die Bestellung eine schwache Entität ist. Bezüglich der Tabellen würde nur die Tabelle BestellGeschichte entfallen.

16.11 SQL-Statements

Im folgenden geben wir die SQL2-Statements an, mit denen die Tabellen des laufenden Beispiels angelegt werden. Wertebereiche werden nicht angelegt, was bei ganz sauberer Vorgehensweise zu tun wäre.

```
CREATE TABLE ABTEILUNG
   (
   PK CHAR (6) NOT NULL,
   ABTNAME CHAR (12),
   PRIMARY KEY (PK)
   );

CREATE TABLE ANGESTELLTER
   (
   PK CHAR (6) NOT NULL,
   ANR CHAR (6) NOT NULL,
   FKABTEILUNG CHAR (6) NOT NULL,
   VORN CHAR (12),
   MIN CHAR (12),
   NAN CHAR (12),
   GEHALT DECIMAL (10, 2),
   PRIMARY KEY (PK),
   FOREIGN KEY (FKABTEILUNG)
      REFERENCES ABTEILUNG (PK)
      ON DELETE NO ACTION
      ON UPDATE CASCADE,
   CHECK (GEHALT > 0.0 AND GEHALT < 1000000.00)
   );

CREATE TABLE ADRESSEN
   (
   PK CHAR (6) NOT NULL,
   FKANGESTELLTER CHAR (6) NOT NULL,
   ADRESSE CHAR VARYING (256),
   PRIMARY KEY (PK),
   FOREIGN KEY (FKANGESTELLTER)
      REFERENCES ANGESTELLTER (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE ANGEHOERIGER
   (
   VORNAME CHAR (12) NOT NULL,
   FKANGESTELLTER CHAR (6) NOT NULL,
   PRIMARY KEY (VORNAME, FKANGESTELLTER),
   FOREIGN KEY (FKANGESTELLTER)
      REFERENCES ANGESTELLTER (FK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE PROJEKT
```

```
   (
   PK INTEGER NOT NULL,
   PNAME CHAR (20),
   FKANGESTELLTER CHAR (6) NOT NULL,
   PRIMARY KEY (PK),
   FOREIGN KEY (FKANGESTELLTER)
      REFERENCES ANGESTELLTER (PK)
      ON DELETE NO ACTION
      ON UPDATE CASCADE
   );

CREATE TABLE ARBEITETAN
   (
   FKANGESTELLTER CHAR (6) NOT NULL,
   FKPROJEKT INTEGER NOT NULL,
   PRIMARY KEY (FKANGESTELLTER, FKPROJEKT),
   FOREIGN KEY (FKANGESTELLTER)
      REFERENCES ANGESTELLTER (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   FOREIGN KEY (FKPROJEKT)
      REFERENCES PROJEKT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE TAGESPROJEKT
   (
   PK INTEGER NOT NULL,
   TAG DATE,
   PRIMARY KEY (PK),
   FOREIGN KEY (PK)
      REFERENCES PROJEKT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE LANGZEITPROJEKT
   (
   PK INTEGER NOT NULL,
   START DATE,
   ENDE DATE,
   PRIMARY KEY (PK),
   FOREIGN KEY (PK)
      REFERENCES PROJEKT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );
```

```
CREATE TABLE ENTWICKLUNGSPROJEKT
   (
   PK INTEGER NOT NULL,
   PRIMARY KEY (PK),
   FOREIGN KEY (PK)
      REFERENCES LANGZEITPROJEKT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE LIEFERANT
   (
   PK CHAR (6) NOT NULL,
   LNAME CHAR (12),
   STATUS INTEGER,
   STADT CHAR (20),
   PRIMARY KEY (PK)
   );

CREATE TABLE TEIL
   (
   PK INTEGER NOT NULL,
   PRIMARY KEY (PK)
   );

CREATE TABLE LIEFERTTEILFUERPROJEKT
   (
   FKLIEFERANT CHAR (6) NOT NULL,
   FKTEIL INTEGER NOT NULL,
   FKPROJEKT INTEGER NOT NULL,
   PRIMARY KEY (FKLIEFERANT, FKTEIL, FKPROJEKT),
   FOREIGN KEY (FKLIEFERANT)
      REFERENCES LIEFERANT (PK)
      ON DELETE NO ACTION
      ON UPDATE CASCADE,
   FOREIGN KEY (FKTEIL)
      REFERENCES TEIL (PK)
      ON DELETE NO ACTION
      ON UPDATE CASCADE,
   FOREIGN KEY (FKPROJEKT)
      REFERENCES PROJEKT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE LIEFERTTEIL
   (
   FKLIEFERANT CHAR (6) NOT NULL,
   FKTEIL INTEGER NOT NULL,
```

```
   PRIMARY KEY (FKLIEFERANT, FKTEIL),
   FOREIGN KEY (FKLIEFERANT)
      REFERENCES LIEFERANT (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   FOREIGN KEY (FKTEIL)
      REFERENCES TEIL (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE
   );

CREATE TABLE STRUKTUR
   (
   FKETEIL INTEGER NOT NULL,  -- Das Teil FKETEIL enthält
   FKITEIL INTEGER NOT NULL,  -- das Teil FKITEIL
   MENGE INTEGER,
   PRIMARY KEY (FKETEIL, FKITEIL),
   FOREIGN KEY (FKETEIL)
      REFERENCES TEIL (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   FOREIGN KEY (FKITEIL)
      REFERENCES TEIL (PK)
      ON DELETE CASCADE
      ON UPDATE CASCADE,
   CHECK (MENGE > 0 OR MENGE = NULL)
   );
```

Kapitel 17

Objektdatenbank (ODB)

17.1 Umfeld

Traditionelle oder **konventionelle** Datenbanken haben sich sehr gut bewährt bei normalen Anwendungen der Wirtschaftsinformatik. Sie zeigen aber Defizite bei komplexeren Anwendungen wie:

- ingenieurmäßigen Entwicklungsarbeiten (CAD/CAM/CAE/CASE)
- integrierter Produktion (CIM)
- Bild- und Graphikdatenbanken
- wissenschaftlichen Datenbanken
- geographischen Informationssystemen
- Multimedia-Datenbanken
- Entwicklung einer einheitlichen Schnittstelle für viele unterschiedliche Datenbanksysteme
- komplexe administrative oder Geschäftsanwendungen

Diese neueren Anwendungen haben Anforderungen, die sich von traditionellen Forderungen in folgendem wesentlich unterscheiden:

- Die Strukturen der Entitäten oder Objekte sind wesentlich komplizierter. Insbesondere handelt es sich um gehäuft auftretende Many-to-Many-Beziehungen und hierarchische Beziehungen.
- Transaktionen dauern manchmal sehr lange (Stunden, Tage, so genannte Objekt-Transaktionen).
- Neue Datentypen zur Speicherung von Bildern und langen Texten werden benötigt.

- Man muss die Möglichkeit haben, nicht-standardisierte, von der jeweiligen Anwendung abhängige Operationen zu definieren.

Der objekt-orientierte Ansatz bietet die Flexibiltät, einige dieser Anforderungen zu erfüllen. Denn das wesentliche Merkmal **objekt-orientierter** Datenbanken oder **Objektdatenbanken (ODB)** ist, dass der Datenbankdesigner *sowohl* die Struktur komplexer Objekte *als auch* die Operationen, die auf diese Objekte angewendet werden können, spezifizieren kann.

Beispiele für experimentelle Prototypen objekt-orientierter Datenbanksysteme (ooDBS) sind:

1. ORION von MCC (Microelectronics and Computer Technology Corporation)
2. OpenOODB von Texas Instruments
3. IRIS von Hewlett-Packard
4. ODE von ATT Bell Labs
5. ENCORE/ObServer von Brown University

Beispiele für kommerziell erhältliche ooDBS sind:

1. GEMSTONE/OPAL von Gemstone Systems
2. ONTOS von Ontology
3. Objectivity von Objectivity
4. Versant von Versant Technologies
5. ObjectStore von Object Design
6. O2 von O2 Technology
7. POET von POET Software
8. db4o von Versant

Was sind die Unterschiede zwischen relationalem und objektorientiertem Modell?

- Paradigma der Datenrepräsentation: Im relationalen Modell werden Daten um *Eigenschaften* gruppiert, im objektorientierten Modell um *Objekte*.

  Eine Person hat die Eigenschaften Angestellter, Manager, Abteilungsmitglied, Projektmitarbeiter zu sein, Telefonnumern zu haben. Die Daten zu diesen Eigenschaften finden sich im relationalen Modell nicht bei der Person, sondern in vielen, die jeweilige Eigenschaft betreffenden Relationen *(Fragmentierung)*. Anfragen bezüglich einer Person müssen Informationen aus vielen Relationen zusammentragen.

  Im oo Modell gehen wir von einer bestimmten Person aus. Alle ihre Eigenschaften sind bei ihr zu finden, u.U. sind das nur Verweise auf andere Objekte (z.B. ein Projekt, das die Person bearbeitet).

- Identität: Im objektorientierten Modell hat jedes Objekt automatisch eine Identität. Im relationalen Modell beruht die Identität auf einer Kombination von Datenwerten.
- Das Problem des sogenannten *impedance mismatch* zwischen Anwendungsprogrammierung (Verarbeitung einzelner Tupel oder Objekte) und Datenbankanfragen (mengenorientiert) wird durch das objektorientierte Modell wesentlich reduziert.

  Das E/R-Modell ist eben schon ein objektorientiertes Modell und kein relationales Modell.

  (*Impedance mismatch:* Datenstrukturen, die das DBMS liefert, sind inkompatibel zu Strukturen, die das Anwendungsprogramm benötigt.)

  Durch unterschiedliche Typsysteme wird das Problem noch vergrößert.
- Das objektorientierte Modell kann auf natürliche Weise komplexe Datenstrukturen oder Objekte repräsentieren. Insbesondere können Hierarchien (– gemeint sind nicht nur Vererbungs-Hiearchien, sondern auch Struktur-Hierarchien, die es zulassen, dass Datenelemente selbst wieder Tabellen sind –) besser abgebildet werden. Bei sich ständig verändernden Hierarchien hat das eine dynamische Änderung von Tabellenstrukturen zur Folge.

  ODBs eignen sich speziell für alle Arten von Stücklistenstrukturen, die bei den meisten kommerziellen Internet-Anwendungen eine Rolle spielen. Alle Katalogsysteme lassen sich in Stücklisten überführen. Komplexe Beziehungen von Elementen untereinander oder rekursive Referenzen können mit einer ODB einfacher abgebildet werden als in einem RDB.

  Eine ODB reduziert den Navigieraufwand von einem Objekt zum anderen ("navigierende Zugriffe").
- Das objektorientierte Modell kann auf natürliche Weise Verhalten von Objekten repräsentieren. Im relationalen Modell muss das Verhalten von Objekten von der Anwendung übernommen werden.
- Mit einer ODB ist eine durchgängige, objekt-orientierte Entwicklung von DBS und Unternehmenssoftware möglich.
- **Orthogonale Persistenz:** Dieses Prinzip bedeutet die Automatisierung der Speicherung von Daten, so dass der Entwickler sich ausschließlich um die Anwendungslogik kümmern kann. ODBs leisten das weitgehend.

  Die Programmiersprache muss die Persistenz-Mechanismen voll und transparent unterstützen.

Was erwartet man von einem ooDBS ?

- Komplizierte, verschachtelte Objekte können als Ganzes oder teilweise manipuliert werden. Die Objekte können weiter verfeinert werden. Objekte können als Ganzes reserviert werden.
- Beliebig lange Daten können gespeichert werden. In den Daten kann gesucht werden.
- Beliebige Datentypen können repräsentiert werden.
- Versionen sollten verwaltbar sein.
- Langandauernde kooperative Transaktionen sollen unterstützt werden.
- Es sollte möglich sein, Regeln zu spezifizieren, damit wissensbasierte Systeme repräsentiert werden können.

17.2 Datenmodell

Ein objekt-orientiertes Datenmodell ist eine Menge von Begriffen zur Modellierung von Daten. Einen allgemein akzeptierten Standard eines oo Datenmodells gibt es noch nicht. Allerdings ODMG scheint sich dazu zu entwickeln. Die im folgenden vorgestellten Begriffe wird man in ähnlicher Form in den meisten oo Programmiersprachen, oo DBS und oo Methodologien wiederfinden.

Objekt und Objekt-Identifikator: Jede Entität der realen Welt ist ein Objekt und hat einen **systemweit** eindeutigen Identifikator.

Attribute und Methoden: Ein Objekt hat Attribute und Methoden, die mit den Werten der Attribute arbeiten. Die Werte von Attributen sind auch Objekte.

Datenkapselung *(encapsulation)* **und Botschaften** *(message passing)***:** Einem Objekt werden Botschaften geschickt, um auf die Werte der Attribute und die Methoden des Objekts zuzugreifen. Die möglichen Botschaften bilden die Oberfläche *(interface)* des Objekts.

Klasse: Alle Objekte, die dieselben Attribute (mit eventuell unterschiedlichen Werten) und Methoden haben, können als zu einer Klasse gehörig betrachtet werden. Ein Objekt gehört nur zu einer Klasse als **Instanz** der Klasse. Eine Klasse ist auch ein Objekt, ist eine Instanz einer **Metaklasse** ***(metaclass)***.

Klassenhierarchie und Vererbung: Die Klassen eines Systems bilden eine Hierarchie oder einen gerichteten, nicht-zyklischen Graphen mit Wurzel. Sei `A` eine Klasse in der Hierarchie und `B` eine Klasse in derselben Hierarchie auf tieferem Niveau. `B` sei irgendwie mit `A` verknüpft. Dann ist `B` eine Spezialisierung von `A` und `A` eine Generalisierung von `B`. `B` ist eine **Subklasse (Unterklasse)** ***(subclass)*** von `A` und `A` ist eine **Superklasse (Oberklasse)** ***(superclass)*** von `B`. Jede Subklasse erbt alle in der Superklasse definierten Attribute und Methoden. Die Subklasse kann zusätzliche Attribute und Methoden haben. Eine Instanz einer Subklasse kann wie eine Instanz der Superklasse verwendet werden.

17.3 Normalisierung

These: *Eine ooDB ist* **normalisiert***, wenn jedes Datenelement oder Objekt höchstens einmal in der DB vorkommt.*

Jedes Datenelement muss unabhängig von anderen Datenelementen manipulierbar sein.

Identifikatoren von Objekten können beliebig oft vorkommen; aber diese Identifikator-Redundanzen müssen von den Objekten transparent für den Benutzer der Objekte verwaltet werden.

Das Problem der Normalisierung ist bei ooDBs nicht so vordringlich wie bei relationalen Datenbanken, da man erstens von einem Objekt-Modell ausgeht, das i.a. "natürlicherweise" normalisiert ist, und da man zweitens die Objekte für die Erhaltung der Konsistenz von Redundanzen verantwortlich machen kann, indem man geeignete Zugriffsmethoden schreibt. Ob eine ooDB normalisiert ist, hängt vom Klassendesign ab. Ein gutes Klassendesign sollte einer Anwendung des Satzes von Kent standhalten, wenn man dort "key" durch "object" ersetzt.

In relationalen Systemen enstehen viele Normalisierungsprobleme in Folge der Verwendung von Datenwerten als Schlüssel. Datenwerte repräsentieren keine Objekte, sondern sind Aussagen über Objekte.

17.4 Persistenzmodell

Es gibt verschiedene Möglichkeiten, die Sprachanbindung zu realisieren:

- Vererbung
- Delegation
- Explizite Datenbank-Aufrufe (z.B. `store`, `create`, `update`, `delete`).

Das Persistenzverhalten sollte eher nicht durch Vererbung, sondern durch Delegation realisiert werden. Sonst steht die Vererbung im eigenen Design nicht mehr zur Verfügung. Aber es ist sehr bequem und elegant, Vererbung zu verwenden. Außerdem sind sowieso fast alle Klassen persistent.

17.5 Anfragen – Queries

Anfragen sind in ooDBs oft irgendwelche Anfrage-Strings, deren Syntax von der verwendeten oo Sprache deutlich abweicht. Das führt zu folgenden Problemen:

- IDEs oder Compiler können die Syntax und Semantik der Anfrage nicht überprüfen.
- Bezeichner, die in Strings vorkommen, werden nicht automatisch bei einem Refactoring nachgezogen.
- Oft werden private Datenelemente direkt angesprochen, was dem Prinzip der Datenkapselung widerspricht.
- Die Entwickler müssen zwischen Programmier- und Anfrage-Sprache hin- und herschalten.
- Typische oo Möglichkeiten (Polymorphismus) stehen bei Anfragen nicht zur Verfügung.

Daher sollte es möglich sein, Anfragen **nativ** zu formulieren, d.h. in der verwendeten oo Sprache. Das Hauptproblem hierbei ist, ob diese Anfragen vernünftig optimiert werden können.

Kapitel 18

Objektorientierter Datenbankentwurf

Wir werden im folgenden einen objektorientierten Entwurf unter Verwendung von ODL der ODMG vorstellen.

Aus Gründen der Korrespondenz und direkten Vergleichbarkeit wird dasselbe Beispiel wie beim E/R-Modell mit den Namen und der Schreibweise der SQL-Implementierung verwendet, obwohl es im oo Umfeld vernünftigere Namenskonventionen gibt.

18.1 Entitäten

Für jede Entität wird eine Klasse definiert. Ein Schlüssel ist nicht nötig, da dies das ooDBS automatisch übernimmt. Um Objekte eindeutig wiederzufinden, kann es allerdings nützlich sein, ein Datenelement als Schlüssel anzulegen. Wenn die Anforderungsanalyse einen Schlüssel fordert, dann muß man ihn einführen. Schlüssel haben keine Bedeutung für Beziehungen. Die Objekte einer Klasse können in ODL immer persistent sein. Sie bekommen einen systemweit eindeutigen Identifikator.

Die Entitäten Abteilung und Angestellter werden zu:

```
class ABTEILUNG
   (extent  Abteilungen)
   {
   // ---
   }

class ANGESTELLTER
   (extent Angestellte key ANR)
   {
   attribute string  ANR;
   // ---
```

```
   }
```

18.2 Eigenschaften

Jede Eigenschaft wird ein Datenelement, das i.a. ein Objekt einer Klasse sein kann. Insbesondere für zusammengesetzte Eigenschaften können Strukturen definiert werden.

Mehrwertige Eigenschaften ergeben entsprechende Collections. Eine übliche Collection ist `set`.

Abgeleitete Eigenschaften werden zu Methoden.

Das Verhalten wird in geeignete Methoden umgesetzt.

Beispiel:

```
struct   aname
   {
   string VORN,
   string MIN,
   string NAN
   }

struct   adresse { string strasse, string ort }

class ANGESTELLTER
   (extent Angestellte key ANR)
   {
   attribute string  ANR;
   attribute aname   ANAME;
   attribute double  GEHALT;
   double   monatsgehalt ();
   attribute set<adresse>  ADRESSE;
   }
```

18.3 Beziehungen

Wenn eine Beziehung Eigenschaften und/oder Methoden hat, dann sollte sie als eigene Klasse angelegt werden. Diese Klasse hat dann einfache many-to-one-Beziehungen zu den Teilnehmern an der Beziehung.

Beziehungen ohne Eigenschaften oder Methoden können ohne Definition einer eigenen Klasse als `relationship` implementiert werden.

Dreier- und höherwertige Beziehungen werden immer als eigene Klasse implementiert.

Als Beipiel implementieren wir die Beziehung "AbtAngest" ohne eigene Klasse und die Beziehung "ArbeitetAn" mit eigener Klasse:

Beziehung "AbtAngest":

```
class ABTEILUNG
   {
   relationship set<ANGESTELLTER> hatAngestellte
      inverse ANGESTELLTER::istInAbteilung;
   }

class ANGESTELLTER
   {
   relationship ABTEILUNG  istInAbteilung
      inverse ABTEILUNG::hatAngestellte;
   }
```

Beziehung "ArbeitetAn":

```
class ANGESTELLTER
   {
   relationship set<ARBEITETAN>  bearbeitet
      inverse ARBEITETAN::angestellter;
   }

class PROJEKT
   {
   relationship set<ARBEITETAN>  wirdBearbeitet
      inverse ARBEITETAN::projekt;
   }

class ARBEITETAN
   {
   relationship ANGESTELLTER  angestellter
      inverse ANGESTELLTER::bearbeitet;
   relationship PROJEKT projekt
      inverse PROJEKT::wirdBearbeitet;
   }
```

Ohne eine eigene Klasse `ARBEITETAN` sieht das folgendermaßen aus:

```
class ANGESTELLTER
   {
   relationship set<PROJEKT>  bearbeitet
      inverse PROJEKT::wirdBearbeitet;
   }

class PROJEKT
   {
```

```
   relationship set<ANGESTELLTER>   wirdBearbeitet
      inverse ANGESTELLTER::bearbeitet;
   }
```

18.4 Aggregation, Komposition – "Hat-ein" – Beziehung

Die "Hat-ein" – Beziehung (`has-a`, Whole-Part, Teil-Ganzes, "ist-Teil-von", Aggregation, Komposition) ist eine One-to-One-Beziehung oder eine One-to-Many-Beziehung ("Hat-viele") und hat keine Eigenschaften. Sie wird als `relationship` oder als `attribute` implementiert.

18.5 Schwache Entitäten

Schwache Entitäten werden wie Eigenschaften bzw wie "Hat-ein"-Beziehungen behandelt.

18.6 Benutzung – "Benutzt-ein" – Beziehung

Die "Benutzt-ein" – Beziehung (`uses-a`) ist meistens eine Many-to-One-Beziehung, selten eine Many-to-Many-Beziehung ("Benutzt-viele") und hat keine Eigenschaften. Sie wird als `relationship` oder als `attribute` implementiert.

```
class User
   {
   relationship Used uses
      inverse Used::used;
   }

class Used
   {
   relationship set<User>  used
      inverse User::uses;
   }
```

Im E/R-Modell sollte angegeben sein, welche Methode des Servers benutzt wird. Diese Information kann in den Namen der Beziehung integriert sein. Diese Methode muß natürlich in der Server-Klasse deklariert und schließlich implementiert werden.

18.7 Vererbung

18.7.1 Erweiterung

```
class TAGESPROJEKT extends PROJEKT
   {
   // ---
   attribute date TAG;
   // ---
   }
```

18.7.2 Obertyp-Untertyp

```
interface VorhabenIfc
   {
   // ---
   }

interface ProjektIfc : VorhabenIfc
   {
   // ---
   }

class PROJEKT : ProjektIfc
   {
   // ---
   }
```

18.8 Ähnlichkeit

Mit der Beziehung "Ähnlichkeit" ist eine Beziehung gemeint, die *nicht* mit Vererbung implementiert werden kann, weil keine Entität die andere substituieren kann.

Gemeinsamkeiten ähnlicher Entitäten müssen in einer eigenen Klasse definiert werden, von der die einander ähnlichen Entitäten dann erben.

18.9 Kategorien

Eine Kategorie `K` kann von verschiedenen Typen `T1` bis `Tn` exclusiv erben. Die Gemeinsamkeiten dieser Typen (gemeinsame Methoden), die die Kategorie `K` benötigt, werden in einer abstrakten Klasse oder Schnittstelle `G` deklariert.

```
interface G
   {
   Returntype  methode ();
   }
```

```
class Ti : G
   {
   Returntype  methode () context ( Methoden-Implementation );
   }

class K
   {
   Returntype  methodeVonK () context ( ... g.methode () ... );
   attribute G g;
   }
```

18.10 Implementation in C++ mit POET^{TM}

Das hier gezeigte C++ Beispiel verwendet das ooDBS POET^{TM}.

```
// AbtAngPLT.hcd
// POET-Beispiel für DB Abteilung, Angestellter, Projekt, Lieferant, Teil

// Copyright 1996 kfg

#ifndef AbtAngPLT_h
#define AbAngPLT_h

#include <poet.hxx>

persistent class  ANGESTELLTER;

persistent class  ABTEILUNG
   {
   private:
      PtString ABTNAME;
      cset<ANGESTELLTER*>  contAngestellter;
         // Behälter, der Zeiger vom Typ ANGESTELLTER* enthält.
   };

class ANAME
   {
   private:
      PtString VORN;
      PtString MIN;
      PtString NAN;
   };

class ADRESSE
   {
```

```
   private:
      PtString adresse;
   };

class ANGEHOERIGER
   {
   private:
      PtString VORNAME;
   };

persistent class  ARBEITETAN;
persistent class  PROJEKT;

persistent class  ANGESTELLTER
   {
   public:
      double   monatsgehalt () { return GEHALT / 13.2; }
      double   gehalt () { return GEHALT; }
      int   gehalt (double GEHALT)
         {
         if (GEHALT > 0.0 && GEHALT < 1000000.00)
            {
            this->GEHALT = GEHALT;
            return 0;
            }
         else return -1;
         }
   private:
      PtString ANR[7];  // Primärschlüssel
      useindex ANRPS;   // POET-Konstrukt für Schlüssel
      ANAME aname;
      double   GEHALT;
      cset<ADRESSE>  contADRESSE;
      cset<ANGEHOERIGER>   contANGEHOERIGER;
      ABTEILUNG*  abteilung;
      cset<ARBEITETAN*> contArbeitetan;   // optional
      cset<PROJEKT*> contLeitet; // optional
   };

unique indexdef   ANRPS : ANGESTELLTER
   {
   ANR;
   };

persistent class  PROJEKT
   {
   private:
      PtString PNAME;
      ANGESTELLTER*  ANR;  // Projektleiter
```

```
      cset<ARBEITETAN*> contArbeitetan;   // optional
   };

persistent class  ARBEITETAN
   {
   private:
      ANGESTELLTER*  angestellter;
      PROJEKT* projekt;
   };

persistent class  TAGESPROJEKT : public PROJEKT
   {
   private:
      PtDate   TAG;
   };

persistent class  LANGZEITPROJEKT : public PROJEKT
   {
   private:
      PtDate   START;
      PtDate   ENDE;
   };

persistent class  ENTWICKLUNGSPROJEKT : public LANGZEITPROJEKT
   {
   };

persistent class  TEIL;

persistent class  LIEFERANT
   {
   private:
      PtString LNAME;
      int   STATUS;
      PtString STADT;
      cset<TEIL*> contTNR; // Teile, die geliefert werden
   };

persistent class  STRUKTUR;

persistent class  TEIL
   {
   private:
      cset<LIEFERANT*>  contLNR; // Lieferanten
      STRUKTUR*   SNR;
   };

persistent class  LIEFERTTEILFUERPROJEKT
   {
```

```
   private:
      LIEFERANT*  LNR;
      TEIL*       TNR;
      PROJEKT*    PNR;
   };

persistent class  STRUKTUR
   {
   private:
      TEIL* TNRE;
      cset<TEIL*> contTNRI;
      int   MENGE;
   };

#endif
```

Nun beginnt die eigentliche Arbeit: Definition und Implementation geeigneter Konstruktoren, Destruktoren, Zugriffsmethoden, Anfragemethoden und Datenpräsentationsmethoden.

Kapitel 19

Datenbeschreibungssprache XML

XML ***(Extensible Markup Language)*** ist eine universelle Konvention zur Beschreibung von Daten auf Basis von SGML und geht weit über den im Web hauptsächlich verwendeten HTML-Standard *(Hypertext Markup Language)* hinaus.

- Mit XML werden Anwendungen auf Datenebene entkoppelt. Die Kopplung zwischen Daten ist "lose".
- **Trennung** von Inhalt und Formatierung
- XML-Dateien sind von Menschen lesbar.
- XML ist von **aufwärtskompatibel**. D.h. Strukturen können erweitert werden, ohne daß Code gebrochen wird. Eine simultane Umstellung von Software ist nicht erforderlich. Auch kann die Reihenfolge innerhalb einer Struktur – in Grenzen – verändert werden.
- Daten können bei der Übertragung von einem System zum anderen durch den Sender und den Empfänger **validiert** werden. Unterschiedliche Systemarchitekturen werden durch XML kompatibel.
- Daten können in – fast – beliebige andere Formate transformiert werden.

19.1 Überblick

XML erlaubt es, jeden Datensatz mit einer Selbstbeschreibung auszustatten und ihn in strukturlosen Behältern zu speichern. Damit würde eine Flut von immer neuen, anders strukturierten Tabellen vermieden.

Wie HTML basiert auch XML auf **SGML** ***(Standard Generalized Markup Language)***

Gegenüber HTML hat XML folgende Erweiterungen:

- Neue *Tags* (Datenauszeichnungen, die eine Bedeutung beinhalten) können jederzeit vom Benutzer definiert werden.

- Daten- und Dokumentstrukturen können beliebig geschachtelt werden, d.h. man kann sehr komplexe Objekte darstellen.
- Jedes Dokument bzw jeder Datensatz kann eine Bauvorschrift (Grammatik) etwa für eine Syntaxprüfung enthalten.

Der Aufbau eines XML-Dokuments und seiner Tags kann in einer eigenen Datei, der ***document type definition*** **(DTD)** oder besser einem **XML-Schema**, definiert werden. Durch die Weitergabe der XML-Datei zusammen mit der DTD-Datei oder dem XML-Schema lassen sich die Daten von verschiedenen unabhängigen Programmen verarbeiten.

XML definiert abstrakte Semantiken. Damit sind die Dokumente für die Publikation in unterschiedlichen Medien (Bücher, CDs, Web) geeignet. Dies macht XML dem Ansatz von EDI *(electronic data interchange)* überlegen, das operationale Semantiken definiert und daher spezielle, teure Implementierungen erfordert.

XML-Dokumente sind regelmäßig komplexe Baumstrukturen, die mit dem relationalen Modell schwer zu beschreiben sind. Daher eignen sich hier besonders objekt-orientierte DBMSs. Denn dort ist kein Paradigmenwechsel notwendig.

Wenn die XML-Daten nur schwer in Tabellen eines RDBS zu normalisieren sind, dann sollte man eventuell ein sogenanntes "natives" XML-DBS **(NXD)** ***(native XML database)*** verwenden, das eine sehr große Ähnlichkeit mit objektorientierten oder objektrelationalen Datenbanken hat. Ein Beispiel für ein NXD ist **Xindice**, das von der Apache Software Foundation entwickelt wird. Zur Suche und Aktualisierung von Daten in Xindice wird **XPath** und **XUpdate** verwendet.

Fogende XML-API seien kurz erwähnt:

SAX: Simple API for XML
: Das ist eine sogenannte "Ereignis-basiertes" API ***(event-based)***. Dabei wird das XML-Dokument geparst, und jegliche Information (Start oder Ende oder Wert eines Elements) als Ereignis an die aufrufende Instanz zurückgegeben. Der Vorteil ist, daß wenig Speicher verwendet wird. Allerdings muß die XML-Datei für jede Anfrage neu geparst werden.

DOM: Document Object Model
: Das ist ein "Baum-basiertes" API ***(tree-based)***. Die XML-Datei wird einmal geparst und es wird im Speicher ein Modell davon erstellt, das die Struktur und den Inhalt repräsentiert. Bei großen XML-Dokumenten wird möglicherweise zu viel Speicher benötigt.

Bemerkung: Bei vielen Zugriffen und mäßig großen XML-Dokumenten empfiehlt sich DOM, bei seltenen Zugriffen oder sehr großen XML-Dokumenten empfiehlt sich SAX.

Kapitel 20

Netzwerk-Datenmodell

Vor dem Aufkommen von relationalen DB spielten Netzwerk- und hierarchisches Modell (Datenmodelle der "1. Generation") eine große Rolle bei kommerziellen DBMS.

Netzwerk-Modell-Strukturen und -Sprachkonstrukte wurden von CODASYL-DBTG (Conference on Data System Languages – Database Taskgroup) 1971 definiert. Seit 1984 gibt es von ANSI eine Netzwerk-Definitions-Sprache (NDL).

IDMS (Integrated Database Management System) von IBM und UDS (Universal Database System) von Siemens sind kommerzielle Netzwerk-DBMSs.

20.1 Netzwerk-Datenstrukturen

Im Netzwerk-Modell gibt es zwei Strukturen: **Datensätze** ***(record)*** und **Mengen** ***(set)***.

Daten werden in Datensätzen gespeichert. Jeder Datensatz hat einen benannten **Datensatztyp** ***(record type)***, der den Aufbau des Datensatzes beschreibt. Der Datensatz besteht aus benannten **Datenelementen** ***(data item, attribute)***, die alle ein **Format** ***(format, data type)*** haben. Als Beispiel betrachten wir den Datensatztyp STUDENT:

STUDENT		
NAME	MATRIKEL	GEBURTSTAG

Datenelementtyp	Format
NAME	CHARACTER 30
MATRIKEL	CHARACTER 6
GEBURTSTAG	CHARACTER 8

Die Datenelemente können eine komplexe Struktur haben: Insbesondere sind **Wiederholungsgruppen** ***(repeating group)*** erlaubt. Zum Beispiel könnte ein STUDENT eine Wiederholungsgruppe LEISTUNG haben, die aus zwei Datenelementen FACH und NOTE besteht, wobei die LEISTUNG öfters vorkommt.

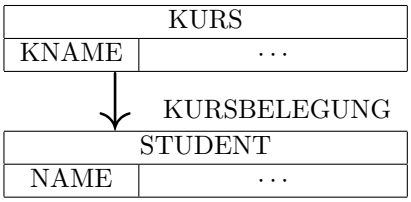
STUDENT				
NAME	MATRIKEL	GEBURTSTAG	LEISTUNG	
			FACH	NOTE

Dies kann sehr tief verschachtelt sein, so dass sehr komplexe Strukturen darstellbar sind.

Es gibt **virtuelle** ***(virtual)*** oder **abgeleitete** ***(derived)*** Datenelemente, wo der Wert nicht gespeichert wird, sondern mit einer Prozedur aus anderen Daten abgeleitet wird (z.B. ALTER).

Beziehungen zwischen Datensätzen werden mit Mengen ausgedrückt: Ein **Mengentyp** ***(set type)*** beschreibt eine 1:M-Beziehung zwischen zwei Datensatztypen, einem **Besitzertyp** ***(owner record type)*** und einem **Mitgliedstyp oder Teilnehmertyp** ***(member record type)***.

Z.B. hat der Mengentyp KURSBELEGUNG als Besitzertyp den KURS und als Mitgliedstyp STUDENT und wird im folgenden **Bachmann**-Diagramm dargestellt:



Die Pfeilrichtung ist in manchen Büchern anders herum. Die Bachmann-Diagramme bilden das sogenannte **Netzwerkschema**.

Ein STUDENT darf nur in *einer* Beziehung KURSBELEGUNG vorkommen, da ja eine 1:M-Beziehung dargestellt werden soll. D.h. ein Student kann nur zu einem Kurs gehören, aber er kann natürlich noch Beziehungen zu anderen Besitzertypen haben, z.B. Firma. Jede Menge kann daher entweder durch den Besitzer oder ein Mitglied eindeutig identifiziert werden. Jede Menge muss einen Besitzer haben, muss aber kein Mitglied haben. Daher werden Mengen häufig mit dem Besitzer identifiziert.

Die Mitglieder einer Menge sind geordnet. Wir können vom ersten, zweiten, i-ten und letzten Datensatz sprechen. Im Unterschied zur mathematischen Menge spricht man im Netzwerk-Modell vom ***owner-coupled set*** oder ***co-set***. Sie sind eigentlich Listen und werden auch so implementiert (Verzeigerung der Datensätze). Die Reihenfolge der Datensätze spielt eine große Rolle bei der Anwendungsprogrammierung. Die ProgrammiererIn verlässt sich darauf.

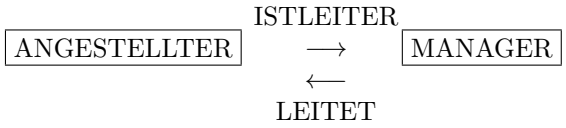
Das Netzwerk-Modell kennt nur binäre Beziehungen. Mehrstellige Beziehungen werden durch einen zusätzlichen Datensatztyp *(kett-record-type)* dargestellt. Jede Entität der mehrstelligen Beziehung hat als Besitzer eine binäre Beziehung zu dem Kett-record.

System-eigene Mengen ***(system-owned, singular set)*** sind spezielle Mengen ohne Besitzerdatensatztyp. Stattdessen ist die DBMS-Software der Besitzer. Solche Mengen werden als ***entry points*** in die DB benutzt. Die Verarbeitung beginnt mit dem Zugriff auf ein Mitglied einer System-eigenen Menge. Ferner können mehrere System-eigene Mengen auf denselben Mitgliedern definiert werden, die verschieden sortiert werden können, wodurch unterschiedliche Zugriffsmöglichkeiten realisiert werden können.

Bei **Multi-Mitglieds-Mengen** ***(multimember set)*** können die Mitglieder unterschiedlichen Typs sein.

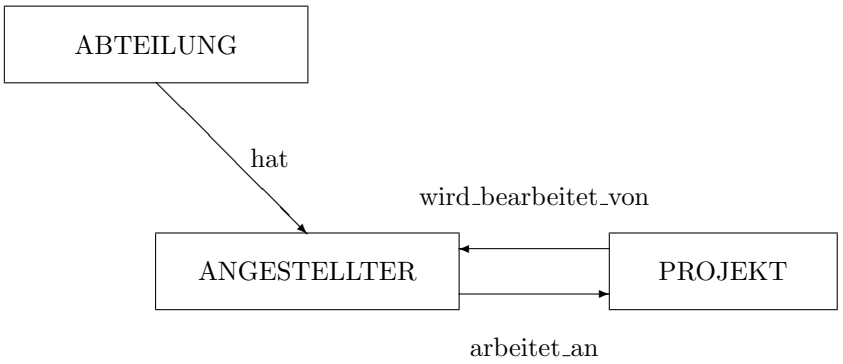
Bei **rekursiven** Mengen ***(recursive set)*** sind Besitzer und Mitglied vom gleichen Typ.

Rekursive Mengen sind in den meisten DBMS nicht erlaubt. Man muss sich mit ***dummy*** oder ***linking*** Datensatztypen behelfen.



Dabei wird eine Menge zur 1:1-Beziehung. 1:1-Beziehungen können im Modell nicht besonders dargestellt werden. Der DB-Programmierer ist verantwortlich für die Einhaltung einer 1:1-Beziehung.

M:N-Beziehungen, wie im folgenden Bild zwischen ANGESTELLTER und PROJEKT,



können nur über einen zusätzlichen Datensatztyp dargestellt werden:

20.2 Netzwerk-Integritätsbedingen

Siehe Kapitel 10 von Elmasri/Navathe [17].

20.3 Netzwerk-Datenbankentwurf

Die Übersetzung eines E/R-Modells in ein Netzwerkmodell geht bis auf Vererbungsbeziehungen ziemlich direkt.

Kapitel 21

Hierarchisches Datenmodell

Das hierarchische Datenmodell wurde entwickelt, um die vielen hierarchischen Organisationen der realen Welt zu modellieren. Das berühmteste Beispiel ist das Klassifikationsschema von Linné. Die natürliche Anwendung sind hierarchische Organisationsstrukturen in Unternehmen und Verwaltungen.

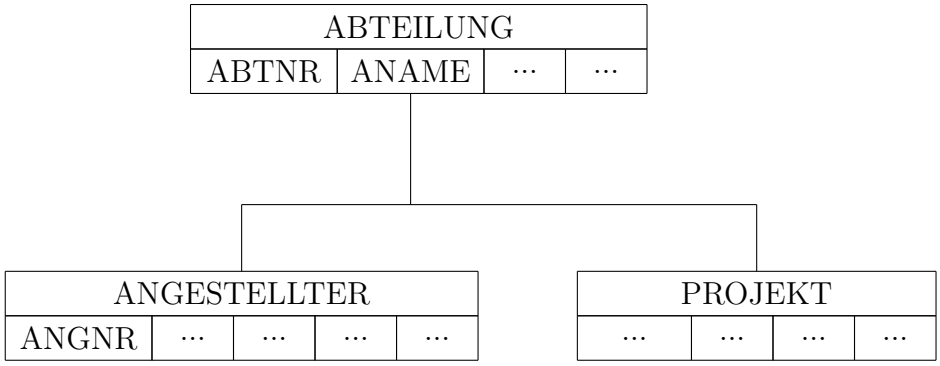
Beispiele für hierarchische Informations-Management-Systeme sind MARS VI (Multi-Access Retrieval System) von Control Data, IMS (Information Management System) von IBM, System-2000 von MRI/SAS oder IMAGE 9000 von HP.

21.1 Datenstruktur

Das hierarchische Modell hat zwei Strukturierungsprinzipien: **Datensätze** ***(record)*** und **Elter-Kind-Beziehung** ***(parent-child relationship, PCR-type)***. Datensätze bestehen aus **Feldwerten** ***(field value, data item)***, die Information über eine Entität oder eine Beziehung enthalten. Die Art und die Namen der Felder bestimmen den Typ des Datensatzes. Der Datensatz-Typ hat auch einen Namen. Die Felder haben Datentypen wie Integer, Real oder String.

Eine Elter-Kind-Beziehung ist eine 1:M-Beziehung zwischen zwei Datentypen. Der Datensatz auf der 1-Seite ist der **Eltertyp** ***(parent record type)***, der auf der M-Seite der **Kindtyp** ***(child record type)***. Die Vererbung wird hier für alle möglichen Beziehungen – nicht nur "Ist-ein" – verwendet. Häufig ist es hier eine "Hat-ein"-Beziehung.

Ein **hierarchisches Datenbankschema** ***(hierarchical database schema)*** besteht aus mehreren **Hierarchien** ***(hierarchy)*** ("ein Wald von Bäumen"). Ein hierarchisches Schema wird als Diagramm dargestellt, wobei Datensatztypen in Rechtecken erscheinen, optional darunter Feldnamen. Beziehungen werden durch Linien dargestellt.

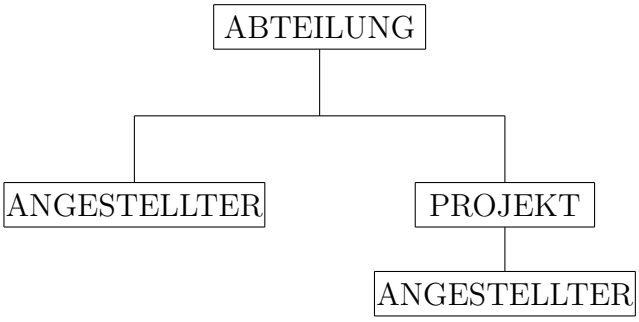


Beziehungen werden durch Nennung der Partner bezeichnet:
(ABTEILUNG, ANGESTELLTER)
(ABTEILUNG, PROJEKT)

Eigenschaften eines hierarchischen Schemas:

1. Ein spezieller Datensatztyp, die **Wurzel** ***(root)***, ist niemals Kind.
2. Jeder Datensatztyp kann in einer Hierarchie als Kind nur in genau einer Beziehung vorkommen.
3. Als Elter kann ein Datensatztyp in beliebig vielen Beziehungen vorkommen.
4. Ein Datensatz, der kein Elter ist, ist ein **Blatt** ***(leaf)***.
5. Wenn ein Elter mehrere Kindtypen hat, dann sind diese von links nach rechts geordnet.

Eine M:N-Beziehung kann nicht direkt dargestellt werden. Als Beispiel betrachten wir die M:N-Beziehung (ANGESTELLTER, PROJEKT):



Das bedeutet, dass Datensatzinstanzen *dupliziert* werden müssen bzw noch öfter vorkommen. Diese Probleme können mit **virtuellen** ***(virtual)*** Beziehungen behandelt werden.

Die Instanziierung eines hierarchischen (Teil-)Schemas heißt **hierarchische Instanz** ***(hierarchical occurrence)*** oder **Instanzenbaum** ***(occurrence tree)***, wobei ein Typindikator mitgeführt wird.

Im Speichermedium wird solch ein Instanzenbaum i.a. als **hierarchischer Datensatz** ***(hierarchical record)*** repräsentiert, der die Datensätze im Instanzenbaum **linear** ***(preorder traversal, hierarchical sequence)*** nach folgendem rekursiven Algorithmus anordnet:

```
procedure preorder_traverse (parent_record);
   begin
   output (parent_record);
   for each child_record of parent_record in left to right order
      do preorder_traverse (child_record);
   end;
```

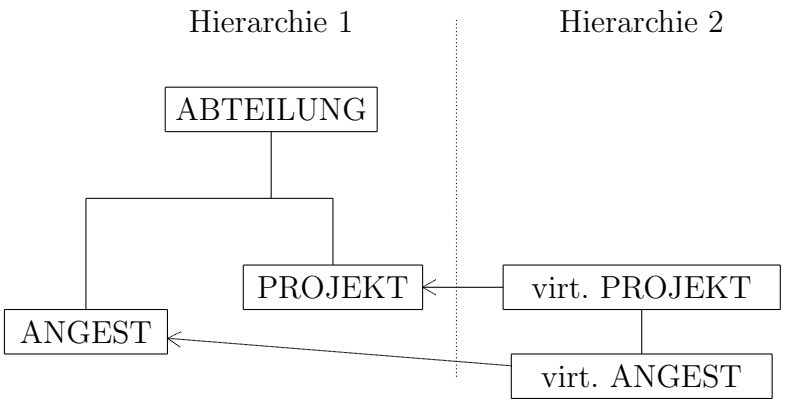
Das bedeutet, dass nach jeder Elterinstanz alle Kindinstanzen und deren Nachkommen aufgeführt werden, wobei ein Typindikator mitgeführt wird, damit die Sequenz wieder gelesen werden kann.

Das Durchlaufen von Hierarchien oder Teilhierarchien ist die wichtigste Operation bei hierarchischen Datenbanken.

Das Modell hat Probleme mit

1. M:N-Beziehungen
2. Söhnen mit mehreren Vätern
3. mehrstelligen (ternären und höherwertigen) Beziehungen

Datensatzduplikation ist äußerst problematisch. Im IMS wird das Konzept virtueller Datentypen verwendet, um diese Probleme zu lösen. Ein virtueller Datensatz enthält nur einen Zeiger auf einen Datensatz. Es wird dabei auch die Elter-Kind-Terminologie verwendet, wobei der virtuelle Datensatz das Kind ist und die Beziehung eine virtuelle Elter-Kind-Beziehung (VPCR) ist.



Die genannten Probleme werden mit mehreren Hierarchien, mit durch Pfeilen dargestellte VPCRs und unter Verwendung verschiedenster Tricks gelöst (virtuelle dummy-Datensatztypen).

Kapitel 22

Transaktionsverarbeitung

Das Thema der **Transaktionsverarbeitung** ***(transaction processing)*** ist ein riesiges Thema [20]. Wir werden daher nur die wichtigsten Gesichtspunkte darstellen. Nach einer Klärung des Begriffs **Transaktion** ***(transaction)*** gehen wir auf die beiden Aspekte **Nebenläufigkeit** ***(concurrency)*** und in einem eigenen Kapitel **Wiederherstellung** ***(recovery)*** ein, wobei wir häufig die englischen Begriffe verwenden werden, weil sie eingeführt sind.

22.1 Transaktionen

Definition: *Eine Transaktion ist eine logische Arbeitseinheit, die zu einem einzelnen Vorgang oder einer elementaren Ausführungseinheit in dem Unternehmen korrespondiert, das in der Datenbank abgebildet ist.*

Als Beispiel nehmen wir an, dass die Teile-Relation P als zusätzliches Attribut TOTQTY habe, nämlich die Menge aller gelieferten Teile eines Typs. Nun soll die neue Lieferung "S5 liefert 180 P1-Teile" in die Datenbank eingetragen werden. Dazu müssen die Relationen SP und P aktualisiert werden:

```
EXEC SQL INSERT
   INTO SP (SNR, PNR, QTY)
   VALUES ('S5', 'P1', 180);
EXEC SQL UPDATE   P
   SET   TOTQTY = TOTQTY + 180
   WHERE PNR = 'P1';
```

Dies ist ein Beispiel für eine Transaktion. Eine Transaktion erhält die Konsistenz und Korrektheit der Datenbank, d.h. die Datenbank wird durch die Transaktion von einem konsistenten und korrekten Zustand in einen anderen konsistenten und korrekten Zustand überführt. Zwischenzustände (etwa nach der INSERT-Anweisung) können inkonsistent sein. Um die Konsistenz der Datenbank zu garantieren, darf eine Transaktion entweder nur vollständig oder garnicht durchgeführt werden, d.h. Transaktionen müssen **unteilbar, atomar** ***(atomic)*** sein. Es muss für die sogenannte **Ablauf-** oder **operationale** Integrität gesorgt werden.

Die Systemkomponente **Transaktions-Manager** ***(transaction manager)*** stellt Methoden zur Verfügung, die die Atomarität garantieren:

`begin transaction (BOT)` : Mit diesem Statement beginnt eine Transaktion. Oft ergibt sich dieses Kommando implizit und wird weggelassen. Auch in SQL gibt es dafür kein explizites Statement. In SQL beginnt eine Transaktion, wenn gerade keine Transaktion läuft und ein **transaktionsinitiierendes** (fast alle DDL- und DML-Statements) Statement gegeben wird.

`commit transaction (EOT)` : Mit diesem Statement sagt man dem Transaktions-Manager, dass die logische Arbeitseinheit erfolgreich abgeschlossen wurde und die Datenbank wieder in einem konsistenten Zustand ist. Alle durch diese Transaktion veranlassten Änderungen an der Datenbank können permanent gemacht werden. In SQL heißt das Statement:
`COMMIT;`

`rollback (abort) transaction (EOT)` : Durch dieses Statement wird dem Transaktions-Manager gesagt, dass die Transaktion nicht erfolgreich zu Ende gebracht wurde und dass die Datenbank möglicherweise in einem inkonsistenten Zustand ist. Alle seit Beginn der Transaktion durchgeführten Änderungen an der Datenbank müssen rückgängig gemacht werden. In SQL heißt das Statement:
`ROLLBACK;`

In SQL wird mit `COMMIT` oder `ROLLBACK` eine Transaktion beendet.

Die oben gezeigte Transaktion hat mit diesen Mechanismen nun folgende Form:

```
EXEC SQL WHENEVER SQLERROR GO TO rollback;
EXEC SQL INSERT -- Beginn der Transaktion
   INTO SP (SNR, PNR, QTY)
   VALUES ('S5', 'P1', 180);
EXEC SQL UPDATE   P
   SET   TOTQTY = TOTQTY + 180
   WHERE PNR = 'P1';
EXEC SQL COMMIT -- Ende der Transaktion;
goto ende;
rollback: EXEC SQL ROLLBACK  -- Ende der Transaktion;
ende: ;
```

Diese Mechanismen sind normalerweise nur mit embedded SQL erlaubt, da sie interaktiv zu gefährlich sind. Manche Implementationen aber erlauben interaktive Anwendung und stellen entsprechende Mechanismen zur Verfügung.

Transaktionen sollten die sogenannten ACID-Eigenschaften haben:

- **A***tomicity* : Atomarität. Transaktionen sind atomar ("Alles oder Nichts"). Eine Transaktion wird entweder ganz durchgeführt oder hat keine Auswirkung auf die Datenbank.

- **C***onsistency* : Konsistenz. Integritätserhaltung. Eine Transaktion transformiert die Datenbank von einem konsistenten (korrekten) Zustand in einen anderen konsistenten Zustand. Während einer Transaktion kann es Datenbank-Zustände geben, die nicht konsistent oder korrekt sind.
- **I***solation* : Transaktionen sind voneinander isoliert. Keine Transaktion kann Zwischenzustände einer anderen Transaktion sehen. Die Transaktion T_1 kann die Änderungen von Transaktion T_2 nach deren `COMMIT` sehen, oder T_2 kann die Änderungen von T_1 nach deren `COMMIT` sehen. Beides oder etwas dazwischen ist nicht möglich.
- **D***urability* : Dauerhaftigkeit. Persistenz. Änderungen an der Datenbank sind nach einem `COMMIT` permanent, auch wenn es sofort danach zu einem Systemabsturz kommt.

22.2 Nebenläufigkeit *(Concurrency)*

22.2.1 Probleme bei Nebenläufigkeit

Ursache für Datenbank-Fehler ist der "gleichzeitige" Zugriff von verschiedenen Transaktionen auf dasselbe Datenobjekt.

Folgende drei Probleme können auftreten, wenn Transaktionen (im folgenden A und B) parallel durchgeführt werden. (Für die Zeiten t_i gilt $t_i < t_j$, falls $i < j$):

lost update **(verlorene Aktualisierung)** bei folgendem Szenario:

1. Transaktion A ermittelt Tupel p zur Zeit t_1.
2. Transaktion B ermittelt Tupel p zur Zeit t_2.
3. Transaktion A aktualisiert Tupel p zur Zeit t_3 (auf der Basis von Werten, die zur Zeit t_1 gesehen wurden).
4. Transaktion B aktualisiert Tupel p zur Zeit t_4 (auf der Basis von Werten, die zur Zeit t_2 gesehen wurden).

Zur Zeit t_4 ist die Aktualisierung durch A verloren, da B sie überschrieben hat.

uncommitted dependency (dirty read, **Abhängigkeit von nicht permanenten Daten)** bei folgendem Szenario:

1. Transaktion B aktualisiert Tupel p zur Zeit t_1.
2. Transaktion A ermittelt (oder aktualisiert gar) Tupel p zur Zeit t_2.
3. Transaktion B macht ein `ROLLBACK` zur Zeit t_3.

A hat falsche Daten (oder hat seine Aktualisierung verloren).

inconsistent analysis (non repeatable read, **Arbeit mit inkonsistenten Daten)** bei folgendem Szenario: A soll die Summe S von drei Konten $K_1 = 40, K_2 = 50, K_3 = 60$ bestimmen. B transferiert einen Betrag 10 von Konto K_3 nach Konto K_1.

1. Transaktion A ermittelt $K_1(= 40)$ und summiert $(S = 40)$.
2. Transaktion A ermittelt $K_2(= 50)$ und summiert $(S = 90)$.

3. Transaktion B ermittelt $K_3(= 60)$.
4. Transaktion B aktualisiert $K_3 = K_3 - 10 = 50$.
5. Transaktion B ermittelt $K_1(= 40)$.
6. Transaktion B aktualisiert $K_1 = K_1 + 10 = 50$.
7. Transaktion B `COMMIT`.
8. Transaktion A ermittelt $K_3(= 50)$ und summiert $(S = 140)$.

A hat einen inkonsistenten Zustand der Datenbank gesehen. Nach 6. ist das erste Lesen von K_1 nicht wiederholbar.

phantoms (repeatable read, **Lesen von Phantomen)** Eine zweite Transaktion fügt ein Tupel ein, das der Selektionsbedingung einer ersten Transaktion genügt. Wenn die erste Transaktion die Selektion wiederholt, dann liest sie beim zweiten Mal mehr Tupel. Entsprechendes gilt für Tupel, die zwischendurch gelöscht werden.

22.2.2 Sperren *(Locking)*

Parallele Transaktionen müssen geeignet **synchronisiert** ***(synchronized)*** werden. Die im vorhergehenden Abschnitt angesprochenen Probleme kann man mit **Sperrmechanismen** ***(locking)*** lösen. Überlicherweise gibt es folgende Arten von **Sperren** ***(locks)***:

1. **X-Lock** *(exclusive lock, write-Lock, write-exclusive,* exclusives Schreiben, Schreibsperre): Erlaubt Schreib- und Lesezugriff auf ein Objekt nur für die das X-Lock beantragende Transaktion. Hat die Transaktion das X-Lock auf ein Objekt bekommen, werden Lock-Beantragungen anderer Transaktionen zurückgewiesen, d.h. diese Transaktionen werden in Warteschlangen für das betreffende Objekt gestellt.

2. **S-Lock** *(shared lock, read-lock, read-sharable,* gemeinsames Lesen, Lesesperre): Ein Lesezugriff auf ein Objekt ist erlaubt. Ein Schreibzugriff ist nicht möglich. Hat eine Transaktion ein S-Lock auf ein Objekt, dann können weitere Transaktionen ein S-Lock auf dasselbe Objekt bekommen. Versuche, ein X-Lock zu bekommen, werden zurückgewiesen.

3. **U-Lock** *(upgrade lock,* Änderungssperre): Dieses Lock findet man bei OODBS. Bei richtiger Verwendung (siehe unten) verhindert es Verklemmungen. Nur eine Transaktion darf ein U-Lock auf ein Objekt haben. Andere Transaktionen können aber S-Locks für dieses Objekt bekommen. Eine Transaktion, die auf ein Objekt ein U-Lock setzt, beabsichtigt, dieses Lock im Laufe der Transaktion zu einem X-Lock zu verschärfen *(upgrade)*.

Diese Regeln werden üblicherweise in sogenannten **Kompatibilitätsmatrizen** zusammengestellt:

	`S`-Lock	`X`-Lock
`S`-Lock	ja	nein
`X`-Lock	nein	nein

	S-Lock	U-Lock	X-Lock
S-Lock	ja	ja	nein
U-Lock	ja	nein	nein
X-Lock	nein	nein	nein

In SQL gibt es keine expliziten Lockingmechanismen. Normalerweise werden die entsprechenden Locks (S- bzw X-Lock) implizit von einer Retrieve- oder Update-Anweisung angefordert. Die Locks sollten bis zum Ende der Transaktion (COMMIT oder ROLLBACK) gehalten werden.

Die drei genannten Probleme werden mit Lockingmechanismen folgendermaßen gelöst:

lost update :

1. Transaktion A fordert S-Lock für Tupel p, bekommt das S-Lock und ermittelt Tupel p zur Zeit t_1.
2. Transaktion B fordert S-Lock für Tupel p, bekommt das S-Lock und ermittelt Tupel p zur Zeit t_2.
3. Transaktion A fordert X-Lock für Tupel p zur Zeit t_3 und muss auf das X-Lock warten, da Transaktion B ein S-Lock auf Tupel p hat.
4. Transaktion B fordert X-Lock für Tupel p zur Zeit t_4 und muss ebenfalls auf das X-Lock warten, da Transaktion A ein S-Lock auf Tupel p hat.
5. ...

Verklemmung *(Deadlock!)*!!

uncommitted dependency :

1. Transaktion B fordert X-Lock für Tupel p, bekommt das X-Lock und aktualisiert Tupel p zur Zeit t_1.
2. Transaktion A fordert S-Lock (X-Lock) für Tupel p zur Zeit t_2 und muss warten, bis B das X-Lock freigibt.
3. Transaktion B macht ein ROLLBACK zur Zeit t_3 und gibt damit alle Locks frei.
4. Transaktion A bekommt das S-Lock (X-Lock) und ermittelt (aktualisiert) Tupel p zur Zeit t_4.

inconsistent analysis :

1. Transaktion A fordert S-Lock für K_1, bekommt dies, ermittelt $K_1(= 40)$ und summiert $(S = 40)$.
2. Transaktion A fordert S-Lock für K_2, bekommt dies, ermittelt $K_2(= 50)$ und summiert $(S = 90)$.
3. Transaktion B fordert S-Lock für K_3, bekommt dies und ermittelt $K_3(= 60)$.
4. Transaktion B fordert X-Lock für K_3, bekommt dies und aktualisiert $K_3 = K_3 - 10 = 50$.
5. Transaktion B fordert S-Lock für K_1, bekommt dies und ermittelt $K_1(= 40)$.
6. Transaktion B fordert X-Lock für K_1 und bekommt dies nicht, weil A ein S-Lock darauf hat. Wartet.

7. Transaktion A fordert S-Lock für K_3 und bekommt dies nicht, weil B ein `X`-Lock darauf hat. Wartet.
8. ...

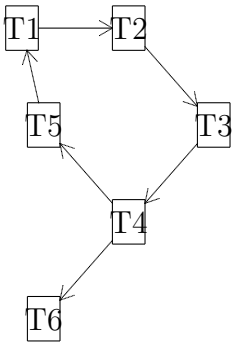
Verklemmung *(Deadlock)*!!!

Behandlung von Verklemmung:

Voranforderungsstrategie *(preclaim strategy)***:** Jede Transaktion fordert alle ihre Sperren an, bevor sie mit der Ausführung beginnt ("konservativer Scheduler"). Das verringert den Grad der Nebenläufigkeit bei Datenbanksystemen derart, dass dies i.a. nicht akzeptabel ist. Ferner ist es oft unmöglich, vor der Ausführung einer Transaktion zu wissen, was zu sperren ist.

Toleranzstrategie: Verklemmungen werden erlaubt und gelöst ("aggressiver Scheduler"). Zur Erkennung einer Verklemmung muss ein **Vorranggraph (Konfliktgraph, *Wait-For-Graph)*** aufgebaut werden, dessen Knoten die derzeit laufenden Transaktionen sind und dessen Kanten wie folgt bestimmt sind: Sofern eine Transaktion T_i eine Sperre auf ein Objekt, das derzeit von T_j gesperrt ist, anfordert, dann wird eine Kante vom Knoten T_i zum Knoten T_j gezogen. Die Kante wird entfernt, sobald T_j die Sperre aufhebt. Ein Zyklus im Graphen zeigt eine Verklemmung an.

Zur Auflösung einer Verklemmung wird eine der verklemmten Transaktionen zurückgesetzt (`ROLLBACK`) – eventuell mit Meldung an den Benutzer – und später wieder neu gestartet.



Strategie mit U-Locks: Vor Beginn fordert jede Transaktion für alle Objekte, die gelesen werden sollen, ein S-Lock, und für alle Objekte, die aktualisiert werden sollen, ein U-Lock an. Während einer Transaktion werden dann vor dem Aktualisieren die entsprechenden X-Locks angefordert. Ansonsten wird das Zwei-Phasen-Sperr-Protokoll (siehe unten) befolgt. Diese Strategie vermindert die Wahrscheinlichkeit von Deadlocks – insbesondere im Falle von *einem* Schreiber und *mehreren* konkurrierenden Lesern – und erhöht die Nebenläufigkeit gegenüber der Voranforderungsstrategie. Auch hier gibt es die Problematik, dass von vornherein oft unklar ist, was zu sperren ist.

22.2.3 Serialisierbarkeit

Serialisierbarkeit ist ein Kriterium, das die Korrektheit von nebenläufigen Transaktionen bestimmt. Das Kriterium lautet:

- *Eine verschachtelte Durchführung **(schedule)** von mehreren korrekten Transaktionen ist dann korrekt, wenn sie serialisierbar ist, d.h. wenn sie – für alle erlaubten Ausgangswerte (korrekte Datenbankzustände vor Durchführung der Transaktionen) – dasselbe Resultat liefert wie mindestens **eine** nicht verschachtelte (serielle) Durchführung der Transaktionen.*

Bemerkungen:

1. Individuelle Transaktionen sind korrekt, wenn sie die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführen. Daher ist die serielle Durchführung von korrekten Transaktionen auch wieder korrekt.
2. Es mag serielle Durchführungen korrekter Transaktionen geben, die unterschiedliche Resultate – insbesondere unterschiedlich zur verschachtelten Durchführung – liefern. Es kommt darauf an, dass *eine* serielle Durchführung dasselbe Ergebnis liefert.

   Betrachten wir die Transaktion A mit den Schritten

   ```
   Lies x
   Addiere 1 zu x
   Schreib x
   ```

   und die Transaktion B:

   ```
   Lies x
   Verdopple x
   Schreib x
   ```

   Wenn `x` anfänglich 10 war, dann ist `x` = 22 nach AB (d.h. erst A, dann B) und nach BA `x` = 21. Beide Serialisierungen sind korrekt, liefern aber unterschiedliche Datenbankzustände.
3. Die Anzahl der möglichen Verschachtelungen ist "astronomisch". Bei zwei Transaktionen zu je 2 atomaren Anweisungen sind es 6, bei je 3 Anweisungen 20, bei je 4 Anweisungen 70, bei je 5 Anweisungen 252 Verschachtelungen. Bei drei Transaktionen zu je 1 atomaren Anweisungen sind es 6, bei je 2 Anweisungen 90, bei je 3 Anweisungen 1680, bei je 4 Anweisungen 34650, bei je 5 Anweisungen 756756 Verschachtelungen.

Das **Zwei-Phasen-Sperrtheorem** ***(two-phase-locking theorem)*** lautet:

- *Wenn alle Transaktionen das Zwei-Phasen-Sperrprotokoll beachten, dann sind alle möglichen verschachtelten Durchführungen serialisierbar, wobei das Zwei-Phasen-Sperrprotokoll folgendermaßen aussieht:*
  1. *Bevor eine Transaktion ein Objekt benützt, muss sie eine Sperre auf das Objekt setzen.*
  2. *Nach der Freigabe einer Sperre darf eine Transaktion keine weiteren Sperren setzen (auch keine Upgrade-Locks).*

Mit dem Abschluss einer Transaktion durch `COMMIT` oder `ROLLBACK` – und damit automatische Freigabe aller Locks – erzwingt SQL automatisch mindestens (eigentlich mehr als) dieses Protokoll. Diese stärkere Forderung hat zur Folge, dass es nicht zu **kaskadierenden** Abbrüchen kommt.

Es gibt aber Systeme, die die Befolgung dieses Protokolls freistellen.

22.2.4 Isolationsniveau *(Level of Isolation)*

Isolationsniveau ist der Grad von Interferenz durch andere Transaktionen, den eine Transaktion toleriert. Wenn Serialisierbarkeit garantiert werden soll, dann sollte überhaupt keine Interferenz möglich sein. Es gibt aber pragmatische Gründe (Erhöhung des Grades der Nebenläufigkeit) eine gewisse Interferenz zu gestatten. SQL definiert vier Niveaus, die mit dem Statement

```
SET TRANSACTION ISOLATION LEVEL
   SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED;
```

eingestellt werden können. Default ist `SERIALIZABLE`. Wenn alle Transaktionen dieses Niveau haben, dann sind die Transaktionen serialisierbar. Durch die anderen, in der angegebenen Reihenfolge schwächeren Optionen wird die Serialisierbarkeit nicht mehr gewährleistet.

`REPEATABLE READ` ermöglicht das Lesen von *Phantomen*: Transaktion T_1 liest eine Anzahl Tupel, die eine gewisse Bedingung erfüllen. Transaktion T_2 fügt ein neues Tupel ein, das derselben Bedingung genügt. T_1 wiederholt das Lesen dieser Tupel und liest jetzt ein Tupel – das Phantom – mehr.

`READ COMMITTED` erlaubt zusätzlich nicht wiederholbares Lesen ***(nonrepeatable read)***: T_1 liest ein Tupel. T_2 liest dasselbe Tupel, aktualisiert dieses Tupel und macht ein Commit. T_1 liest dieses Tupel wieder, hat aber jetzt zwei verschiedene Werte gesehen.

`READ UNCOMMITTED` erlaubt zusätzlich "schmutziges" Lesen ***(dirty read)***: T_1 aktualisiert ein Tupel. T_2 liest dieses Tupel. T_1 terminiert mit einem `ROLLBACK`. T_2 hat ein Tupel gesehen, das nie so existiert hat.

Reale Systeme verwenden diese Niveaus manchmal in anderer Bedeutung.

Folgende Tabelle fast das zusammen:

	Typ der Verletzung der Serialisierbarkeit		
Isolationsniveau	***dirty read***	***nonrepeatable read***	***phantoms***
`READ UNCOMMITTED`	ja	ja	ja
`READ COMMITTED`	nein	ja	ja
`REPEATABLE READ`	nein	nein	ja
`SERIALIZABLE`	nein	nein	nein

22.2.5 Sperrgranularität *(Degree of Granularity)*

Die Größe der Dateneinheit, die gesperrt werden kann, wird mit **Granularität** bezeichnet. Folgende sperrbare Einheiten sind bekannt:

- die gesamte Datenbank
- eine Relation
- ein physischer Plattenblock

- ein Tupel
- ein Attributwert

Sind die Einheiten groß, dann ist der Systemaufwand (Verwaltung der Sperren) für die Nebenläufigkeit klein; aber auch der Grad der Nebenläufigkeit ist klein. Bei kleinen Einheiten wird der Systemaufwand groß; dafür wächst auch der Grad der Nebenläufigkeit.

Durch weitere Sperrmechanismen ***(intent locking protocol)*** kann der Systemaufwand einigermaßen beherrscht werden.

`X`-Locks und `S`-Locks haben wir schon kennengelernt. Sie können auf einzelne Tupel und ganze Relationen angewendet werden. Die zusätzlichen Locks machen nur Sinn für Relationen (R):

`IS` *(intent shared)* : Transaktion T beabsichtigt `S`-Locks auf einzelne Tupel von R zu setzen, um einzelne Tupel in R zu lesen.

`S` *(shared)* : Transaktion T erlaubt konkurrierende Leser für R. Aktualisierende Transaktionen sind in R nicht erlaubt. T selbst wird keine Aktualisierungen von R vornehmen.

`IX` *(intent exclusive)* : Transaktion T beabsichtigt `X`-Locks auf einzelne Tupel von R zu setzen.

`SIX` *(shared intent exclusive)* : Transaktion T erlaubt für R Leser, die beabsichtigen zu lesen. Lesende und aktualisierende Transaktionen sind in R nicht erlaubt. Aber T selbst wird einzelne Tupel von R aktualisieren und `X`-Locks auf einzelne Tupel von R setzen.

`X` *(exclusive)* : T erlaubt überhaupt keinen konkurrierenden Zugriff auf R. T selbst wird eventuell einzelne Tupel von R aktualisieren.

Die Kompatibiltätsmatrix hat folgende Form:

	X	SIX	IX	S	IS	–
X	nein	nein	nein	nein	nein	ja
SIX	nein	nein	nein	nein	ja	ja
IX	nein	nein	ja	nein	ja	ja
S	nein	nein	nein	ja	ja	ja
IS	nein	ja	ja	ja	ja	ja
–	ja	ja	ja	ja	ja	ja

Formulierung des (intent) Sperrprotokolls:

- Bevor eine Transaktion ein `S`-Lock auf ein Tupel setzen kann, muss sie erst ein `IS`-Lock oder stärkeres Lock auf die entsprechende Relation setzen.
- Bevor eine Transaktion ein `X`-Lock auf ein Tupel setzen kann, muss erst ein `IX`-Lock oder stärkeres Lock auf die entsprechende Relation setzen.

Da SQL diese Mechanismen nicht explizit unterstützt, wird implizit gesperrt. D.h., wenn eine Transaktion nur liest, wird für jede betroffene Relation ein `IS`-Lock implizit gesetzt. Für eine aktualisierende Transaktion werden für die betroffenen Relationen `IX`-Locks gesetzt.

Manche Systeme arbeiten mit ***lock escalation***: Wenn der Aufwand für das Sperren zu groß wird, dann werden z.B. automatisch individuelle `S`-Locks auf einzelnen Tupeln ersetzt durch ein `S`-Lock auf der ganzen Relation (Konvertierung eines `IS`-Locks zu einem `S`-Lock).

22.3 Transaktions-Modell der ODMG

Es werden die S-, U- und X-Locks unterstützt. Alle Locks werden bis zum Ende einer Transaktion gehalten. Die Locks werden implizit beantragt, können aber auch explizit über Methoden der Objekte (`lock` und `try_lock`) beantragt werden. Das Upgrade-Lock kann nur explizit beantragt werden.

Jeder Zugriff (Erzeugen, Löschen, Aktualisieren, Lesen) auf persistente Objekte muss innerhalb einer Transaktion erfolgen ***(within the scope of a transaction)***.

Innerhalb eines Threads kann es nur eine lineare Folge von Transaktionen geben. D.h. ein Thread kann höchstens eine **aktuelle** Transaktion haben. Aber ein Thread kann eine Transaktion verlassen und eine andere betreten. Eine Transaktion bezieht sich auf genau eine DB, die allerdings eine verteilte DB sein kann.

Eine Transaktion kümmert sich nicht um transiente Objekte. D.h. transiente Objekte werden beim Abbruch einer Transaktion nicht automatisch wiederhergestellt.

In der Schnittstelle `Transaction` sind die Methoden definiert, mit denen eine Transaktion manipuliert werden kann:

```
interface   Transaction
   {
   void     begin ()       raises (TransactionInProgress, DatabaseClosed);
   void     commit ()      raises (TransactionNotInProgress);
   void     abort ()       raises (TransactionNotInProgress);
   void     checkpoint ()  raises (TransactionNotInProgress);
   void     join ()        raises (TransactionNotInProgress);
   void     leave ()       raises (TransactionNotInProgress);
   boolean  isOpen ();
   };
```

Transaktionen werden erzeugt mit Hilfe einer `TransactionFactory`, die Methoden `new ()` und `current ()` zur Verfügung stellt.

```
interface   TransactionFactory
   {
   Transaction new ();
   Transaction current ();
   };
```

Nach der Erzeugung einer Transaktion ist sie zunächst geschlossen. Sie muss explizit mit der Methode `begin ()` geöffnet werden. Die Operation `commit ()` sorgt dafür, dass alle in der Transaktion erzeugten oder modifizierten Objekte in die Datenbank geschrieben werden. Nach einem `abort ()` wird die DB in den Zustand vor der Transaktion zurückgesetzt. Nach einem `commit ()` oder `abort ()` werden alle Locks freigegeben. Die Transaktion wird geschlossen.

Die Operation `checkpoint ()` ist äquivalent zu einem `commit ()` gefolgt von einem `begin ()`, außer dass alle Locks behalten werden. D.h. es kommt dabei zu permanenten Änderungen an der DB, die von einem späteren `abort ()` nicht mehr rückgängig gemacht werden können.

Damit eine Thread auf persistente Objekte zugreifen kann, muss sie erst mit einer Transaktion assoziiert werden. Das erfolgt entweder implizit durch das `begin ()` oder explizit durch ein `join ()`. Mit `join ()` kann eine Thread eine aktive Transaktion betreten. Ist die Thread schon mit einer Transaktion assoziiert, wird vorher automatisch ein `leave ()` für diese Transaktion durchgeführt. Ein Thread kann höchstens mit *einer* Transaktion assoziiert sein. Beim Wechsel von einer Transaktion zu einer anderen wird kein `commit ()` oder `abort ()` gemacht.

Mehrere Threads eines Adressraums können mit derselben Transaktion assoziiert sein. Zwischen diesen Threads gibt es kein Lockmechanismus. Dafür muss der Programmierer sorgen.

22.3.1 Java Binding

Unter Java sind Transaktionen Objekte von Klassen, die die Schnittstelle `Transaction` implementieren.

```
package  kj.ovid.odmg;

public interface  Transaction
   {
   public void begin ()
      throws TransactionInProgressException, DatabaseClosedException;
   public void commit ()
      throws TransactionNotInProgressException;
   public void abort ()
      throws TransactionNotInProgressException;
   public void checkpoint ()
      throws TransactionNotInProgressException;
   public void join ()
      throws TransactionNotInProgressException;
   public void leave ()
      throws TransactionNotInProgressException;
   public boolean isOpen ();
   public void lock (Object obj, int mode)
      throws LockNotGrantedException;
   public boolean try_lock (Object obj, int mode);
   public static final int READ = 1;
   public static final int UPGRADE = 2;
   public static final int WRITE = 4;
   }
```

Bevor eine Thread irgendeine Datenbankoperation macht, braucht sie ein Transaktionsobjekt, wofür sie `begin ()` oder `join ()` aufruft, jenachdem, ob sie das Objekt selbst erzeugt hat oder nicht. Das ist dann ihre aktuelle ***(current)*** Transaktion, und nur auf der kann sie Operationen ausführen. Sonst werden die entsprechenden Exceptions geworfen.

Aber mehrere Threads können unter derselben Transaktion laufen. Allerdings muss in diesem Fall der Programmierer selbst für die Nebenläufigkeits-Kontrolle sorgen.

Ein `Transaction`-Objekt wird erzeugt, indem für das `Implementation`-Objekt die Methode

```
newTransaction ()
```

aufgerufen wird.

(Auf das `Implementation`-Objekt gehen wir hier nicht weiter ein. Es bildet den Hersteller-abhängigen Einstieg in das Java-Binding.)

Da Java selbst eine Wurzelklasse `Object` hat, kann dieser Name nicht mehr für das ODMG-`Object`-Interface verwendet werden. Beim Java-Binding verzichtet man daher ganz auf diese Schnittstelle und verteilt die dort definierten Operationen auf verschiedene andere Klassen. Für die Operation `copy ()` wird das `clone ()` in `Object` verwendet. `delete ()` taucht in der Schnittstelle `Database` als `deletePersistent (Object o)` auf. `same_as (in Object anObject)` habe ich noch nicht gefunden. Die Lock-Operationen sind nun in der Schnittstelle `Transaction` untergebracht.

Transktions-Objekte können nicht persistent gemacht werden. "Lange" Transaktionen sind daher (bisher) nicht möglich.

22.4 Übungen

22.4.1 U-Locks

Zeigen Sie, dass es bei *lost update* und bei *inconsistent analysis* nicht zu einer Verklemmung kommt, wenn die Strategie mit U-Locks verwendet wird.

22.4.2 Serialisierbarkeit

Ist folgende verschachtelte Durchführung `V1` von zwei Transaktionen `T1` und `T2` serialisierbar? (Die Anfangswerte seien `x = 10` und `y = 5`)

```
T1: Liest x
T2: Liest y
T2: Liest x
T1: x = x + 7
T2: if (x > 12) then x = x + y
T2: Schreibt x
T1: Schreibt x
```

22.4.3 Zwei-Phasen-Sperrtheorem

Wird von den folgenden beiden Transaktionen das Zwei-Phasen-Sperrtheorem eingehalten.

```
T1.Slock (x)
T1.read (x)
T1.x = x + 7
T1.Xlock (x)
T1.write (x)
T1.free (x)

T2.Slock (y)
T2.read (y)
T2.Slock (x)
T2.read (x)
T2.free (x)
T2.if (x > 12) then x = x + y
T2.Xlock (x)
T2.write (x)
T2.free (y)
T2.free (x)
```

Kapitel 23

Wiederherstellung

Um die Wiederherstellung ***(recovery)*** einer Datenbank zu ermöglichen, führt das DBMS ein **Journal** ***(log)***, dessen **aktiver** Teil normalerweise auf einer Platte und dessen **Archiv**teil auf einem Band gespeichert wird. Insbesondere werden in dem Journal alte und neue Werte eines aktualisierten Objekts festgehalten. Der aktive Teil wird regelmäßig in den Archivteil überführt.

Wir unterscheiden Recovery nach der Art des aufgetretenen Fehlers:

- ***Transaction Recovery*** bei Transaktionsfehlern
- ***System Recovery*** bei Systemabsturz ohne Schädigung der Speichermedien
- ***Media Recovery*** bei Hardware-Fehler des Speichermediums

23.1 Transaction Recovery

In diesem Abschnitt beschreiben wir das Recovery bei Fehlern, die während einzelner Statements einer Transaktion auftreten und die nicht zum Absturz des Systems führen. Wir nennen diese Fehler **Transaktionsfehler**. Solche Fehler können z.B. sein, daß ein Tupel nicht gefunden wird, oder daß versucht wird, ungültige Daten einzutragen, oder daß es zu einer Verklemmung kommt. Diese Fehler können vom DBMS oder vom Anwendungsprogrammierer erkannt werden, die dann eventuell ein Rollback der Transaktion veranlassen.

Ein `COMMIT`- oder `ROLLBACK`-Statement bedeutet Etablierung eines **Syncpoints** oder ***commit point***, d.h. eines Zeitpunkts, zu dem die Datenbank in einem konsistenten Zustand ist.

Bei einem `COMMIT` wird ein *neuer* Syncpoint etabliert. Dabei werden alle Änderungen an der Datenbank seit dem letzten Syncpoint permanent gemacht. Alle Tupel-Locks werden freigegeben. Tupeladressierbarkeit geht verloren (Kursoren). Aber es gibt Implementationen, wo die Tupeladressierbarkeit erhalten werden kann. Die Transaktion wird beendet.

Ein `ROLLBACK` bedeutet: zurück zum vorhergehenden Syncpoint.

In dem Beispiel zu Beginn des letzten Kapitels hat die ProgrammiererIn einige Fehler abgefangen, sodaß die Transaktion normalerweise immer an ihr "natürliches" Ende läuft (eventuell auch mit

einem `ROLLBACK`). Die ProgrammiererIn kann aber nicht alle Fehler abfangen. Das DBMS wird daher ein implizites `ROLLBACK` veranlassen, wenn die Transaktion nicht an ihr natürliches Ende kommt (etwa auf Grund eines Systemabsturzes).

23.2 System Recovery

System Recovery bedeutet die Behandlung von globalen Systemfehlern, die die Datenbank nicht physikalisch beschädigen ***(soft crash)***. Das sind meistens Softwarefehler, aber auch Stromausfälle.

Wir müssen von der Forderung ausgehen, daß nach einem `COMMIT` die Änderungen permanent sind. Es kann aber zu einem Systemabsturz gleich nach dem `COMMIT` kommen, wenn die Änderungen eventuell noch in einem Hauptspeicherpuffer stehen. In solch einem Fall wird die Restart-Prozedur des DBMS die Transaktion mit Hilfe des Journals wiederholen und das Resultat auf die Datenbank schreiben. Das bedeutet, daß der Journaleintrag vor dem `COMMIT` erfolgen muß und physikalisch gespeichert werden muß ***(write-ahead log rule)***.

Bei einem Systemabsturz geht der Inhalt des Hauptspeichers verloren. Der Zustand einer während des Absturzes laufenden Transaktion ist daher nicht mehr bekannt. Sie kann also nicht mehr zu Ende geführt werden. Als einzige Möglichkeit bleibt ein `ROLLBACK`.

Nach einem Systemabsturz müssen einige Transaktionen wiederholt werden, andere müssen ein Rollback bekommen. Woher weiß das System, welche Transaktionen wiederholt werden müssen, für welche ein `ROLLBACK` durchgeführt werden muß? Das DBMS setzt in regelmäßigen Abständen (z.B. Anzahl Journaleinträge) **Checkpoints**, wo die Datenbankpuffer physikalisch auf das Speichermedium geschrieben werden.

t_c sei der Zeitpunkt des letzten Checkpoints, t_f sei der Zeitpunkt eines Systemabsturzes. Wir können fünf verschiedene Transaktionszustände mit den notwendigen bzw möglichen Recovery-Maßnahmen unterscheiden:

1. Transaktion T_1 wurde vor t_c beendet.
   Recovery : Nicht nötig.

2. Transaktion T_2 wurde vor t_c gestartet und zwischen t_c und t_f beendet.
   Recovery : Muß wiederholt werden.

3. Transaktion T_3 wurde vor t_c gestartet und nicht vor t_f beendet.
   Recovery : `ROLLBACK`

4. Transaktion T_4 wurde nach t_c gestartet und vor t_f beendet.
   Recovery : Muß wiederholt werden.

5. Transaktion T_5 wurde nach t_c gestartet und nicht vor t_f beendet.
   Recovery : `ROLLBACK`

Bemerkung: Manche Systeme erlauben die dynamische Einstellung von weniger sicheren Recovery-Stufen, um etwa die Performanz bei Bulk-Operationen zu erhöhen.

23.3 Media Recovery

Media Recovery bedeutet die Behandlung von Fehlern, die das physikalische Speichermedium der Datenbank betreffen ***(hard crash)***. Typisches Beispiel ist ein Head-Crash.

Die Wiederherstellung der Datenbank bei solch einem Fehler besteht im **Laden** ***(reloading, restoring)*** der Datenbank von einem Backup-Medium und Benutzung des Journals, um alle Transaktionen seit des Backups zu wiederholen. Voraussetzung ist natürlich, daß die Datenbank regelmäßig **entladen** ***(unload, dump)*** wird und das Journal unbeschädigt ist.

Kapitel 24

Sicherheit

Die Begriffe **Sicherheit** ***(security)*** und **Integrität** ***(integrity)*** beziehen sich auf zwei Aspekte des Schutzes der Daten einer Datenbank:

Sicherheit von Daten bedeutet, dass nur *berechtigte (privileged, authorized)* DB-Benutzer auf Daten zugreifen können. Die Berechtigungen können von der Art der Daten abhängen.

Integrität von Daten bedeutet, dass Datenmanipulationen *korrekt* sind. Integrität wird im nächsten Kapitel behandelt.

Sicherheit von Daten berührt viele Problemkreise:

- juristisch (Datenschutz): Welche Personen sind legitimiert bestimmte Informationen zu kennen und zu manipulieren?
- organisatorisch: Wer entscheidet in einer Firma über den Zugang zu Daten?
- technisch: Können gewisse Hardware-Komponenten sicher untergebracht werden? Wie werden Passwörter geheimgehalten? Unterstützt das Betriebssystem ein Sicherheitskonzept.
- DB-spezifische Fragen der Sicherheit.

Wir beschäftigen uns hier nur mit DB-spezifischen Fragen der Sicherheit. Bei DBS kann man zwei Ansätze unterscheiden:

1. ***discretionary access control:*** Vergabe von **Berechtigungen** (Lesen, Schreiben) für bestimmte Datenobjekte an bestimmte Benutzer ***(privileges, authorities)***
2. ***mandatory access control:*** Vergabe von unterschiedlichen **Sicherheitsstufen** an Benutzer ***(clearance level)*** und Datenobjekte ***(classification level)***. Diese statische Art der Kontrolle findet man bei großen Organisationen (Großfirmen, Miltär, öffentliche Verwaltung).

Ein gutes DBMS bietet *audit-trail*-Möglichkeiten. Für alle Transaktionen kann nachvollzogen werden, wer sie wann und von wo aus durchgeführt hat.

24.1 SQL2 Implementation

SQL2 kennt nur eine Zugangskontrolle über Berechtigungen für bestimmte Datenobjekte *(discretionary access control)*. **Sicherheitsregeln *(security rules)*** sind die Definitionskomponente dieser Kontrolle.

In SQL2 ist die Form einer Sicherheitsregel:

security-rule

```
     ::=  GRANT privilege-commalist
            ON { [TABLE] table | DOMAIN domain }
            TO { user-commalist  | PUBLIC} [ WITH GRANT OPTION ]
          |
          REVOKE [ WITH GRANT OPTION ] privilege-commalist
            ON { [TABLE] table | DOMAIN domain }
            FROM { user-commalist  | PUBLIC} { RESTRICT | CASCADE };
```

privilege

```
     ::=    USAGE
          | SELECT
          | INSERT [ (column-commalist) ]
          | UPDATE [ (column-commalist) ]
          | DELETE
          | REFERENCES
```

Beispiel:

```
CREATE VIEW SR3 AS
   SELECT   S.SNR, S.SNAME, S.CITY
      FROM  S
      WHERE S.CITY <> 'London';
GRANT SELECT, UPDATE (S.SNAME), DELETE
   ON SR3
   TO Pschorr, Salvator WITH GRANT OPTION;
REVOKE WITH GRANT OPTION DELETE
   ON SR3
   FROM  Pschorr;
```

Erklärungen:

1. Das `USAGE`-Privileg wird benötigt, um die Benutzung eines Wertebereichs zu erlauben.
2. Nur die Privilegien `INSERT` und `UPDATE` erlauben die Spezifizierung von Spalten.
3. `REFERENCES` erlaubt die Verwendung einer Tabelle in einer Integritätsbedingung.
4. `TABLE` *table* kann eine Basistabelle oder ein View sein.
5. `PUBLIC` bedeutet alle dem System bekannte Benutzer.

6. `WITH GRANT OPTION` erlaubt den angegebenen Benutzern ihrerseits die spezifizierten Privilegien weiterzugeben.

7. `REVOKE` löscht Privilegien. Damit es nicht zu "herrenlosen" Privilegien kommt, muss `RESTRICT` oder `CASCADE` angegeben werden. Wenn A ein Privileg B gegeben hat, und B dies Privileg an C weitergegeben hat, dann wird eine Löschung des Privilegs für B durch A zurückgewiesen, wenn `RESTRICT` spezifiziert war, oder auch C verliert das Privileg, wenn `CASCADE` angegeben war.

8. Views können sehr gut verwendet werden, um Information zu verstecken.

9. Löschung einer Tabelle, eines Views, einer Spalte oder eines Wertebereichs entfernt alle Privilegien auf diese Objekte für alle der Datenbank bekannte Benutzer.

24.2 Datenverschlüsselung

Datenverschlüsselung **(Kryptographie)** ist eine Möglichkeit, um kriminellen Zugang (Umgehung des normalen DB-Zugangs) zu den Daten zu verhindern oder wesentlich zu erschweren.

Datenverschlüsselung bedeutet, dass Daten in verschlüsselter Form gespeichert und übertragen werden.

Die unverschlüsselte Form der Daten ist der *Klartext (plaintext)*, der mit Hilfe eines Schlüssels *(encryption key)* und eines Verschlüsselungsalgorithmus *(encryption algorithm)* verschlüsselt wird. Die verschlüsselte Form heißt *Geheimtext (ciphertext)*. Sie wird eventuell mit einem anderen Schlüssel *(decryption key)* und einem Entschlüsselungsalgorithmus entschlüsselt.

Es kommt darauf an, dass die Kosten für die Entschlüsselung durch Unbefugte höher sind als mögliche Gewinne. Aus der Kenntnis von Klartext, Geheimtext und Verschlüsselungsalgorithmus darf es nicht leicht möglich sein, den Schlüsel zu bestimmen.

DES *(Data Encryption Standard)* ist ein Standard-Algorithmus, der auf dem symmetrischen oder ***secret key*** Verfahren beruht, wo jeder Kommunikationspartner über denselben, geheimen Schlüssel verfügt. Der von belgischen Wissenschaftlern entwickelte "Rijndael"-Algorithmus wird als Verschlüsselungsstandard ***Advanced Encryption Standard* (AES)** die Nachfolge von DES antreten.

Asymmetrische Verfahren beruhen darauf, dass zwei Schlüssel verwendet werden: ein öffentlich bekannter zum Verschlüsseln und ein geheimer Schlüssel zum Entschlüsseln. Als Algorithmus wird **RSA** nach Rivest, Shamir und Adleman verwendet.

Asymmetrische Verfahren sind sehr aufwendig. Nur kurze Botschaften können damit übertragen werden. Symmetrische Verfahren dagegen sind sehr performant und erlauben daher die Übertragung großer Datenmengen. Allerdings ist der Austausch des geheimen Schlüssels ein Problem. In der Praxis wird daher folgendes **hybride** Verfahren verwendet:

1. Mit einem asymmetrischen Verfahren wird der für das symmetrische Verfahren benötigte geheime Schlüssel ausgetauscht.

2. Anschließend werden mit einem symmetrischen Verfahren die Daten übertragen.

Beispiel: A möchte Daten nach B verschlüsselt übertragen.

1. A meldet diese Absicht unverschlüsselt bei B an.
2. Asymmetrisches Verfahren zum Schlüsselaustausch:
   (a) B denkt sich einen geheimen Schlüssel GK und einen öffentlichen Schlüssel PK aus.
   (b) B schickt A unverschlüsselt den öffentlichen Schlüssel PK.
   (c) A denkt sich einen geheimen Schlüssel SK aus.
   (d) A verschlüsselt SK mit einem asymmetrischen Verfahren unter Verwendung von PK.
   (e) A schickt den verschlüsselten SK an B.
   (f) B entschlüsselt nach demselben asymmetrischen Verfahren den verschlüsselten SK unter Verwendung von GK.
3. Übertragung der Daten unter Verwendung eines symmetrischen Verfahrens:
   (a) A verschlüsselt die Daten mit einem symmetrischen Verfahren unter Verwendung von SK.
   (b) A überträgt die verschlüsselten Daten.
   (c) B entschlüsselt die Daten mit demselben symmetrischen Verfahren unter Verwendung von SK.

Kapitel 25

Integrität

Integrität ***(integrity)*** einer DB bezieht sich auf ihre Korrektheit, die – im Unterschied zur Sicherheit – unabhängig vom Benutzer ist.

Eine DB speichert Informationen über einen Teil der realen Welt (**Miniwelt** oder ***universe of discourse)***. Gewisse Regeln ***(integrity constraints)*** oder Geschäftsregeln ***(business rules)*** beherrschen diese Miniwelt.

Analyse und Design beinhaltet das Auffinden und die Definition dieser Regeln.

Zur Gewährleistung der Integrität hat das DBMS verschiedene Arten von Integritätsregeln zu beachten. Man unterscheidet:

- Wertebereichsintegritätsregeln
- Datenbank-Integritätsregeln
- Basistabellen-Integritätsregeln
- Spalten-Integritätsregeln

Ferner unterscheiden wir **Zustands- und Transaktionsintegritätsregeln** ***(state and transaction constraints)***. Zustandsintegritätsregeln sollen die Korrektheit einer konkreten DB vor und nach jeder Transaktion gewährleisten. Transaktionsintegritätsregeln berücksichtigen mehrere Zustände einer DB; z.B. "das Gehalt eines Angestellten soll nur steigen". Zur Prüfung muss der Zustand vor und nach einer Transaktion betrachtet werden.

In SQL2 gibt es dafür keine Sprachkonstrukte. Transaktionsintegritätsregeln müssen explizit in eine Transaktion hineinprogrammiert werden.

In SQL2 sind diese Regeln folgendermaßen definierbar:

Wertebereichsintegritätsregel : Die Verwendung von Wertebereichen an sich bedeutet eine Integritätsregel. Sie spezifizieren den Typ und die möglichen Werte eines Attributs. Ein Wertebereich kann mit verschiedenen benannten oder nicht benannten Wertbeschränkungen definiert werden.

Syntax:

domain-definition
::= `CREATE DOMAIN` *domain* `[ AS ]` *data-type*
`[` *default-definition* `]`
`[` *domain-constraint-definition-list* `]`

domain-constraint-definition
::= `[ CONSTRAINT` *constraint* `] CHECK (` *conditional-expression* `)`

und mit `ALTER DOMAIN` haben wir die Möglichkeit Integritätsregel hinzuzufügen oder fallen zu lassen:

domain-constraint-alteration-action
::= `ADD` *domain-constraint-definition*
| `DROP CONSTRAINT` *constraint*

Beispiele:

```
CREATE DOMAIN GESCHLECHT CHAR (1)
   CHECK ( VALUE IN ( 'm', 'w'));

CREATE DOMAIN STATUS INTEGER
   CONSTRAINT KEINENULL CHECK ( VALUE IS NOT NULL )
   CONSTRAINT MAXSTATUS CHECK ( VALUE <> 0 )
   CONSTRAINT MAXSTATUS CHECK ( VALUE <= 100 );
```

Datenbank-Integritätsregeln können unter Verwendung von Spalten verschiedener Tabellen definiert werden. Sie heißen auch *allgemeine* Integritätsregeln ***(assertions)***.

Syntax:

`CREATE ASSERTION` *constraint* `CHECK (` *conditional-expression* `);`

`DROP ASSERTION` *constraint*;

Beispiele:

1. Jeder Lieferant muss mindestens Status 5 haben:
```
CREATE ASSERTION REGEL1 CHECK
   ( (SELECT MIN(S.STATUS) FROM S) > 4);
```

2. Alle roten Teile müssen in London gelagert sein:
```
CREATE ASSERTION REGEL2 CHECK
   (NOT EXISTS (SELECT  *
                FROM  P
                WHERE P.COLOR = 'Red'
                      AND P.CITY <> 'London'));
```

3. Jeder liefernde Lieferant hat mindestens den Status 10:
```
CREATE ASSERTION REGEL3 CHECK
   (NOT EXISTS (SELECT  *
                FROM  S JOIN SP USING (SNR)
                WHERE S.STATUS < 10));
```

Nur `REGEL3` ist eine wirkliche DB-Integritätsregel, d.h. tabellenübergeifend.

Assertionen sind eine elegante Methode zur Spezifikation von Integritätsregeln. Es ist aber relativ schwierig, sie effektiv zu implementieren, so dass die Systeme eventuell sehr langsam werden.

Basistabellen-Integritätsregeln werden innerhalb der Definition einer zugehörigen Basistabelle definiert. Es gibt drei Möglichkeiten:

1. ***Schlüsseldefinitionen*** haben die Form:
   `[ CONSTRAINT` *constraint* `]`
   {`PRIMARY KEY | UNIQUE }` `(` *column-commalist* `)`
   Bei `UNIQUE` sind Nullen erlaubt, bei `PRIMARY KEY` nicht.
2. ***Fremdschlüsseldefinitionen*** haben die Form:
   `[ CONSTRAINT` *constraint* `]`
   `FOREIGN KEY (` *column-commalist* `)` *references-definition*
3. ***Check-Constraint-Definitionen*** haben die Form:
   `[ CONSTRAINT` *constraint* `] CHECK (` *conditional-expression* `)`
   Diese Bedingungen können auch als allgemeine Bedingungen definiert werden. Sie sind, wie folgende Beispiele zeigen, als Tabellenbedingungen häufig einfacher zu formulieren:
   (a) Jeder Lieferant muss mindestens Status 5 haben:
   ```
   CHECK (S.STATUS > 4)
   ```
   (b) Alle roten Teile müssen in London gelagert sein:
   ```
   CHECK (P.COLOR <> 'Red' OR P.CITY = 'London')
   ```

Mit `ALTER TABLE` können Bedingungen gelöscht und hinzugefügt werden.

Spalten-Integritätsbedingungen beziehen sich nur auf eine Spalte und können an die Definition der Spalte angehängt werden. Möglich ist:

1. `NOT NULL`
2. `PRIMARY KEY` oder `UNIQUE`
3. *references-definition*
4. *check-constraint-definition*

Bisher sind wir davon ausgegangen, dass die Integritätsregeln bei jedem SQL-Statement überprüft werden. Dies kann aber sehr hinderlich sein, wenn man z.B. sich gegenseitig referenzierende Tabellen verwendet. (Ein Verein muss ein Mitglied als Vorsitzenden haben und jedes Mitglied muss einem Verein angehören. Dann wäre "Verein" Fremdschlüssel (`NOT NULL`) in Tabelle "Mitglied" und "Mitglied" (als Vorsitzender) Fremdschlüssel (`NOT NULL`) in Tabelle "Verein". Die Tabellen "Verein" und "Mitglied" könnten nie gefüllt werden.)

Es gibt daher die syntaktische Möglichkeit, an verschiedenen Stellen einer Transaktion die Integrität zu überprüfen.

In SQL2 kann jede `CONSTRAINT`-Definition ergänzt werden durch

`INITIALLY IMMEDIATE NOT DEFERRABLE` (Default)
: Integritätsbedingung wird sofort bei der Ausführung eines SQL-Statements überprüft.

`INITIALLY IMMEDIATE DEFERRABLE`
: Mit `SET CONSTRAINTS` Statement (siehe unten) kann der Mode der Regel auf `IMMEDIATE` oder `DEFERRED` gesetzt werden.

`INITIALLY DEFERRED [ DEFERRABLE ]`
: Integritätsregeln werden nur am Ende der Transaktion überprüft.

`SET CONSTRAINTS {`***constraint-commalist*** `| ALL} {DEFERRED | IMMEDIATE}`
: Die genannten Constraints müssen `DEFERRABLE` sein. Das Statement kann vor oder während einer Transaktion gegeben werden.

Kapitel 26

Views

26.1 Einleitung

Ein *View* ist im wesentlichen ein mit einem *Namen* versehener Ausdruck der relationalen Algebra. In SQL2 lautet ein einfaches Beispiel:

```
CREATE VIEW SGUT AS
   SELECT   SNR, STATUS, CITY
      FROM  S
      WHERE STATUS > 15;
```

Dieses Statement wird zunächst nicht ausgewertet, sondern vom DBMS nur in den Katalog eingetragen. Für den Benutzer verhält sich `SGUT` wie eine normale Relation mit Attributen und Tupeln, nämlich wie die Relation, die entsteht, wenn der `SELECT`-Ausdruck ausgewertet wird. Da die Relation `SGUT` nicht durch reale Daten repräsentiert wird, sondern nur durch den `SELECT`-Ausdruck, spricht man von einer *virtuellen* Relation.

Ein View ist eine **Sicht** ***(window)*** auf reale Daten. Wenn sich die Daten ändern, indem die zugrundeliegenden Basisrelationen verändert werden, dann ändert sich auch der View. Umgekehrt wirken sich Änderungen am View automatisch auf die zugrundeliegenden Basisrelationen aus.

Anfragen und Manipulationen eines Views werden vom DBMS in Anfragen und Manipulationen der Basisrelationen umgesetzt.

Wozu werden Views benötigt? Die Basisrelationen sind eine logische Datenrepräsentation, die unabhängig ist von der *physikalischen* Repräsentation bzw Speicherung der Daten. Mit Views kann eine weitgehende Unabhängigkeit von der *logischen* – besser *konzeptuellen* – Struktur (repräsentiert durch Basisrelationen) der Datenbank erreicht werden. Änderungen an der konzeptuellen Struktur der Datenbank können für den Benutzer "weitgehend" transparent durchgeführt werden. Das betrifft folgende Änderungen der konzeptuellen Struktur:

Wachstum ***(growth)*** : Neue Attribute werden an bestehende Basisrelationen angefügt. Neue Basisrelationen werden angelegt.

Restrukturierung ***(restructuring)*** : Hierbei geht es um eine Umorganisation der Datenbank, wobei Attribute von einer Basisrelation in eine andere transferiert werden. Alte Basisrelationen werden eventuell in neue Basisrelationen zerlegt (z.B. weitergehende Normalisierung). Eine völlige Transparenz solcher Maßnahmen ist für den Benutzer nicht immer zu erreichen. Eine wichtige Voraussetzung ist, dass die beiden Versionen der Datenbank bezüglich des Informationsgehalts äquivalent sind.

Verschiedene Sichten : Verschiedene Benutzer können dieselben Daten unter verschiedenen Gesichtspunkten sehen.

Die Rechte zur Benutzung von Views werden unabhängig von den Basistabellen vergeben. D.h. um mit einem View zu arbeiten, muss eine BenutzerIn keinerlei Rechte auf den zugrunde liegenden Basistabellen haben. Es genügt, dass sie die entsprechenden Rechte auf dem View hat.

"Macro"-Eigenschaft : Manche Anfragen vereinfachen sich, wenn man vordefinierte Views verwendet. Die Frage nach "*Städten, in denen Teile gelagert sind, die ein Lieferant aus London liefert,*" lautet in SQL ohne View:

```
SELECT   P.CITY AS PCITY
   FROM  S JOIN SP USING (SNR)
         JOIN P USING (PNR)
   WHERE S.CITY = 'London';
```

Mit dem View

```
CREATE VIEW CITIES (PCITY, SCITY) AS
   SELECT   P.CITY, S.CITY
      FROM  S JOIN SP USING (SNR)
            JOIN P USING (PNR);
```

wird die Anfrage zu:

```
SELECT   PCITY
   FROM  CITIES
   WHERE SCITY = 'London';
```

Das lohnt sich natürlich nur, wenn derartige Anfragen häufig vorkommen oder die Anfrage wesentlich komplizierter ist.

Sicherheit : Indem man den Benutzer einer DB zwingt, die DB nur über Views zu verwenden, kann man die Sichtbarkeit und Manipulierbarkeit der Daten genau steuern.

26.2 SQL2-Syntax

Die allgemeine Syntax einer View-Definition lautet:

view-definition
::= `CREATE VIEW` *view* `[ (` *column-commalist* `) ]`
`AS` *table-expression*
`[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]`

view-destruction
::= `DROP VIEW` *view* { `RESTRICT | CASCADE` }

Bemerkungen: Wenn Spaltennamen angegeben werden, müssen alle angegeben werden. Es können neue Spaltennamen sein. Von einem View können andere Views oder Integritätsbedingungen abhängen. Daher muss bei `DROP VIEW` eine der Optionen `RESTRICT` oder `CASCADE` gegeben werden. Auf `WITH CHECK OPTION` wird unten eingegangen.

26.3 Aktualisierung von Views

Aktualisierung von Views ist ein weites Feld, das sich kaum zusammenfassen lässt. Wir beschränken uns daher hier auf wenige Bemerkungen.

Wichtigste Erkenntnis ist, dass ein View aktualisierbar ist oder auch nicht.

Der View `SGUT` ist aktualisierbar. D.h. Einfügung der Zeile (S6, 40, Rome) würde in der Basistabelle `S` die Zeile (S6, `DEFAULT`, 40, Rome) ergeben. Zeilen können gelöscht und aktualisiert werden. Was passiert aber, wenn wir die Zeile (S6, 10, Rome) einfügen wollen?

Der View CITIES ist nicht aktualisierbar. Dort gibt es die Zeile (London, London). Diese Zeile kann z.B. nicht gelöscht werden, da es keinen Satz von Aktualisierungen der Basistabellen gibt, die für den View genau das Resultat ergeben. Letztlich hängen diese Probleme damit zusammen, dass ein View nicht normalisiert ist, so dass sich Aktualisierungen oft auf mehr als ein Tupel auswirken.

Ob ein View theoretisch aktualisierbar ist oder nicht, ist eine schwierige Frage, die hier nicht beantwortet werden kann. Dazu wird auf die verschiedenen Veröffentlichungen von Date hingewiesen.

In SQL ist ein View aktualisierbar, wenn er eine Reihe – ziemlich einschränkender – Bedingungen erfüllt, auf die hier auch nicht eingegangen wird. Mit der `CHECK OPTION` wird das Verhalten eines aktualisierbaren Views geregelt in Fällen, die ähnlich dem Einfügen der Zeile (S6, 10, Rome) in `SGUT` sind. In dem Beispiel würde das Einfügen der Zeile (S6, 10, Rome) in `SGUT` bei Verwendung einer `CHECK OPTION` zurückgewiesen. Die Einzelheiten sind sehr kompliziert.

Resultat ist: Man erreicht das vernünftigste Verhalten von aktualisierbaren Views mit der Spezifikation:

```
WITH CASCADED CHECK OPTION
```

Kapitel 27

Basisalgorithmen für Datenbankoperationen

In diesem Kapitel geht es um Algorithmen von Datenbankoperationen und die Berechnung ihrer Kosten. Diese Daten werden für die Optimierung von Anfragen benötigt.

27.1 Voraussetzungen

27.1.1 Annahmen

Wir machen folgende vereinfachende Annahmen:

- Jede Relation liegt auf dem Sekundärspeicher.
- Jede Zwischenrelation wird auf dem Sekundärspeicher zwischengespeichert. (Bei realen DBMS wird allerdings der DB-Puffer einen großen Teil der Zwischenspeicherung übernehmen.) Die Zugriffe auf Zwischenrelationen werden als externe Speicherzugriffe behandelt.
- Operationen auf Adressmengen (TID-Listen) werden komplett im Hauptspeicher durchgeführt.
- Jeder Index ist ein B^+-Baum und liegt auf dem Sekundärspeicher.
- Der dominiernde Kostenfaktor ist der Zugriff auf den Sekundärspeicher. Ein Zugriff auf den Sekundärspeicher ist die "Kosteneinheit".

  Reine Hauptspeicheralgorithmen (ohne Zugriff auf den Sekundärspeicher) werden bei der Aufwandsabschätzung nicht berücksichtigt. In der Praxis ist es allerdings wichtig, dass auch diese Algorithmen effizient implementiert werden, da sie oft eingesetzt werden. Im wesentlichen gehören dazu:

  - Tupelvergleich zur Erkennung von Duplikaten
  - TID-Zugriff auf Seiten im DB-Puffer

- Der Zugriff auf einen Index liefert als Ergebnis entweder genau ein TID (Index über Schlüssel-Attribut) oder eine Menge von TIDs (Index über Nicht-Schlüssel-Attribut).
- Der Zugriff auf einen **geclusterten** Index liefert direkt ein oder mehrere Tupel.
- Die Transfereinheit zwischen DB-Puffer und Platte ist eine Seite (oBdA ≡ ein Block).

27.1.2 Definitionen

- Relationen bezeichnen wir mit

  $R, R_1, R_2, ...S, S_1, S_2, ...T...$

  und Attribute oder Attributmengen (mehrere Attribute) mit:

  $A, A_1, A_2, ...B, B_1, B_2, ...C...$

  Tupel bezeichnen wir mit den Kleinbuchstaben:

  $r, r_1, r_2, ...s, s_1, s_2, ...t...$

  Attributwerte bezeichnen wir mit den Kleinbuchstaben:

  $a, a_1, a_2, ...b, b_1, b_2, ...c...$

- $I_{A,R}$ bezeichnet einen Index über dem Attribut A in der Relation R. Nach den getroffenen Annahmen ist der Index ein B^+-Baum.

  $L = I_{A,R}(a)$ bezeichnet eine TID-Liste, die der Index für den Attributwert a liefert. Wenn A ein Schlüssel ist, dann enthält die TID-Liste genau ein Element.

  $L = I_{A,R}(a_1, a_2)$ bezeichnet eine TID-Liste, die der Index liefert, wenn wir den Index von indiziertem Attributwert a_1 bis a_2 durchlaufen.

- ρL ist die Menge der Tupel, die wir durch die **Realisierung** einer TID-Liste L erhalten.
- $I^c_{A,R}$ bezeichnet einen geclusterten Index. Die Tupel können wir ohne Realisierung aus dem Index lesen. $I^c_{A,R}(a_1, a_2)$ bezeichnet eine Tupel-Liste.
- $|X|$ bezeichnet die Kardinalität, d.h. die Anzahl der Elemente in der Menge X.
  $|R|$ bezeichnet bezeichnet daher die Anzahl der Tupel (Kardinalität) einer Relation R.
  $|I_{A,R}(a_1, a_2)|$ bezeichnet die Anzahl der TIDs (Kardinalität), die ein Zugriff auf einen Index liefert.
- b_s ist die Blockgröße *(block size)* des Datenbanksystems in Byte.
- b_m ist die Größe des Datenbankpuffers, d.h. ist die Anzahl der Rahmen im DB-Puffer, d.h. gleich der Anzahl der Seiten oder Blöcke, die der DB-Puffer aufnehmen kann.

  Oft versucht man, den Datenbankpuffer ganz mit den Tupeln einer Relation zu füllen, benötigt aber einen oder wenige Blöcke für das Ergebnis einer Operation oder eine zweite Relation in einer geschachtelten Schleife. Da b_m heutzutage sehr groß ist und unsere Schätzungen sowieso ziemlich grob sind, werden wir immer nur b_m anstatt eines genaueren $b_m - 1$ oder $b_m - 2$ verwenden.

- t_{s_R} ist die (mittlere) Größe von Tupeln der Relation R.

- b_R ist die Anzahl von Blöcken, die Tupel aus R enthalten.
- Das abgerundete Verhältnis

$$f_R = \left\lfloor \frac{b_s}{t_{s_R}} \right\rfloor$$

  wird als der Blockungsfaktor bezeichnet.

- Werden die Tupel nur *einer* Relation kompakt in Blöcken gespeichert, gilt der folgende Zusammenhang:

$$b_R = \left\lceil \frac{|R|}{f_R} \right\rceil$$

- $v_{A,R}$ ist die Anzahl der verschiedenen Werte für das Attribut A in der Relation R. Es gilt der Zusammenhang:

$$v_{A,R} = |\pi_A R|$$

  Wenn A ein Schlüssel ist, dann gilt: $v_{A,R} = |R|$
  Wenn A ein "ja/nein"-Attribut ist, dann gilt: $v_{A,R} = 2$
  Wenn $A = \{A_1, A_2 \dots A_n\}$, dann schätzen wir ab:

$$v_{A,R} \approx \min(\frac{|R|}{2}, \prod_{i=1}^{n} v_{A_i,R})$$

  Bemerkung: Da erfahrungsgemäß nicht alle Kombinationen der Attributwerte von A in R vorkommen, insbesondere wenn die $v_{A_i,R}$ groß sind, wird als maximaler Wert von $v_{A,R}$ die halbe Tupelanzahl $\frac{|R|}{2}$ angenommen.

- $l_{I_{A,R}}$ gibt die Anzahl der Indexebenen (Stufen) eines B^+-Baums für den Index $I_{A,R}$ über dem Attribut A in der Relation R an.
- $b_{I_{A,R}}$ gibt die Anzahl der Blöcke an, die für die Blätter eines Indexes benötigt werden.

Wir verzichten manchmal auf das "R", wenn klar ist, um welche Relation es sich handelt, oder wenn nur eine Relation im Spiel ist.

27.2 Scans

Scans sind Operationen, die eine komplette Relation oder einen Teil davon durchlaufen. Wir unterscheiden

Relationen-Scan ***(full table scan)***
Index-Scan ***(index scan)***
Geschachtelter Scan ***(nested scan)***

Die folgende Bereichsselektion wird als Beispiel der verschiedenen Scans verwendet.

```
SELECT *
   FROM S
   WHERE SNAME BETWEEN 'Mai' AND 'Mey';
```

27.2.1 Relationen-Scan

Alle Tupel einer Relation werden in beliebiger Reihenfolge durchlaufen. Die Reihenfolge hängt ab von der Pufferstrategie und der Verteilung der Tupel auf Blöcke.

Das notieren wir mit folgendem Pseudocode:

```
for r ∈ R
   show(r)
```

Dabei bedeutet `show` eine Weiterverarbeitung, im einfachsten Fall eine Anzeige.

Dieser Pseudocode kostet gewöhnlich:

$\rightarrow$ Kosten: b_R

Das oben genannte Beispiel wird mit dieser Notation zu:

```
for s ∈ S
   if "Mai" ≤ s.SNAME ≤ "Mey"
      show(s)
```

Die Kosten dieses Relationen-Scans sind offensichtlich:

$\rightarrow$ Kosten: b_S

27.2.2 Index-Scan

Die Tupel werden unter Verwendung eines Indexes in dessen Sortierreihenfolge ausgelesen. Dabei kann ein Bereich von Index-Attributwerten (von, bis) angegeben werden.

Der Aufwand hängt ab von der Anzahl Tupel im Scan-Bereich.

Wir verwenden folgenden Pseudocode:

```
for r ∈ ρI_{A,R}(a_1, a_2)
   show(r)
```

Diese Pseudocode-Zeile kostet gewöhnlich:

→ Kosten:

$$l_{I_{A,R}} + |I_{A,R}(a_1, a_2)| + \frac{|I_{A,R}(a_1, a_2)|}{|R|} b_{I_{A,R}}$$
$$= l_{I_{A,R}} + |I_{A,R}(a_1, a_2)| \left(1 + \frac{b_{I_{A,R}}}{|R|}\right)$$

Der dritte Term wird für das Lesen der Indexblätter angesetzt. Er kann oft vernachlässigt werden.

Der Term $|I_{A,R}(a_1, a_2)|$ kann bei numerischem Attribut mit $\frac{a_1 - a_2}{A_{\max} - A_{\min}}|R|$ abgeschätzt werden. Bei Strings kann eventuell eine Zeichenstatistik eingesetzt werden (siehe Abschnitt über Selektivität weiter unten).

Wenn der Bereich a_1, a_2 auf einen Wert a zusammenschrumpft, erhalten wir:

```
for r ∈ ρ I_{A,R}(a)
    show(r)
```

→ Kosten: $l_{I_{A,R}} + \frac{|R|}{v_{A,R}}$

Oft kommt es vor, dass wir nur die Werte des Indexfeldes benötigen. Wir können nur Bereiche oder alle Blätter scannen. Ein Scan durch alle Blätter in der Sortierreihenfolge des Indexfeldes wird folgendermaßen kodiert:

```
for a ∈ I_{A,R}
```

→ Kosten: $b_{I_{A,R}}$

a sind die Indexfeldwerte.

Mit

```
for t_id ∈ I_{A,R}
```

→ Kosten: $b_{I_{A,R}}$

bezeichnen wir einen Lauf durch alle Blätter, wobei t_{id} die jeweilige TID-Liste zum jeweiligen Indexfeldwert ist.

Wenn Bereiche noch angegeben werden, dann werden eben nur die Bereiche gescannt.

Bei einem geclusterten Index können wir die Tupel direkt aus den Indexblättern in der Sortier-Reihenfolge des Indexfeldes lesen:

```
for r ∈ I^c_{A,R}
```

→ Kosten: $b_{I^c_{A,R}}$

Beispiel normaler Index:

Wir nehmen an, dass es einen Namensindex $I_{\texttt{SNAME},S}$ in S gibt.

```
for s ∈ ρI_SNAME,S("Mai","Mey")
   show(s)
```

Diese Anweisung holt nur die Tupel, die der Selektionsbedingung genügen. Ferner muss einmal der Baum des Indexes traversiert werden. Die Kosten ergeben sich also zu:

→ Kosten:

$$l_{I_{\texttt{SNAME},\mathrm{S}}} + |I_{\texttt{SNAME},\mathrm{S}}(\texttt{"Mai", "Mey"})| \left(1 + \frac{b_{I_{\texttt{SNAME},\mathrm{S}}}}{|S|}\right)$$

Im allgemeinen ergibt sich durch die Verwendung eines Indexes ein Zeitgewinn.

Bei einem Schlüssel-Index (z.B. über `SNR`) und einer Single-Match-Query (exakte Suche) sind die

→ Kosten: $l_{I_{\texttt{SNR},\mathrm{S}}} + 1$

Bei einem Sekundärindex (z.B. über `SNAME`) und einer Single-Match-Query sind die

→ Kosten: $l_{I_{\texttt{SNAME},\mathrm{S}}} + \frac{|\mathrm{S}|}{v_{\texttt{SNAME},\mathrm{S}}}$

Beispiel geclusterter Index:

Wir nehmen an, dass es einen geclusterten Namensindex $I^c_{\texttt{SNAME},S}$ in S gibt.

```
for s ∈ I^c_SNAME,S("Mai","Mey")
   show(s)
```

Diese Anweisung findet die Tupel, die der Selektionsbedingung genügen, direkt in den Blättern des Indexes. Ferner muss einmal der Baum des Indexes traversiert werden. Die Kosten ergeben sich also zu:

→ Kosten:

$$l_{I^c_{\texttt{SNAME},\mathrm{S}}} + b_{I^c_{\texttt{SNAME},\mathrm{S}}} \frac{|I^c_{\texttt{SNAME},\mathrm{S}}(\texttt{"Mai", "Mey"})|}{|S|}$$

Halloween-Problem: Die Scan-Semantik muss bei Update-Operationen wohldefiniert sein, damit sich der Bereich nicht durch das Update verändert und eventuell zu unendlich vielen Updates führt.

27.2.3 Geschachtelter Scan

Bei einer geschachtelten Schleife

```
for r ∈ R
   for s ∈ S
      show (r‖s)
```

werden die Kosten dadurch drastisch gesenkt, dass der Datenbankpuffer nacheinander mit allen Blöcken der äußeren Relation gefüllt wird. Wie oft der Datenbankpuffer mit Tupeln einer Relation R gefüllt werden kann, bis die ganze Relation abgearbeitet ist, ist gegeben durch:

$\left\lceil \frac{b_R}{b_m} \right\rceil = \text{ceil}(\frac{b_R}{b_m})$

= nächst größere ganze Zahl des Bruchs, d.h. mindestens = 1

Die äußere Relation sollte die kleinere Relation sein, also $b_R < b_S$.

Die Kosten einer geschachtelten Schleife werden daher abgeschätzt mit:

$\rightarrow$ Kosten: $b_R + \left\lceil \frac{b_R}{b_m} \right\rceil \cdot b_S$

Bemerkung: Die Kosten können nur dann so berechnet werden, wenn jedes Tupel r nur mit einem Tupel s kombiniert oder weiterverarbeitet wird. Würde man z.B. für jedes Tupel r die ganze Relation S benötigen, dann müsste man die Kosten anders berechnen. Man müsste den Datenbank-Puffer einmal mit Tupeln aus S füllen und dann das übrige S für jedes Tupel aus R von der Platte holen.

$\rightarrow$ Kosten: $b_R + \min(b_m, b_S) + |R| \cdot \max(0, b_S - b_m)$

Ein Beispiel werden wir bei der Gruppierung sehen.

Bei einer dreifachen Schleife

```
for r ∈ R
   for s ∈ S
      for t ∈ T
         show (r‖s‖t)
```

erhalten wir:

$\rightarrow$ Kosten: $b_R + \left\lceil \frac{b_R}{b_m} \right\rceil \cdot (b_S + \left\lceil \frac{b_S}{\max(1, b_m - b_R)} \right\rceil b_T)$

27.3 Sortieren

Sortieren ist eine der wichtigsten Basisoperationen. Hier ist der Aspekt interessant, inwiefern und mit welchen Kosten der Sekundärspeicher einbezogen wird **(externe Sortieralgorithmen)**.

Wir stellen ein Verfahren exemplarisch vor: ***Sort-Merge***. Dabei wird

1. in einem ersten Durchlauf (Operation `partition-sort`) die Originalrelation in gleich große Stücke aufgeteilt, die jeweils komplett im Hauptspeicher sortiert werden können. Für die Sortierung können die bekannten Sortieralgorithmen verwendet werden. D.h. wir unterteilen eine Relation R in
$$p = \left\lceil \frac{b_R}{b_m} \right\rceil$$
Teilrelationen $R_1 ... R_p$. (Jede Partition wird von der Platte gelesen, sortiert und zurückgeschrieben. Das ergibt $2b_R$ Plattenzugriffe.)

2. In einem zweiten Schritt (Operation `merge`) werden nun n Mischläufe durchgeführt. Dabei werden jeweils zwei (oder mehr, zunächst bleiben wir bei zwei) Teilrelationen gemischt, wobei die Blöcke der beteiligten Teilrelationen in den Hauptspeicher gelesen und wieder auf den Sekundärspeicher zurückgeschrieben werden. Jeder Block muss genau einmal gelesen und geschrieben werden. Jeder Mischlauf liest und schreibt alle Blöcke einer Relation genau einmal. Mit n Mischläufen können 2^n Teilrelationen gemischt werden. Es gilt:
$$2^n = p$$
Für eine Relation R sind somit
$$n = \log_2 \left\lceil \frac{b_R}{b_m} \right\rceil$$
Mischläufe notwendig.

Unter Einbeziehung der Partition ergibt das insgesamt
$$2b_R \left(1 + \log_2 \left\lceil \frac{b_R}{b_m} \right\rceil\right)$$
Plattenzugriffe (lesend und schreibend).

Da in einem Mischschritt pro zu mischender Teilrelation nur ein Block in den Hauptspeicher geladen werden muss, können wir mehr als eine Teilrelation auf einmal mischen. In der Tat können wir $b_m - 1 \approx b_m$ Teilrelationen auf einmal mischen. Damit verbessert sich die Aufwandsabschätzung zu:
$$2b_R \left(1 + \log_{b_m} \left\lceil \frac{b_R}{b_m} \right\rceil\right)$$
Das Wesentliche an dieser Formel ist, dass auch das externe Sortieren mit logarithmischem Aufwand $O(n \log n)$ möglich ist.

Wir werden für die Sortierung im folgenden immer den kompakten Peudocode

$T = \omega_A R$

verwenden, wobei T ein sortiertes Zwischenergebnis auf dem Sekundärspeicher ist. Wir verwenden die vereinfachte Kostenformel:

$\rightarrow$ Kosten: $b_R \log b_R$

Wenn wir dieses Zwischenergebnis wieder vom Sekundärspeicher holen, dann schreiben wir

`for` $r \in \omega_A R$

mit der vereinfachten Kosten-Formel:

$\rightarrow$ Kosten: $b_R \log b_R + b_R$

27.4 Unäre Operationen

27.4.1 Selektion

σ_φ `R`

Selektivität, Kardinalität

Da die Ergebnisrelation einer Selektion normalerweise weiterverarbeitet wird, braucht der Optimierer eine Abschätzung über deren Kardinalität. Die **Selektivität** ***(selectivity)*** einer Selektion ist definiert als

$$\texttt{sel} = \frac{\text{Erwartete Kardinalität des Ergebnisses}}{\text{Kardinalität der Eingangsrelation}}$$

Ein kleiner Wert für die Selektivität bedeutet eine "hohe" Selektivität!

Bezeichnung:

$\texttt{sel}(\varphi, R) = \frac{|\sigma_\varphi R|}{|R|}$

Für die Selektivität werden folgende Abschätzungen vorgeschlagen:

$\texttt{sel}(A = a, R) = \frac{1}{v_{A,R}}$

Wenn die Attributwerte nicht gleichverteilt sind, dann kann für `sel` der Wert $0,1$ genommen werden.

Ferner haben sich folgende Abschätzungen bewährt:

$\texttt{sel}(A < a, R) = \frac{a - A_{\min}}{A_{\max} - A_{\min}}$ oder $= 0,3$ für den Fall, dass die Formel keinen Sinn macht.

$\texttt{sel}(A > a, R) = \frac{A_{\max} - a}{A_{\max} - A_{\min}}$ oder $= 0,3$ für den Fall, dass die Formel keinen Sinn macht.

$\texttt{sel}(a_1 < A < a_2, R) = \frac{a_2 - a_1}{A_{\max} - A_{\min}}$ oder $= 0,25$ für den Fall, dass die Formel keinen Sinn macht.

Bemerkung: Für die Abschätzung bei Zeichenketten kann auch eine Zeichenstatistik verwendet werden, wenn sie vorhanden ist. Im einfachsten Fall z.B. könnte man

$$\texttt{sel}('M' \leq A \leq' Q', R) = \tfrac{5}{26}$$

setzen, da der Bereich 5 von 26 Buchstaben betrifft.

Logische Verknüpfungen:

$$\texttt{sel}(\varphi_1 \wedge \varphi_2, R) = \texttt{sel}(\varphi_1, R) \cdot \texttt{sel}(\varphi_2, R)$$

$$\texttt{sel}(\varphi_1 \vee \varphi_2, R) = \texttt{sel}(\varphi_1, R) + \texttt{sel}(\varphi_2, R) - \texttt{sel}(\varphi_1, R) \cdot \texttt{sel}(\varphi_2, R)$$

$$\texttt{sel}(\neg\varphi, R) = 1 - \texttt{sel}(\varphi, R)$$

Die obere Grenze der Selektivität ist natürlich 1.

$\rightarrow$ Kardinalität: $|\sigma_\varphi R| = \texttt{sel}(\varphi, R)|R|$

Selektion mittels Relationen-Scan

Mittels eines Relationen-Scans wird für alle Tupel $r \in R$ der Ausdruck $\varphi(r)$ ausgewertet. Ergibt sich der Wert `true`, wird r in die Ergebnisrelation übernommen.

```
for r ∈ R
   if φ(r)
      show(r)
```

Der Aufwand ist somit $O(|R|)$. Wir verwenden für die **Selektion durch Relationen-Scan** folgende Notation:

$$\texttt{for } r \in \sigma_\varphi^{\texttt{REL}} R$$

$\rightarrow$ Kosten: b_R

Selektion mittels Index-Scan

Wenn das Prädikat φ **atomar** ist, d.h. mit den Daten in **einem** Index ausgewertet werden kann (z.B. exakte Suche oder Bereichsanfrage), dann ist ein Index-Scan möglich, der eine TID-Liste liefert und folgendermaßen notiert wird:

```
for a ∈ I_{A,R}
   if φ(a)
      for r ∈ ρI_{A,R}(a)
         show(r)
```

Das können wir kürzer notieren als Index-Scan:

$$\texttt{for } r \in \rho\sigma_{\varphi}^{\texttt{IND}} I_{A,R}$$

→ Kosten: $b_{I_{A,R}}$ + Anzahl der zu realisierenden Tupel = $b_{I_{A,R}} + \texttt{sel}(\varphi, R)|R|$

Wenn φ nicht mit den Daten *eines* Indexes ausgewertet werden kann, sondern nur teilweise oder nur mit mehreren Indexen, dann sprechen wir von einem **zusammengesetzten** Selektionsprädikat φ, das eventuell aus atomaren Prädikaten mittels der aussagenlogischen Operatoren `and`, `or`, `not` und Klammerungen gebildet wird.

Wir nennen hier zwei Möglichkeiten:

1. Bildung einer **konjunktiven Normalform (KNF)**:

   $$\varphi = \varphi' \wedge \varphi''$$

   Dabei besteht φ' nur aus atomaren Konjunkten:

   $$\varphi' = \varphi_1 \wedge \ldots \wedge \varphi_n$$

   (a) Auswertung der atomaren Konjunkte mit Hilfe der jeweiligen Indexe

   (b) Schneiden der resultierenden TID-Listen ergibt eine TID-Liste, die realisiert wird.

   (c) Mit den Tupeln der realisierten resultierenden TID-Liste wird φ'' ausgewertet.

2. Wenn die Bildung der konjunktiven Normalform schwierig ist oder nicht zu einer wesentlichen Vereinfachung führt, dann bietet sich als weitere Möglichkeit zur Auswertung komplexer Prädikate die **zweistufige Filtermethode** an.

   (a) Hierbei werden zunächst alle Bedingungen, die **nicht** durch einen Index unterstützt werden, auf `true` gesetzt.

   Das resultierende Prädikat φ' kann eventuell in KNF transformiert werden. Die einzelnen Terme werden nun unter Ausnutzung von Indexen ausgewertet und liefern schließlich eine TID-Liste,

   (b) auf deren Realisierung das ursprüngliche Prädikat φ ausgewertet wird.

In beiden Fällen ergeben sich die Kosten zu:

→ Kosten:

$$\sum_{i=1}^{n} b_{I_{A_i,R}} + \texttt{sel}(\varphi', R)|R|$$

Dabei wird über alle Attribute A_i summiert, für die ein Index eingesetzt werden konnte.

27.4.2 Projektion

$\pi_A R$ mit $A = \{A_1, A_2 \ldots A_n\}$

Bemerkungen:

1. Die relationale Projektion nach Codd erzwingt die Elimination von Duplikaten. Das ist oft nicht notwendig und daher aus Gründen der Effizienz in SQL defaultmäßig nicht vorgesehen.
2. Da eine Projektion *ohne* Duplikatelimination einer modifizierten Ausgabe bei einem Scan entspricht, behandeln wir im folgenden nur Projektionen *mit* Duplikatelimination. Nur diese tragen zu den Kosten einer Anfrage bei.
3. Wenn die Projektionsattribute einen Schlüssel enthalten, dann können keine Duplikate entstehen.
4. Auch ohne Duplikatelimination kann die Projektion die Kosten für nachgelagerte Operationen reduzieren, da durch die Streichung von Attributen mit einem größeren Blockungsfaktor gerechnet werden kann.

Die Kardinalität der Ergebnisrelation kann bei Duplikatelimination durch folgende Formel abgeschätzt werden:

→ Kardinalität:

$$|\pi_A R| \approx v_{A,R}$$

Zur Auswertung von $\pi_A R$ müssen wir die Relation R zunächst nach A sortieren. Den Algorithmus notieren wir in folgendem Pseudocode:

```
prev = null
for r ∈ ω_A R
   if r.A != prev
      show(r.A)
      prev = r.A
```

Der Aufwand für die Projektion ergibt sich aufgrund der Sortierung zu $O(|R| \log |R|)$.

→ Kosten: $b_R \log b_R + b_R$

(b_R wird für das Lesen des sortierten R angesetzt.)

Falls `R` schon nach A sortiert vorliegt, beträgt der Aufwand nur $O(|R|)$ und die Kosten b_R.

Für die Sortierung können natürlich auch die günstigeren Formeln aus dem Abschnitt "Sortieren" eingesetzt werden.

Neben der Sortierung gibt es auch Hashverfahren zur Duplikaterkennung.

Wenn wir Indexe ausnutzen, dann können wir auf die Sortierung verzichten:

- Eine Projektion auf indexierte Attribute ist ohne Zugriff auf gespeicherte Tupel möglich, da diese direkt aus dem Index ausgelesen werden können.

  ```
  for a ∈ I_{A,R}
     show(a)
  ```

  → Kosten: $b_{I_{A,R}}$

- Das sortierte Auslesen eines Indexes auf einem Attribut A_1 aus der Menge der Projektions-Attribute A hilft nur manchmal bei der Duplikatelimination.

  → Kosten: $b_{I_{A_1,R}} + |R| +$ (oder mehr)

  A-Duplikate sind zwar benachbart, aber eventuell müssen wir für jeden A_1-Wert noch aufwändig sortieren.

27.4.3 Aggregatfunktionen und Gruppierung

Ohne Beschränkung der Allgemeinheit betrachten wir den Fall

$$\gamma_{f(B);A}R$$

A ist das Gruppierungsattribut (oder -attributmenge) und f die Aggregatfunktion über dem Attribut B.

Für die Kardinalität der Ergebnisrelation einer Gruppierung erhält man Werte zwischen 1 (nur Aggregation ohne Gruppierung) und $|R|$ (jedes Tupel bildet eine Gruppe). Als Abschätzung bietet sich die gleiche Formel wie bei der Projektion mit Duplikateliminierung an:

→ Kardinalität:

$$|\gamma_{f;A_1,A_2\ldots A_n}R| = |\pi_A R| \approx v_{A,R}$$

mit $A = \{A_1, A_2 \ldots A_n\}$

Ein naheliegender Ansatz basiert auf einem Scan der Relation R und dem sukzessiven Aggregieren in einer temporären Ergebnisrelation T. Da die Relation T für jedes Eingabetupel immer wieder durchlaufen werden muss, benötigen wir eine geschachtelte Schleife. Diese Technik wird daher als ***nested loops*** bezeichnet.

$$\gamma^{\texttt{NL}}_{f;A}R$$

Der folgende Pseudocode zeigt das Beispiel

$$\gamma^{\texttt{NL}}_{\texttt{sum}(B);A}R$$

```
T = createRel(A, B)
for r ∈ R                    // kostet b_R
   found = false
   for t ∈ T   // kostet |R| (max(0,b_T − b_m))/2 · (max(0,b_T − b_m))/b_T
```

```
        if r.A == t.A
            t.B = t.B + r.B  // sum
            found = true
            break // for t
    if !found
        T.insert(r.A, r.B)    // kostet max(0, b_T − b_m)
for t ∈ T                       // kostet max(0, b_T − b_m)
    show(t)
```

(Der zweite Kostenterm entsteht, weil (1.) für jedes $r \in R$ T im Schnitt zur Hälfte durchsucht werden muss und T i.a. nur teilweise in den DB-Puffer passt, wobei $\frac{\max(0,b_T-b_m)}{b_T}$ die Wahrscheinlichkeit ist, dass das gesuchte t auf der Platte ist.)

Im ungünstigsten Fall bildet jedes Tupel von R eine Gruppe und der Aufwand ist $O(|R|^2)$. Im günstigsten Fall gibt es nur eine Gruppe und der Aufwand ist linear.

→ Kosten: $b_R + |R|\frac{\max(0,b_T-b_m)}{2} \cdot \frac{\max(0,b_T-b_m)}{b_T} + 2\max(0, b_T - b_m)$

Wenn wir vorher nach dem Gruppierungsattribut sortieren, dann kann die Aggregation in einem Durchlauf erfolgen:

$\gamma^{\texttt{SORT}}_{f;A} R$

```
first r ∈ ω_A R
t.A = r.A    // Gruppentupel
t.B = 0      // Aggregat
for r ∈ ω_A R
    if r.A!=t.A
        show(t)    // Gruppe fertig
        t.A = r.A
        t.B = 0
    t.B = t.B + r.B    // sum
show(t)    // letzte Gruppe fertig
```

Wenn eine Sortierung notwendig ist, beträgt der Aufwand $O(|R| \log |R|)$.

→ Kosten: $b_R \log b_R + b_R$

Als weitere Variante kann man eine Hashfunktion zur schnellen Identifizierung der Gruppen verwenden. Für jedes Eingabetupel t liefert der Hashwert von $t.A$ die Adresse der aggregierten Gruppe.

$\gamma^{\texttt{HASH}}_{f;A} R$

```
for r ∈ R
    t = H.get(r.A)       // H Hash-Tabelle, enthält Resultat
    if t == null   // Gruppe gibt es noch nicht.
```

```
        t = r.A‖r.B
        H.put(r.A, t)
    else
        t.B = t.B + r.B     // sum
for t ∈ H
    show(t)
```

→ Kosten: b_R, wenn alle Ergebnisgruppen, d.h. die Hashtabelle in den Hauptspeicher passen, sonst im ungünstigsten Fall $b_R + 2|R| + b_T$

Voraussetzung für die NL- und HASH- Verfahren ist die **Additivität** der Aggregatfunktionen (distributive Funktion). Man unterscheidet nämlich

- **distributive** (`count`, `sum`, `max`, `min`),
- **algebraische** (`avg`, können algebraisch aus distributiven Funktionen berechnet werden, $\text{avg} = \frac{\text{sum}}{\text{count}}$) und
- **holistische** (`median`, zur Berechnung wird immer das ganze Ensemble benötigt)

Aggregatfunktionen. Bei holistischen Aggregatfunktionen geht nur $\gamma^{\texttt{SORT}}_{f;A} R$.

27.5 Binäre Operation Join

Die wichtigste binäre Operation ist der Join (Verbund) von zwei Relationen `R` und `S`:

$$R \bowtie_{\varphi(r\|s)} S$$

Hierbei bedeutet $r\|s$ das Verschmelzen von zwei Tupeln $r \in R$ und $s \in S$ zu einem Tupel.

27.5.1 Kardinalität des Joins

Die Kardinalität der Ergebnisrelation eines Verbundes hängt sehr stark von der Art des Prädikats φ ab:

→ Kardinalität: $sel(\varphi) \cdot |R| \cdot |S| = 0 \ldots |R| \cdot |S|$

liegen.

Für den Fall eines allgemeinen Equijoins (Gleichverbund), nämlich dass aus beiden Relationen die Attribute $A = \{A_1, A_2 \ldots A_n\}$ auf Gleichheit überprüft werden, kann folgende Abschätzung verwendet werden:

→ Kardinalität:

$$|R \bowtie_A S| \approx |R| \cdot |S| \cdot \prod_{i=1}^{n} \frac{1}{\max(v_{A_i,R}, v_{A_i,S})}$$

Die Faktoren $\frac{1}{\max(v_{A_i,R}, v_{A_i,S})}$ werden als **Verbundselektivitäten *(join selectivity)*** bezeichnet.

Wenn eine Fremdschlüsselbeziehung $R \overset{K}{\leftarrow} S$ vorliegt (K ist Schlüssel in R und Fremdschlüssel in S.), dann ist die Verbundselektivität $1/|R|$ und die obige Formel liefert wie zu erwarten die

$\rightarrow$ Kardinalität: $|R \bowtie_K S| = |S|$

Dies ist bei weitem der häufigste Fall einer Verbundoperation. Damit ist die Join-Problematik – Generierung sehr vieler Tupel – im Wesentlichen entschärft.

Falls auf die Schlüsseltabelle R eine Selektion angewendet wird, dann gilt:

$\rightarrow$ Kardinalität: $|(\sigma_\varphi R) \bowtie_K S| \approx \texttt{sel}(\varphi, R) \cdot |S|$

Bemerkungen:

1. Den Aufwand für das Schreiben der Ergebnisrelation haben wir bisher nicht besonders diskutiert. Bei einem Verbund, der im Extremfall das Kreuzprodukt sein kann, fällt das mit $O(|R| \cdot |S|)$ eventuell ins Gewicht. Bei einem "normalen" Join über eine Schlüssel-Fremdschlüsselbeziehung `R` $\leftarrow$ `S` haben wir allerdings nur linearen Schreib-Aufwand $O(|S|)$. Jedenfalls werden wir bei den folgenden algorithmischen Varianten die Kosten für das Schreiben der Ergebnisrelation **nicht** berücksichtigen.

2. Falls man das Ergebnis auf die Platte schreibt, muss man eine Abschätzung für den Blockungs-Faktor des Joins haben (Vgl. Übung). Dabei stellt sich unter Umständen die Frage, ob Spalten automatisch gestrichen werden oder nicht.

   Beim `R join S using (A)` kann man davon ausgehen, dass eine `A`-Spalte gestrichen wird.

   Beim `R join S on (R.A=S.B)` kann man davon ausgehen, dass keine Spalte gestrichen wird, da das Prädikat auch komplizierter sein könnte.

27.5.2 Nested-Loops-Verbund

Die einfachste Variante der Verbundberechnung beruht auf dem Prinzip der geschachtelten Schleifen.

$R \bowtie_\varphi^{\texttt{NL}} S$

```
for r ∈ R
   for s ∈ S
      if φ(r‖s)
         show(r||s)
```

Für die Aufwandsabschätzung ergibt sich ohne Indexausnutzung $O(|R||S|)$.

$\rightarrow$ Kosten: $b_R + \left\lceil \frac{b_R}{b_m} \right\rceil b_S$

Dabei ist berücksichtigt, dass jeder Block von R genau einmal gelesen werden muss und das jeweils b_m Blöcke von R im Speicher gehalten werden können (oBdA $b_R < b_S$).

Der Vorteil des Nested-Loops-Algorithmus ist, dass das Prädikat φ beliebig sein kann. Bei allen anderen Techniken muss das Prädikat ein Gleichheits-(u.U. auch Ungleichheits-)Vergleich sein.

Spezialfälle sind Nested-Loops-Verbund mit Indexausnutzung.

Wenn es einen Index $I_{B,S}$ gibt und φ nur – evtl. kompliziert – von $r \in R$ und $s.B (s \in S)$ abhängt, dann können wir den Index ausnutzen:

$$R \bowtie^{\texttt{NL/IND}}_{\varphi(r\|s.B)} S$$

Fall $b_{I_{B,S}} \leq b_R$:

```
for b ∈ I_{B,S}
   for r ∈ R
      if φ(r‖b)
         for s ∈ ρI_{B,S}(b)
            show (r‖s)
```

→ Kosten: $b_{I_{B,S}} + \left\lceil \frac{b_{I_{B,S}}}{b_m} \right\rceil b_R + \texttt{sel}(\varphi) v_{B,S} |R| (\frac{|S|}{v_{B,S}})$
$= b_{I_{B,S}} + \left\lceil \frac{b_{I_{B,S}}}{b_m} \right\rceil b_R + \texttt{sel}(\varphi)|R||S|$
(Jedes Tupel $s \in S$, das ins Resultat geht, muss von der Platte geholt werden.)

Fall $b_R \leq b_{I_{B,S}}$:

```
for r ∈ R
   for b ∈ I_{B,S}
      if φ(r‖b)
         for s ∈ ρI_{B,S}(b)
            show (r‖s)
```

→ Kosten: $b_R + \left\lceil \frac{b_R}{b_m} \right\rceil b_{I_{B,S}} + \texttt{sel}(\varphi)|R||S|$

Wenn es beide Indexe $I_{A,R}$ und $I_{B,S}$ gibt und wenn φ nur von den Indexfeldern $r.A$ und $s.B$ abhängt, dann gilt (oBdA $b_{I_{A,R}} \leq b_{I_{B,S}}$):

```
for a ∈ I_{A,R}
   for b ∈ I_{B,S}
      if φ(a‖b)
         for r ∈ ρI_{A,R}(a)
            for s ∈ ρI_{B,S}(b)
               show (r‖s)
```

$\rightarrow$ Kosten: $b_{I_{A,R}} + \left\lceil \frac{b_{I_{A,R}}}{b_m} \right\rceil b_{I_{B,S}} + \texttt{sel}(\varphi) v_{A,R} v_{B,S} (\frac{|R|}{v_{A,R}} + \frac{|R||S|}{v_{A,R} v_{B,S}})$

$= b_{I_{A,R}} + \left\lceil \frac{b_{I_{A,R}}}{b_m} \right\rceil b_{I_{B,S}} + \texttt{sel}(\varphi)(v_{B,S}|R| + |R||S|)$

$= b_{I_{A,R}} + \left\lceil \frac{b_{I_{A,R}}}{b_m} \right\rceil b_{I_{B,S}} + \frac{1}{\max(v_{A,R}, v_{B,S})}(v_{B,S}|R| + |R||S|)$

(Wenn man kleinlich ist, könnte man für die dritte Schleife eventuell übrigen DB-Puffer ausnützen.)

Für den Spezialfall, dass ein Equijoin über dem Attribut A gebildet wird und ein Index $I_{A,S}$ existiert, können wir die zugehörigen Tupel in S direkt aus dem Index bestimmen:

```
for r ∈ R
   for s ∈ ρI_{A,S}(r.A)
      show (r‖s)
```

- Bei einem Sekundärindex gibt es im Durchschnitt $\frac{|S|}{v_{A,S}}$ Tupel mit dem gleichen A-Wert. Daher ergibt sich im Prinzip wieder ein quadratischer Aufwand:

  $\rightarrow$ Kosten: $b_R + |R|(l_{I_{A,S}} + \frac{|S|}{v_{A,S}})$

- Wenn das Verbundattribut A allerdings ein Schlüssel von S ist und dort ein Index definiert ist, dann ergibt sich wegen $v_{A,S} = |S|$ ein linearer Aufwand:

  $\rightarrow$ Kosten: $b_R + |R|(l_{I_{A,S}} + 1)$

- Wenn das Verbundattribut A ein Schlüssel von R ist, dann ergibt sich wegen $v_{A,S} = |R|$ ein linearer Aufwand:

  $\rightarrow$ Kosten: $b_R + |R| l_{I_{A,S}} + |S|$

27.5.3 Merge-Techniken

Bei Tupelvergleichen bezüglich Gleichheit (Equijoin) kann man mischen. Wir betrachten einen Equijoin über dem Attribut A.

$R \bowtie_A^{\texttt{MERGE}} S$

1. Sortiere R und S bezüglich A.
2. Falls $r \in R$ kleiner als $s \in S$ bezüglich A ist ($r.A < s.A$), wird das nächste $r \in R$ gelesen.
3. Falls $r.A > s.A$, wird das nächste $s \in S$ gelesen.
4. Falls $r.A == s.A$, wird r mit s und allen Nachfolgern s' von s, die auf A mit s übereinstimmen, verbunden. Beim Erreichen des ersten s', das nicht mit s auf A übereinstimmt, wird das ursprüngliche s mit allen Nachfolgern r' von r, die auf A mit r übereinstimmen, verbunden.

Diesen Algorithmus kürzen wir mit

```
for r‖s ∈ merge(ω_A R, ω_A S)
   if r.A == s.A        // Join-Bedingung
      show(r‖s)
```

ab.

Der Aufwand ergibt sich aus den Zugriffskosten für die beiden Relationen. Wenn die Relationen sortiert vorliegen, ergibt das nur:

→ Kosten: $b_R + b_S$

und wenn sortiert werden muss, erhält man:

→ Kosten: $b_R + b_R \log b_R + b_S + b_S \log b_S$

Bemerkung: Das gilt unter der Voraussetzung, dass die "Runs" gleicher A-Werte in den Hauptspeicher passen.

Wir betrachten den – allerdings häufigsten – Spezialfall, dass eine Fremdschlüsselbeziehung $R \xleftarrow{K} S$ vorliegt und dass es in beiden Relationen einen Index über dem Verbundattribut K gibt. Also:

$$R \bowtie_K^{\texttt{MERGE}} S$$

K ist Schlüssel in R und Fremdschlüssel in S. Der Pseudocode für einen Algorithmus, der die beiden Indexe von vorn bis hinten durchläuft und die jeweiligen Tupel holt, lautet:

```
for k ∈ merge(I_{K,R}, I_{K,S})
   if k ∈ I_{K,R} ∧ k ∈ I_{K,S}
      r = ρI_{K,R}(k)
      for s ∈ ρI_{K,S}(k)
         show(r‖s)
```

Die Kosten berechnen sich zu:

→ Kosten: $b_{I_{K,R}} + b_{I_{K,S}} + v_{K,S} + |S|$

Falls es nur den Index $I_{K,R}$ gibt, dann sortieren wir S nach K und erhalten:

```
first s ∈ ω_K S
for k ∈ I_{K,R}
   if k == s.K
      r = ρI_{K,R}(k)
      while k == s.K
         show(r‖s)
         next s ∈ ω_K S
```

→ Kosten: $b_{I_{K,R}} + v_{K,S} + b_S + b_S \log b_S$

27.5.4 Hash-Techniken

Diese Methoden können ebenfalls nur bei einem Equijoin eingesetzt werden.

$R \bowtie_A^{\texttt{HASH}} S$

Der **Hash-partionierte Verbund** wird in zwei Schritten durchgeführt:

1. (Partition) A sei das Join-Attribut. Die Tupel aus R und S werden unter Verwendung einer Hashfunktion $h_1(A)$ (hier auch als Splitfunktion bezeichnet) in p Partitionen R_i und S_i eingeordnet und auf den Externspeicher geschrieben.
   (z.B. $h_1(A) = h_c(A) \bmod p$)
   Die Partitionen der kleineren Relation – oBdA sei das R – sind so gewählt, dass sie in den Hauptspeicher passen.

   ```
   for r ∈ R
       i = h1(r.A)
       Ri.insert(r)
   for s ∈ S
       i = h1(s.A)
       Si.insert(s)
   ```

2. (Verbund) Für jede der Partitionen $i = 1..p$ werden nun folgende Schritte durchgeführt:

   (a) Die Tupel aus R_i werden gelesen und mittels einer zweiten Hashfunktion $h_2(A)$ in eine Hashtabelle H im Hauptspeicher eingetragen. ($h_2(A)$ bildet A ein-eindeutig (injektiv) auf die Eimernummern ab.)

   (b) Die Tupel aus S_i werden gelesen. Unter Anwendung von h_2 können die passenden Verbundpartner in der Hashtabelle H gefunden und der eigentliche Verbund ausgeführt werden. (Dabei kann natürlich durchaus mehr als ein $r \in R$ gefunden werden.) (Das Ergebnis kann blockweise zurückgeschrieben werden.)

   ```
   for i = 1..p
      clear H
      for r ∈ Ri
         H.put(r.A, r)        // (mit h2)
      for s ∈ Si
         for r ∈ H.get(s.A)
            show(r‖s)
   ```

Da die Tupel der zweiten Relation ihre Vergleichspartner mittels einer Hashfunktion finden, kann hier nur das Prädikat "`==`" unterstützt werden. Es gilt hier der Aufwand $O(|R| + |S|)$.

→ Kosten: $b_R + |R| + b_S + |S| + b_R + b_S = 2(b_R + b_S) + |R| + |S|$
Hierbei wurde für `insert` nur **ein** Sekundärspeicherzugriff angesetzt.

27.5.5 Binäre Mengenoperationen

Vereinigung

Ohne Duplikateliminierung:

→ Kardinalität: $|R \cup S| = |R| + |S|$

→ Kosten: $b_R + b_S$

Mit Duplikateliminierung:

→ Kardinalität:

$$|R \cup S| = \max(|R|, |S|) \quad \ldots \quad \frac{\max(|R|, |S|) + |R| + |S|}{2} \quad \ldots \quad |R| + |S|$$

→ Kosten: $3b_R + 3b_S$ (mit Hash-Techniken)

Differenz

→ Kardinalität:

$$|R - S| = 0 \quad \ldots \quad |R| - 1/2|S| \quad \ldots \quad |R|$$

→ Kosten: $3b_R + 3b_S$ (mit Hash-Techniken)

Schnitt

→ Kardinalität:

$$|R \cap S| = 0 \quad \ldots \quad \frac{|R| \cdot |S|}{\max(v_{A,R}, v_{A,S}) \cdot \max(v_{B,R}, v_{B,S}) \cdot \quad \ldots} \quad \ldots \quad \min(|R|, |S|)$$

Bei dem mittleren Term verwenden wir die Verbundselektivität für alle Attribute. (Der Schnitt ist ja ein Verbund bezüglich aller Attribute.)

→ Kosten: $3b_R + 3b_S$ (mit Hash-Techniken)

27.6 Zusammenfassung

In folgender Tabelle listen wir die Kosten für einzelne Anweisungen eines Pseudocodes auf. Damit kann man dann die Kosten für einen Algorithmus abschätzen.

Anweisung:	Kosten:	Kardinalität des Resultats:
`for` $r \in R$	b_R	$\|R\|$
`for` $r \in \rho I_{A,R}(a)$	$l_{I_{A,R}} + \frac{\|R\|}{v_{A,R}}$	$\frac{\|R\|}{v_{A,R}}$
`for` $r \in \rho I_{A,R}(a_1, a_2)$	$l_{I_{A,R}}$ $+ \texttt{sel}(a_1 \leq A \leq a_2, R)$ $\cdot\|R\|(1 + \frac{b_{I_{A,R}}}{\|R\|})$	$\texttt{sel}(a_1 \leq A \leq a_2, R)\|R\|$
`for` $r \in \omega_A R$	$b_R \log b_R + b_R$	$\|R\|$
`for` $r \in \sigma_\varphi R$	b_R	$\texttt{sel}(\varphi, R)\|R\|$
`for` $r.A \in \pi_A R$	$b_R(1 + \log b_R)$	$v_{A,R}$
`for` $a \in I_{A,R}$	$b_{I_{A,R}}$	
$T = \sigma_\varphi R$	$b_R + \texttt{sel}(\varphi, R) b_R$	$\texttt{sel}(\varphi, R)\|R\|$
`if` ...	0	
`show`(...)	0	
$T =$ `operation`R	$b_R + \frac{\|T\| f_R}{\|R\| f_T} b_R$	$\|T\|$

Schleifen:
`for` $r \in R$
  `for` $s \in S$
  ...
	b_R
	$+ \left\lceil \frac{b_R}{b_m} \right\rceil b_S$
		$\|R\|\|S\|$

Genereller Hinweis zur Kostenberechnung für Nested-Loop-Algorithmen:

Kosten
= Kosten-für-geschachtelte-Schleifen
plus Produkt-aus
  Kardinalität-äußere-Schleife
mal
  Kardinalität-innere-Schleife
mal
  Selektivität(φ)
mal
  Anzahl-Tupel-hinter-der-Selektion

Die Selektivität kann dann oft durch die Verbund-Selektivität abgeschätzt werden.

Beispiel:

$$(\pi_A R) \bowtie_\varphi (\sigma_{B==b} S)$$

$R_1 = \omega_A R$	$b_R \log b_R$	$\|R\|$
$S_1 = \sigma_{B==b} S$	$b_S + \texttt{sel}(B == b, S) b_S$	$\texttt{sel}(B == b, S)\|S\|$
`for` $r \in \pi_A R_1$	b_R	$\texttt{sel}(\varphi, R \bowtie S)\|R_1\|\|S_1\|$
`for` $s \in S_1$	$+ \left\lceil \frac{b_R}{b_m} \right\rceil \texttt{sel}(B == b, S) b_S$	
`if` $\varphi(r\|s)$	0	
$\texttt{show}(r\|s)$	0	
Gesamt:	...	...

27.7 Übungen

27.7.1 Blockungsfaktor

Gegeben sind die Blockungsfaktoren f_R und f_S. Geben Sie eine Formel für den Blockungsfaktor des Kreuzprodukts bzw. Joins $f_{R \bowtie S}$ an.

Kapitel 28

Optimierung

Erst seit es relationale Datenbanksysteme gibt, ist eine Automatisierung der Optimierung von Datenbankanfragen möglich. Dies ist bei relationalen Datenbanken auch notwendig, da sie sonst zu langsam wären. Ferner ist der Benutzer mit den low-level Entscheidungen überfordert, die bei relationalen Systemen notwendig sind. Fehler bei diesen Entscheidungen können die Effizienz um einige Größenordnungen beeinflussen.

Optimierung in relationalen Systemen bedeutet, dass es um die Optimierung relationaler Ausdrücke geht.

Zunächst stellen wir zwei Beispiele vor.

28.1 Motivierende Beispiele

Das erste Beispiel verzichtet auf Details der Speicherung. Beim zweiten, ausführlicheren Beispiel wird die Dateiorganisation mit berücksichtigt.

28.1.1 Beispiel 1

Anfrage: *Namen der Teile, die Lieferant* `S2` *liefert:*

```
select distinct PNAME from SP join P using(PNR) where SNR = 'S2';
```

π[PNAME] (σ[SNR = 'S2'] (SP $\bowtie$[PNR] P))

Wir nehmen an, dass wir 5000 Teile und 100 000 Lieferungen haben, von denen 100 Lieferungen den Lieferanten `S2` betreffen. Die Größe des Datenbankpuffers im Hauptspeicher ist winzig, es passen gerade etwa 100 Tupel (egal welcher Größe) rein.

Die naive Auswertung des relationalen Ausdrucks würde folgende Schritte haben:

1. `join` ($\bowtie$): Liest 100 000 Lieferungen. Liest für jede Lieferung 5000 Teile. Schreibt 100 000 gejointe Tupel wieder auf Platte.

2. Selektion (σ) von Lieferant `S2`: Liest 100 000 gejointe Tupel und behält davon 100, die im Hauptspeicher bleiben.
3. Projektion (π): Endresult wird eventuell durch Streichung von Duplikaten erreicht.

Eine optimierte Auswertung würde folgende Schritte haben:

1. Selektion der Lieferungen von Lieferant `S2`: Liest 100 000 Tupel, von denen 100 im Hauptspeicher behalten werden.
2. Resultat von Schritt 1 wird mit Teilen gejoint: Liest 5000 Teile einmal, weil die 100 Lieferungen im Hauptspeicher sind. Resultat enthält 100 Tupel im Hauptspeicher.
3. Projektion.

Die erste Auswertung hat

$$3 \cdot 100\,000 + 100\,000 \cdot 5000 = 500\,300\,000$$

Tupel-I/O-Operationen, die zweite Auswertung hat 105 000 Tupel-I/O-Operationen. Obwohl man eigentlich Seiten-I/O's betrachten müsste, ist klar, dass die zweite Auswertung vorzuziehen ist, die 5000 mal schneller ist. Bei Verwendung von Indexen kann das Verhältnis noch extremer werden.

28.1.2 Beispiel 2

Wir betrachten ein einfaches, aus zwei Relationen bestehendes Schema:

Abteilung: `ABT [ ANR, ANAME, BUDGET ]`
Mitarbeiter: `MIT [ MNR, MNAME, FACH, ANR ]`

Als Anfrage betrachten wir einen einfachen Verbund über die Fremdschlüsselbeziehung:

```
select ANAME, BUDGET
   from ABT join MIT using(ANR)
   where FACH = 'Java';
```

Wir suchen also die Abteilungen, wo es mindestens einen Mitarbeiter gibt, der als Fach "Java" hat.

Wir nehmen an:

1. In der Relation `ABT` sind 200 Tupel gespeichert. Auf eine Seite passen 10 Tupel.
2. In der Relation `MIT` sind 40 000 Tupel gespeichert. Auf eine Seite passen 8 Tupel.
3. Es gibt 60 Mitarbeiter mit dem Fach "Java".

4. Das Ergebnis der Anfrage passt auf eine Seite.

5. 4 Zeilen des Kreuzprodukts `ABT times MIT` passen auf eine Seite.

6. Der Datenbankpuffer hat für jede Relation einen Speicherbereich, der eine Seite aufnehmen kann.

7. Die Tupel werden nicht überspannend gespeichert.

Naive Auswertung

Alle Tupel der ersten Relation werden mit allen Tupeln der zweiten Relation kombiniert. Es wird auf Verbundzugehörigkeit und die Selektionsbedingung getestet. Folgende Schritte werden durchgeführt:

1. Das Berechnen des Kreuzprodukts und anschließende Schreiben des Zwischenergebnisses `Z1` erfordert

   $L_1 = (200/10) + (200/10) \cdot (40\,000/8) = 100\,020$ Lesezugriffe und
   $S_1 = (200 \cdot 40\,000)/4 = 2\,000\,000$ Schreibzugriffe.

   (Die Tupel beider Relationen werden seitenweise gelesen. Wegen der Puffergröße muss für jede Seite der Abteilungs-Relation jede Seite der Mitarbeiter-Relation erneut gelesen werden.)

2. Beim Join und bei der Selektion wird erst jede Seite des Zwischenergebnisses `Z1` gelesen. Geschrieben werden nur 60 Tupel als Zwischenergebnis `Z2`. Das erfordert

   $L_2 = 2\,000\,000$ Lesezugriffe und
   $S_2 = 60/4 = 15$ Schreibzugriffe.

3. Bei der abschließenden Projektion muss das Zwischenergebnis `Z2` gelesen werden. Es wird eine Seite als Resultat zurückgeschrieben.

   $L_3 = 15$ Lesezugriffe und
   $S_3 = 1$ Schreibzugriff.

Insgesamt werden in dieser Variante

$$L_1 + L_2 + L_3 + S_1 + S_2 + S_3 = 4\,100\,051$$

Plattenzugriffe benötigt. Außerdem werden 2 000 000 Seiten zusätzlicher, wenn auch temporärer Plattenplatz benötigt.

Optimierte Auswertung

Bei der naiven Auswertung scheint die Berechnung des vollen Kreuzprodukts nicht effizient zu sein. Eine Vor-Selektion dürfte die Effizienz wesentlich steigern.

1. Zuerst wird die Selektion der Relation `MIT` durchgeführt. Das Lesen von `MIT` und anschließende Schreiben des Zwischenergebnisses `Z1` erfordert

   $L_1 = 40\,000/8 = 5000$ Lesezugriffe und
   $S_1 = 60/8 = 8$ Schreibzugriffe.

2. Bei der Berechnung des Joins wird für jede Seite von `ABT` jede Seite von `Z1` gelesen. Geschrieben werden maximal 60 Tupel als Zwischenergebnis `Z2`. Das erfordert

   $L_2 = (200/10) \cdot 8 = 160$ Lesezugriffe und
   $S_2 = 60/4 = 15$ Schreibzugriffe.

3. Bei der abschließenden Projektion muss das Zwischenergebnis `Z2` gelesen werden. Es wird eine Seite als Resultat zurückgeschrieben.

   $L_3 = 15$ Lesezugriffe und
   $S_3 = 1$ Schreibzugriff.

Insgesamt werden in dieser Variante

$$L_1 + L_2 + L_3 + S_1 + S_2 + S_3 = 5199$$

Plattenzugriffe benötigt. Wir haben eine Verbesserung um den Faktor 789 erreicht. Der zusätzlich benötigte Plattenspeicherplatz ist unwesentlich.

Auswertung unter Indexausnutzung

Bisher haben wir keine Zugriffsstrukturen verwendet. Wir nehmen jetzt an, dass wir für die Relation `ABT` einen Sekundär-Index auf dem Schlüssel `ANR` und für die Relation `MIT` einen Sekundär-Index auf dem Attribut `FACH` haben. Über die Indexe machen wir die Annahme, dass es sich jeweils um einen mehrstufigen Sekundärindex handelt mit Blockungsfaktor 20. Die Datensätze zu einem Indexeintrag sind willkürlich verstreut.

Zum Lesen aller Datensätze in der Datei `MIT` mit dem Fach "Java" werden demnach höchstens $3 + 4 + 60 = 67$ Lesezugriffe benötigt.

1. Zuerst wird die Selektion der Relation `MIT` durchgeführt. Das Lesen aller "Java"-Datensätze von `MIT` und anschließende Schreiben des Zwischenergebnisses `Z1` erfordert

   $L_1 = 67$ Lesezugriffe und
   $S_1 = 60/8 = 8$ Schreibzugriffe.

2. `Z1` wird nach `ANR` sortiert und ergibt das Zwischenergebnis `Z2`. Dies erfordert etwa

   $X_2 = 8 \lg_2(8) = 24$

Lese- und Schreibzugriffe.

3. Der Sekundär-Index auf `ANR` habe eine B^+-Struktur. Daher lässt sich die Datei effektiv in der Sortierreihenfolge auslesen. Dabei werden $200/20 = 10$ Blätter und $200/10 = 20$ Daten-Blöcke gelesen. Zusammen mit dem Lesen des Zwischenergebnisses `Z2` und Schreiben des Zwischenergebnisses `Z3` erfordert das bei einem Merge-Join

   $L_3 = 10 + 20 + 8 = 38$ Lesezugriffe und
   $S_3 = 60/4 = 15$ Schreibzugriffe.

4. Bei der abschließenden Projektion muss das Zwischenergebnis `Z3` gelesen werden. Es wird eine Seite als Resultat zurückgeschrieben.

   $L_4 = 15$ Lesezugriffe und
   $S_4 = 1$ Schreibzugriff.

Insgesamt werden in dieser Variante

$$L_1 + L_3 + S_1 + S_3 + X_2 = 168$$

Plattenzugriffe benötigt. Wir haben noch einmal eine Verbesserung um den Faktor 31 erreicht.

28.2 Optimierprozess

Der Optimierprozess besteht aus vier Schritten:

1. Übersetzung der Anfrage in eine **interne** Repräsentation
2. Umwandlung des Ausdrucks in eine **kanonische** Form mit Transformationsregeln (**logische** oder **algebraische** Optimierung)
3. Auswahl von ***low-level*** Prozeduren (Interne Optimierung)
4. Erzeugung von Anfrageplänen mit Kostenschätzung und Auswahl des billigsten Plans

Im folgenden werden diese vier Schritte näher beschrieben.

28.3 Interne Repräsentation

Eine Anfrage wird überlicherweise in einer Hochsprache z.B. SQL formuliert. Bei relationalen Systemen ist die Sprache der internen Repräsentation die relationale Algebra.

Zugriffe auf Sichten werden durch die Sichtdefinition ersetzt **(Sichtexpansion, *view expansion*)**.

28.3.1 Operatorenbaum

Die Darstellung der relationalen Ausdrücke ist ein **Operatorenbaum** ***(abstract syntax tree, query tree, query graph)***, wobei die Operatoren als innere Knoten und die Basistabellen als Blätter notiert werden.

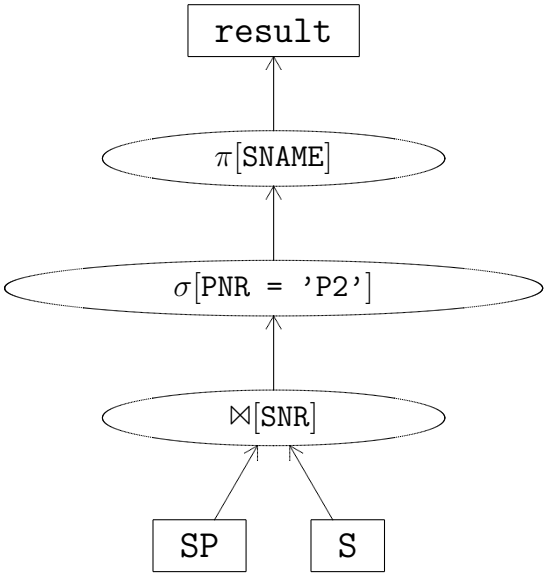
Für die SQL-Anfrage

```
select distinct SNAME
   from SP join S using(SNR)
   where PNR = 'P2'
```

ergibt sich der algebraische Ausdruck:

π[SNAME] (σ[PNR = 'P2'] (SP $\bowtie$[SNR] S))

und der Operatoren-Baum:



Eine wichtige Erweiterung der relationalen Algebra ist der γ-Operator (`group by`-Operator). Normalerweise wird dieser Operator sehr weit oben im Baum erscheinen. Die Optimierung wird versuchen, diesen Operator so weit wie möglich nach unten wandern zu lassen, weil er i.a. die Anzahl der Tupel drastisch reduziert.

Viele Ausdrücke sind einander äquivalent. Nur *ein* Ausdruck hat **kanonische Form** ***(canonical form)***. Anwendung der im nächsten Abschnitt vorgestellten Transformationsregeln ergibt die kanonische Form.

28.3.2 Entschachtelung von Anfragen

Bei der Entschachtelung ***(transformation to non-nested form)*** von Anfragen geht es darum, Subselects in Verbundoperationen umzuwandeln. Man kann dann gemeinsame Teilanfragen leichter erkennen. Außerdem ergibt sich eine Performanzverbesserung, indem geschachtelte Iterationen durch effizientere Verbundoperationen ersetzt werden.

Die verschiedenen Entschachtelungen sollen nur durch Beispiele verdeutlicht werden. Wegen einer formalen Diskussion sei auf den Artikel von W. Kim[27] verwiesen.

Typ-A-Schachtelung:

Beispiel:

```
select   PNR
   from  SP join S using(SNR)
   where STATUS in (select max (STATUS)
                    from  S
                    group by CITY);
```

Die Typ-A-Schachtelung ist dadurch charakterisiert, dass die **innere** Anfrage (Subselect) ein oder mehrere Aggregate (daher "A") berechnet und **keine** Relation der **äußeren** Anfrage referenziert.

Solch eine Anfrage kann ausgewertet werden, indem zuerst der innere Block ausgewertet wird. Das Ergebnis wird dann als Konstante (bei *einem* Aggregat) oder als Tabelle in die äußere Anfrage eingesetzt.

```
T = select distinct  max(STATUS) from  S group by CITY;
```

```
select   PNR
   from  SP join S using(SNR), T
   where STATUS = T.STATUS;
```

Etwas formaler:

$$\sigma_{A\in(\gamma_{f(B);C}R)}S$$
$$\Longrightarrow$$
$$T = \gamma_{f(B);C}R$$
$$S \bowtie_{S.A==T.B} T$$

Typ-N-Schachtelung

Beispiel:

```
select distinct   PNR
   from  SP join S using(SNR)
   where STATUS in (select STATUS
                    from  S
                    where CITY = 'London');
```

Der Typ-N unterscheidet sich von Typ-A dadurch, dass in der inneren Anfrage keine Aggregate gebildet werden, sondern mehrere (d.h. n, daher "N") Tupel resultieren. Relationen der äußeren Anfrage werden nicht referenziert.

Grundsätzlich kann man eine solche Anfrage immer in eine äquivalente Anfrage mit Verbund umformen:

```
select distinct   PNR
   from  SP join S using(SNR), S as S2
   where S.STATUS  = S2. STATUS
      and   S2.CITY = 'London';
```

Etwas formaler:

$$\sigma_{A\in(\pi_B\sigma_\varphi R)}S$$
$$\Longrightarrow$$
$$S \bowtie_{A==B\wedge\sigma_\varphi} R$$

Typ-J-Schachtelung

Beispiel:

```
select distinct   PNR
   from  P join SP using(PNR) join S using(SNR)
   where STATUS in (select STATUS from S
                    where P.CITY = S.CITY);
```

Typ-J ist dadurch charakterisiert, dass die innere Anfrage auf Relationen der äußeren Anfrage verweist (**verzahnt geschachtelte** oder **korrelierte** Unteranfragen). Dies lässt sich umformen in:

```
select distinct   PNR
   from  P join SP using(PNR) join S using(SNR), S as S2
   where S.STATUS = S2.STATUS
      and   P.CITY = S2.CITY;
```

Etwas formaler:

$$\sigma_{A \in (\pi_B(\sigma_{S.C==R.D}R))} S$$
$$\Longrightarrow$$
$$S \bowtie_{A==B \wedge C==D} R$$

Typ-JA-Schachtelung

Beispiel:

```
select   PNR
   from  P join SP using(PNR) join S using(SNR)
   where STATUS = (select  max(STATUS) from  S
                    where P.CITY = S.CITY);
```

Typ-JA ist dadurch charakterisiert, dass der innere Block auf den äußeren Block verweist und ein Aggregat liefert. Dies lässt sich nur schrittweise umformen. Wie bei Typ-A gibt es kein äquivalentes SQL-Statement:

```
T = select  max(STATUS) as MS, CITY
      from  S
      group by CITY;
```

```
select   PNR
   from  P join SP using(PNR) join S using(SNR)
   where STATUS = (select  MS from  T
                     where P.CITY = T.CITY);
```

Letztere Anfrage ist wieder vom Typ-J und kann in

```
select   PNR
   from  P join SP using(PNR) join S using(SNR), T
   where STATUS = MS
      and   P.CITY = T.CITY;
```

umgeformt werden.

Etwas formaler:

$$\sigma_{A==(\gamma_{f(B)}(\sigma_{S.C==R.D}R))}S$$
$$\Longrightarrow$$
$$T = \gamma_{f(B)\,\mathtt{as}\,b;D}R$$
$$\sigma_{A\in(\pi_b(\sigma_{S.C==T.D}T))}S$$
$$\Longrightarrow$$
$$T = \gamma_{f(B)\,\mathtt{as}\,b;D}R$$
$$S \bowtie_{A==b \wedge C==D} T$$

Zusammenfassung

Typ der Schachtelung	Subselect liefert	Subselect referenziert äußere Relationen	Entschachtelung
A	ein oder mehrere Aggregate	nein	zwei Selects
N	n Tupel ohne Aggregate	nein	ein Select mit zusätzlichem Join
J	n Tupel ohne Aggregate	ja	ein Select mit zusätzlichem Join
JA	n aggregierte Tupel	ja	zwei Selects, Resultate werden gejoined

28.4 Transformations-Regeln, Algebraische Optimierung

Die in den algebraischen Ausdruck übersetzte SQL-Anfrage wird mittels **Transformations-Regeln** in die kanonische Form gebracht. Das wird oft auch als logische, algebraische oder regelbasierte Optimierung bezeichnet, da die interne Struktur der Datenbank nicht berücksichtigt wird. Die Regeln lauten:

1. Verschmelzung von Selektionen:

   $\sigma_{\varphi_1}(\sigma_{\varphi_2}R)$
   $\Longrightarrow$
   $\sigma_{\varphi_1 \wedge \varphi_2}R$

2. Verschmelzung von Projektionen:

   $\pi_{A_1}(\pi_{A_2}R)$
   $\Longrightarrow$
   $\pi_{A_1}R$
   wenn $A_1 \subseteq A_2$

   Spezialfall triviale Projektion:

   $\pi_A R$
   $\Longrightarrow$
   R
   wenn $A \equiv$ `Attribute von`R

3. Selektion vor Projektion:

   $\sigma_\varphi(\pi_A R)$
   $\Longrightarrow$
   $\pi_A(\sigma_\varphi R)$
   wenn
   $\varphi(A)$, d.h. φ hängt **nicht** von Nicht-Projektionsattributen ab.

   Wegen der Elimination von Duplikaten ist es meistens gut, eine Restriktion früher als eine Projektion zu machen.

4. Distribution der Selektion: Ein unärer Operator `f` distributiert über einen binären Operator `g`, wenn gilt:

   ```
   f(X g Y) ≡  f(X) g f(Y)
   ```

   Selektion distributiert über Vereinigung, Schnitt und Differenz. ("σ" distributiert über "$\cup$", "$\cap$" und "$-$".) Selektion distributiert auch über Join, sofern das Selektionsprädikat φ in zwei durch $\wedge$ verknüpfte Teile für jeden Join-Partner aufgespalten werden kann. Selektionen sollten so früh wie möglich durchgeführt werden, d.h. σ sollte nach Möglichkeit distributiert werden.

$\sigma_{\varphi}(R \bowtie_{\varphi_J} S)$
$\Longrightarrow$
$(\sigma_{\varphi_1} R) \bowtie_{\varphi_J} (\sigma_{\varphi_2} S)$
wenn
$\varphi = \varphi_1 \wedge \varphi_2$ und
$\varphi_1(r \in R)$ und
$\varphi_2(s \in S)$

Bemerkung: Wenn $\varphi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$, wobei φ_1 und φ_2 oben genannte Bedingungen erfüllen, φ_3 aber nicht, so kann man φ_3 mit in die Join-Bedingung φ_J nehmen: $(\sigma_{\varphi_1} R) \bowtie_{\varphi_J \wedge \varphi_3} (\sigma_{\varphi_2} S)$

5. Distribution der Projektion: Die Projektion distributiert über Vereinigung und Schnitt. ("π" distributiert über "$\cup$" und "$\cap$".) Die Projektion distributiert auch über Join, sofern die Join-Attribute in der Projektion vorkommen. Formal:

   $\pi_A(R \bowtie_{\varphi(D)} S)$
   $\Longrightarrow$
   $(\pi_{A_1} R) \bowtie_{\varphi(D)} (\pi_{A_2} S)$
   wenn
   $A = A_1 \cup A_2$ und
   $D \subseteq A_1 \cap A_2$ und
   $A_1 \subseteq$ `Attribute von` R und
   $A_2 \subseteq$ `Attribute von` S

   Es ist meistens gut, Projektionen vor einem Join zu machen, da durch Elimination von Duplikaten eventuell weniger Tupel zu joinen sind.

6. Vereinigung und Schnitt ($\cup$ und $\cap$) sind **kommutativ** und **assoziativ**. Der Optimierer kann daher frei über die Reihenfolge entscheiden.

   Der Join ist i.a. nur kommutativ! Nur natürliche Joins sind assoziativ. Diese haben aber keine praktische Bedeutung.

7. Vereinigung und Schnitt ($\cup$, $\cap$) sind **idempotent**.

   $R \cup R \Longrightarrow R$
   $R \cap R \Longrightarrow R$

   Beim Join ist nur der natürliche Join idempotent.

8. Bei `extend` und `summarize` müssen auch skalare Ausdrücke optimiert werden.

   $a \cdot b + a \cdot c \equiv a \cdot (b + c)$

9. Bedingungsausdrücke und Prädikte sind auch zu optimieren. Sie sind in die **konjunktive Normalform (KNF)** ***(conjunctive normal form, CNF)*** zu transformieren:

   `c` $\Longrightarrow$ `c1 and c2 and c3 ...`
   $\varphi \Longrightarrow \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \ldots$

   Sobald ein `ci` oder φ_i falsch ist, kann man aufhören. Der Optimierer nimmt das einfachste `ci` oder φ_i zuerst.

10. Semantische Transformationen: Es gilt

    π`[PNR] (S` $\bowtie$`[SNR] SP)` $\Longrightarrow$ π`[PNR] SP`

    nur deshalb, weil das ein Join über eine Fremdschlüssel-Schlüssel-Beziehung ist und {`PNR`} nur eine Teilmenge der Attribute von `SP` ist.

    Allgemein gilt:

    $\pi_A(R \bowtie_D S) \Longrightarrow \pi_A S$
    wenn
    D Schlüssel in R und Fremdschlüssel in S und
    $A \subseteq$ (`Attribute von` S) und
    $A \cap$ (`Attribute von` R) $==$ (`leere Menge`)

    Transformationen, die wegen einer Integritätsbedingung gültig sind, heißen **semantische** Transformationen.

11. **Gruppierungen** $\gamma_{f;A}$ mit f Liste von Aggregat-Funktionen und A Liste von Gruppierungsattributen werden typischerweise als letzte Operation ausgeführt, d.h. als Wurzel des Operatorenbaums. Da eine Gruppierung im Nomalfall die Kardinalität eines Zwischenergebnisses reduziert, sollte versucht werden, sie vorzuziehen. Dabei lassen sich zwei Varianten unterscheiden.

    - **Invariante Gruppierung:** Der Gruppierungsoperator – er bleibt unverändert – wird soweit im Baum nach unten bewegt, bis er direkt oberhalb eines Knotens κ plaziert ist, der folgende Eigenschaften hat:
      (a) Alle Attribute, über die in der Anfrage aggregiert wird, sind so genannte Aggregatkandidaten von κ, d.h. sind Attribute, die weder Verbund-, Selektions- oder Gruppierungsattribute der Anfrage sind.
      (b) Jedes Verbundattribut von κ ist gleichzeitig Gruppierungsattribut der Anfrage.
      (c) Alle nachfolgenden (im Baum oben) Verbunde sind Gleichverbunde über Fremdschlüsselbeziehungen.
    - **Vorgruppierung (Verschmelzungsgruppierung *(coalescing grouping)***: Die Anwendungsmöglichkeiten der invarianten Gruppierung sind wegen der genannten Bedingungen sehr eingeschränkt.

      Wenn man die zweite Bedingung fallen lässt, so könnten zu den Tupeln, die die Gruppen repräsentieren, mehrere Ergebnistupel produziert werden. Diese müssen dann wieder durch eine weitere Gruppierung (Vorgruppierung, Verschmelzung) zusammengefasst werden.

    Wir begnügen uns bei diesem Randgebiet mit der Darstellung der Theorie. Wegen Beispielen verweisen wir auf die Literatur[34].

Algorithmus: Wende die Regeln in der angegebenen Reihenfolge iterativ solange an, bis keine Ersetzungen mehr möglich sind:

```
found = true
while found
```

```
found = false
for rule = Regel1 to Regel11
   if rule is applicable
      found = true
      apply rule
      break // for rule
```

Dabei werden Selektionen und Projektionen möglichst weit in Richtung Blätter (des Operatoren-Baums) verschoben.

Das ist ein Algorithmus, der brauchbare Ergebnisse liefert. Für ein kommerzielles System muss allerdings noch wesentlich mehr "Fingerspitzengefühl" und "Know-How", das sich nicht leicht formalisieren lässt, eingesetzt werden.

Auf die Verbundoptimierung mit Tableaus gehen wir nicht ein[34].

28.5 Auswahl von low-level Prozeduren

Für jede low-level Operation (σ, π, $\bowtie$)

hat der Optimierer eine Menge vordefinierter, implementierte Prozeduren, die jeweils eine Kostenformel haben.

Einige Prozeduren wurden im Kapitel "Basisalgorithmen für Datenbankoperationen" mit den anfallenden Kosten und der Kardinalität des Ergebnisses vorgestellt.

28.6 Erzeugung von Anfrageplänen

Es werden Anfragepläne erzeugt und deren Kosten berechnet. Der billigste Plan wird genommen. Allerdings kann es sein, dass die Anzahl der Pläne kombinatorisch wächst, so dass die Kosten für das Finden des billigsten Plans zu hoch werden.

Das größte Problem dabei ist, die optimale Reihenfolge der Verbundoperationen zu finden. Dafür werden im allgemeinen folgende Optimierungsmethoden eingesetzt:

Greedy-Suche: Es werden die Kardinalitäten der Ergebnisrelationen aller möglichen Kombinationen von Verbunden aus zwei Relationen abgeschätzt. Der Verbund mit der kleinsten Ergebniskardinalität wird als erster berechnet. Der nächste Verbund wird auf dieselbe Art aus den übriggebliebenen Verbunden bestimmt.

Das Verfahren ist einfach. Nachteil ist aber, dass das Minimum nicht garantiert ist.

Dynamische Programmierung: Das Verfahren beruht darauf, dass eine optimale Lösung nur optimale Teillösungen enthalten kann. Wir verzichten auf eine detailierte Darstellung des relativ komplexen Algorithmus.

Als ein einfaches Beispiel für einen Plan betrachten wir die folgende Anfrage:

```
select *
   from  S
   where CITY = 'Paris'
      and (SNAME = 'Mayer' or SNAME = 'Müller')
      and STATUS > 20;
```

Ein Anfrageplan wäre:

$\sigma^{\texttt{REL}}_{\texttt{CITY='Paris'}\wedge(\texttt{SNAME='MAYER'}\vee\texttt{SNAME='Müller'})\wedge\texttt{STATUS>20}} S$

$\rightarrow$ Kosten: b_S

Falls es keine Zugriffsstrukturen gibt, ist dieser Anfrageplan eine vernünftige Wahl.

Falls wir einen Index $I_{\texttt{CITY,S}}$ haben, dann kann man folgende Ersetzung vornehmen:

$\sigma^{\texttt{REL}}_{(\texttt{SNAME='MAYER'}\vee\texttt{SNAME='Müller'})\wedge\texttt{STATUS>20}} \rho \sigma^{\texttt{IND}}_{\texttt{CITY='Paris'}, I_{\texttt{CITY,S}}}$

$\rightarrow$ Kosten: $l_{I_{\texttt{CITY}}} + \frac{|S|}{v_{\texttt{CITY,S}}}$
Wenn wir das Ergebnis der Index-Selektion nicht zwischenspeichern, sondern gleich mit der Relationen-Scan-Selektion verschmelzen, dann waren das die Kosten. Bei einer Zwischenspeicherung würden zusätzlich noch die Kosten $\frac{|S|}{f_S \cdot v_{\texttt{CITY,S}}}$ anfallen.

Wenn es noch einen weiteren Index gibt, nämlich $I_{\texttt{SNAME},S}$, dann können wir unter Ausnutzung der Tatsache, dass Mengenoperationen auf TID-Listen sehr effizient ausführbar sind, folgende effizientere Ersetzung vornehmen:

$\sigma^{\texttt{REL}}_{\texttt{STATUS>20}} \rho(\sigma^{\texttt{IND}}_{\texttt{CITY='Paris'}, I_{\texttt{CITY,S}}} \cap (\sigma^{\texttt{IND}}_{\texttt{SNAME='Mayer'}, I_{\texttt{SNAME,S}}} \cup \sigma^{\texttt{IND}}_{\texttt{SNAME='Müller'}, I_{\texttt{SNAME,S}}}))$

$\rightarrow$ Kosten: $l_{I_{\texttt{CITY}}} + 2 l_{I_{\texttt{SNAME}}} + \min(\frac{|S|}{v_{\texttt{CITY,S}}}, 2\frac{|S|}{v_{\texttt{SNAME,S}}})$

Die Regel "Setze so viele Indexe wie möglich ein!" scheint vernünftig. Betrachten wir aber folgendes Beispiel:

```
select *
   from  S
   where STATUS > 20
      or SNAME = 'Kahn';
```

Der Ausführungsplan ohne Indexe wäre:

$\sigma^{\texttt{REL}}_{\texttt{STATUS>20}\vee\texttt{SNAME='Kahn'}} S$

Der Aufwand ist ein voller Relationenscan.

Wenn wir jetzt den Index $I_{\texttt{SNAME,S}}$ einsetzten, würden wir

$(\sigma^{\texttt{REL}}_{\texttt{STATUS>20}} S) \cup (\rho\sigma^{\texttt{IND}}_{\texttt{SNAME='Kahn'}}, I_{\texttt{SNAME,S}})$

erhalten. Da für die Auswertung des ersten Terms ein voller Relationenscan benötigt wird, bedeutet der Einsatz des Indexes im zweiten Term nur zusätzlichen Aufwand und führt zu einer Erhöhung der Kosten.

28.7 Pipelinig und Verschmelzung von Operatoren

28.7.1 Pipelining

Wenn die Ergebnisse einzelner Operatoren auf der Platte zwischengespeichert werden müssen, dann führt das zu hohen Auswertezeiten. Daher sollte ein Operator jedes Ergebnistupel direkt an seinen Nachfolger als Eingabetupel weiterleiten. Diese Technik wird ***Pipelining*** oder **strombasierte Verarbeitung** genannt.

Am einfachsten lässt sich das realisieren, wenn jeder Operator ein **Iterator-Protokoll** mit den drei Funktionen `open`, `next` und `close` befolgt.

28.7.2 Verschmelzung von Operatoren

Um die Materialisierung von Zwischenergebnissen zu vermeiden, können Operatoren eventuell zusammengefasst ("verschmolzen") werden. Oft bietet sich dafür der Selektionsoperator an:

- Kombination von Selektion und Projektion
- Kombination einer Selektion mit einer Verbundberechnung
- Integration einer Selektion in die äußere Schleife eines Nested-Loops-Verbundes
- Integration von Selektionen in den Merge-Join
- Kopplung der Selektion mit der Realisierung

28.8 Übungen

28.8.1 Schachtelungs-Typen

Welche Schachtelungs-Typen finden Sie in folgendem SQL-Statement?

```
select distinct A.SNR
   from V as A
   where (select count(*) from W)
      = (select count(W.PNR) from V as B, W
            where A.SNR = B.SNR and W.PNR = B.PNR);
```

(Das Statement bezieht sich auf die Tabellen $V[SNR, PNR]$ und $W[PNR]$.)

Oder interessanter wird das Beispiel, wenn wir statt W das erweiterte $KW[KNR, PNR]$ verwenden:

```
select distinct K.KNR, S.SNR, K.C
   from (select KW.KNR, count(*) as C from KW group by KW.KNR) as K,
      (select KW.KNR, V.SNR, count(KW.PNR) as C from KW, V
            where KW.PNR = V.PNR group by KW.KNR, V.SNR) as S
   where K.C = S.C;
```

28.8.2 Entschachtelung

Formulieren Sie ein SQL-Statement für eine Anfrage zu folgender Tabelle:

ID	FID
1	30
2	30
3	31
4	32
5	33

Gesucht sind alle IDs und FIDs, wo die Beziehung ID–FID nicht ein-eindeutig ist, d.h. wo es zu einem FID mehrere IDs gibt.

Wenn Sie ein Subselect verwendet haben, welche(r) Schachtelungstyp(en) liegen vor?

Wie kann man entschachteln?

28.8.3 Logische Optimierung und Kostenschätzung

Gegeben sei folgende SQL-Anweisung:

```
select distinct STATUS, WEIGHT, QTY, PNR, SNR
   from  S join SP using(SNR) join P using(PNR)
   where STATUS < 20 and QTY > 200 and WEIGHT > 15;
```

Übersetzen Sie diese Anweisung in eine interne Repräsentation, d.h. einen Ausdruck der relationalen Algebra und wenden Sie dann die Transformationsregeln an, um die kanonische, d.h. algebraisch optimierte Form zu erhalten.

Diskutieren Sie die Kosten unter folgenden Annahmen:

$|S| = 1000$
$|SP| = 10\,000\,000$
$|P| = 10\,000$
$b_m = 10$
$f_S = 10$
$f_{SP} = 50$
$f_P = 5$
$\text{STATUS}_{min} = 10$
$\text{STATUS}_{max} = 30$
$\text{QTY}_{min} = 100$
$\text{QTY}_{max} = 400$
$\text{WEIGHT}_{\text{min}} = 12$
$\text{WEIGHT}_{\text{max}} = 19$

Kapitel 29

Verteilte Datenbanken

Laufen Daten-Server-Prozesse auf unterschiedlichen Rechnern, dann spricht man von einer **verteilten Datenhaltung** oder einem **verteilten Datenbanksystem, (DDBS *distributed database system,* verteiltes Datenbank-Management-System, DDBMS *distributed database management system)***. Der Datenbestand wird fragmentiert. Dabei unterscheidet man:

Replikation: Dieselben Daten werden auf mehreren Servern gehalten aus Gründen der Effizienz oder Ausfallsicherheit.

Verteilung oder Partitionierung: Bestimmte Teile der Daten werden auf bestimmten Servern gehalten.

In jedem Fall gilt für alle Datenbankdaten ein gemeinsames Datenbank-Schema. Konzeptuell gibt es z.B. nur *eine* "Lieferanten"-Relation, auch wenn diese oft repliziert wird oder auf verschiedene Rechner verteilt ist. Der Benutzer der verteilten DB merkt davon nichts. Für ihn verhält sich die verteilte DB wie ein normales, zentrales DBS **(Verteilungstransparenz)**.

29.1 Architektur

Die bekannten neun Regeln von Codd für RDBS wurden von Date[13] auf 12 Regeln für DDBS erweitert. Dabei wird oft die oben genannte Verteilungstransparenz als Regel 0 vorangestellt.

Die dreistufige ANSI-SPARC-Architektur gilt natürlich auch für DDBS. Der Unterschied zu einem zentralen DBS wird deutlich in der Struktur der internen Ebene, die die verschiedenen Komponenten-DBSs zu verwalten hat. Da ein Komponenten-DBS normalerweise auch eine ANSI-SPARC-Architektur hat, spricht man bei der externen und konzeptuellen Ebene der DDBS auch von **globalen externen Ebenen** bzw der **globalen konzeptuellen Ebene**.

Es ergeben sich dann folgende Ebenen:

(Globale) externe Ebenen: Die (globalen) externen Schemata beschreiben anwendungsspezifische Darstellungen der Daten. Sie werden ausschießlich auf dem (globalen) konzeptuellen Schema definiert.

(Globale) konzeptuelle Ebene: Das (globale) konzeptuelle Schema **(GKS)** beschreibt den integrierten globalen Datenbestand ohne Hinweise auf die Verteilung der Daten. Auf dieser Ebene wird Verteilungstransparenz garantiert.

(Globale) interne Ebene: Diese Ebene besteht nun aus verschiedenen genormten Ebenen:

Fragmentierungsebene: Das Fragmentierungsschema beschreibt die Aufteilung der Datenbankobjekte auf Teilobjekte, z.B. Relationen auf Teil-Relationen.

Wenn nur ganze Relationen verteilt werden, entfällt diese Ebene praktisch.

Allokationsebene: Das Allokationsschema beschreibt die Zuordnung von Teilobjekten (Fragmenten) zu Komponenten-DBSs (Knoten, konkrete Rechner). Wird ein Teilobjekt mehreren Knoten zugeordnet, spricht man von Replikation.

Ebene der Komponenten-DBS: Jeder Knoten ist wieder ein DBS, dessen externe Ebene eine **Transformations-Ebene *(mapping)*** ist, deren Schemata die Vereinheitlichung heterogener Datendarstellungen übernimmt (wegen unterschiedlicher Anfrage-Sprachen oder SQL-Dialekten). Ansonsten existieren dort (lokale) konzeptuelle und (lokale) interne Ebenen. Die Autonomie der lokalen Systeme ist insofern stark eingeschränkt, als dass es keine anderen lokalen externen Schemata gibt, die einen lokalen Zugriff ermöglichen.

Im folgenden gehen wir genauer auf die Fragmentierungs- und Allokationsebene, verteilte Anfragebearbeitung und verteilte Transaktionen ein.

29.2 Fragmentierung und Allokation

Die folgende Diskussion bezieht sich nur auf relationale DBS.

29.2.1 Horizontale Fragmentierung

Eine Relation wird durch die Definition von geeigneten Restriktionen in *disjunkte* Teil-Relationen *vollständig* partitioniert. Z.B. die Relation S der Lieferanten wird aufgeteilt in die Lieferanten mit Status größer 15 und die Lieferanten mit Status kleiner-gleich 15.

29.2.2 Vertikale Fragmentierung

Eine Relation wird durch die Definition von geeigneten Projektionen in Teil-Relationen aufgespalten, wobei in den Teil-Relationen immer der Primäschlüssel übernommen werden muss, damit die Ursprungsrelation korrekt wiederhergestellt werden kann. Z.B. könnte die Relation S der Lieferanten in die Relationen {SNR, SNAME}, {SNR, STATUS} und {SNR, CITY} aufgeteilt werden.

29.2.3 Gemischte Fragmentierung

Hier wird eine Relation horizontal und vertikal fragmentiert.

29.2.4 Abgeleitete Fragmentierung

Analog zur Clusterung über Fremdschlüssel macht es auch bei der Verteilung von Daten Sinn, semantisch zusammengehörige Fragmente verschiedener Relationen gemeinsam zu allokieren. Dabei wird eine Relation – im folgenden R_r mit Primärschlüssel `K` – üblicherweise horizontal fragmentiert **(Referenzfragmentierung)**. Die Primärschlüssel der Fragmente bilden disjunkte Mengen und bestimmen dann über Fremdschlüsselbeziehungen die Fragmentierung anderer Relationen (R_i), die den Primärschlüssel von R_r als Fremdschlüssel enthalten. Die Fragmente von R_i (F_{R_i}) ergeben sich dann aus den Fragmenten von R_r (F_{R_r}) formal zu:

$$F_{R_i} = R_i \bowtie_K (\pi_K F_{R_r})$$

Die Fragmente von R_i sind dann auch disjunkt. Ferner ist die Partitionierung vollständig, wenn `K` ein echter Fremdschlüssel in R_i ist, d.h. dort niemals `NULL` wird.

29.3 Replikation

Wichtigstes Ziel eines normalisierten Datenbankentwurfs ist die Redundanzfreiheit der Daten. Aus Gründen der

- Effizienz
- Ausfallsicherheit
- Autonomie (Knoten, die nicht ständig am Netz sind.)

wird bei verteilten Systemen davon *kontrolliert* abgewichen, d.h. Daten werden mehreren Rechnerknoten zugewiesen **(Replikation)**.

Das Problem dabei ist natürlich die Erhaltung der Datenkonsistenz, wofür die Verfahren der **Replikationskontrolle** zur Verfügung stehen:

Master-Slave (Primary Copy): Es gibt eine ausgezeichnete Replikation, die Primärkopie, auf der alle Aktualisierungen (Datenänderungen) zuerst durchgeführt werden. Die Aktualisierung der anderen Replikate wird zu einem späteren Zeitpunkt nachgeholt. Anfrageoperationen dürfen auf nicht aktuellen Replikaten laufen. Das hat zur Folge, dass die Konsistenz der Daten innerhalb von Anfragen nicht unbedingt gewährleistet ist.

Majority-Consensus: Beim **Abstimmungsverfahren** nehmen die Rechnerknoten an einer Abstimmung darüber teil, ob ein Zugriff auf replizierte Daten erlaubt werden kann oder nicht. Wenn eine entscheidungsfähige Anzahl **(Quorum)** zustimmt, kann der Zugriff ausgeführt werden. Je nach Zugriff kann es Lese- und Schreibquoren geben.

Eine Replikation von Daten ist normalerweise nur dann sinnvoll, wenn es wesentlich mehr Lese- als Schreibzugriffe gibt.

29.4 Verteilter Katalog

Ein (globaler) Datenbank-Katalog ist natürlich auch ein Teil des Datenbestandes und kann fragmentiert, repliziert und allokiert werden. Wir können folgende Möglichkeiten haben:

- Ein **zentralisierter** Katalog wird auf einem Rechner zentral gehalten. (Das widerspricht allerdings der Forderung von Date, möglichst keine zentralen Dienste in einem verteilten System zu haben.)
- Ein **vollredundanter** Katalog wird vollständig auf allen Knoten repliziert. Änderungen am Katalog können dann allerdings nur wenig effizient durchgeführt werden.
- Ein **Cluster**- oder **Mehrfach**katalog ist ein Kompromiss zwischen den ersten beiden Möglichkeiten. Falls das Netz in Teilnetze gegliedert ist, dann wird der Katalog pro Teilnetz einmal repliziert.
- Schließlich können ausschließlich **lokale** Kataloge gehalten werden, die zu einem **virtuellen globalen** Katalog integriert werden. Zur Übersetzung einer Anfrage müssen dann Katalogdaten über das Netz verschickt werden. Daher werden Katalogdaten oft in einem Cache gehalten.

29.5 Verteilte Transaktionen

Bei verteilten Transaktionen erfordern das Commit und die Erkennung von Deadlocks besondere Algorithmen.

29.5.1 Verteiltes Commit

Damit das Commit-Protokoll Atomarität und Dauerhaftigkeit von Transaktionen garantieren kann, müssen folgende Anforderungen gestellt werden:

- Alle Rechnerknoten kommen global zu *einer* Entscheidung. Dabei ist nur Commit oder Rollback möglich.
- Ein Knoten kann eine getroffene Entscheidung nicht mehr rückgängig machen.
- Eine Commit-Entscheidung kann nur getroffen werden, wenn alle Knoten mit "ja" gestimmt haben.
- Treten keine Fehler auf und votieren alle Knoten mit "ja", so lautet die Entscheidung auf Commit.
- Das Protokoll ist robust. D.h. unter den vorgegebenen Randbedingungen terminieren alle Prozesse.

Zur Erfüllung dieser Anforderungen werden zwei Protokolle eingeführt, das Zwei-Phasen-Commit-Protokoll und das Drei-Phasen-Commit-Protokoll.

Zwei-Phasen-Commit-Protokoll

Das Zwei-Phasen-Commit **(2PC, *two-phase-commit)*** ist das meistbenutzte Protokoll. Dabei wird ein *System-weiter* Koordinator bestimmt, der das Protokoll durch Vergeben von Aufträgen an die Teilnehmer (DBMSs) abarbeitet.

Die zwei Phasen sind:

1. Wahlphase: Der Koordinator veranlasst jedes beteiligte DBMS, alle Journaleinträge, die die Transaktion betreffen, auf das physikalische Speichermedium zu schreiben ***(Prepare-Commit)***. Jedes DBMS meldet dann an den Koordinator "OK" ***(Vote-Commit)***, wenn sie ein Commit durchführen können (bzw "NOT OK" ***(Vote-Abort)***, falls das nicht möglich ist).
2. Entscheidungsphase: Wenn der Koordinator von allen beteiligten DBMS eine Antwort hat, dann schreibt er in sein eigenes physikalisches Journal seine Entscheidung. Wenn alle Antworten "OK" waren, ist diese Entscheidung "commit" sonst "rollback". Dann benachrichtigt der Koordinator alle DBMS, die "OK" gestimmt haben, über seine Entscheidung, die dann entsprechend dieser Entscheidung verfahren müssen.

Problematisch bei diesem Verfahren ist der Ausfall des Koordinators. Zwei Varianten versuchen diese Problematik zu lindern. Beim **linearen 2PC** wird das Sammeln der Antworten durch eine lineare Abarbeitung der Teilnehmer ersetzt. Beim **verteilten 2PC** sendet nach Initiation durch den Koordinator jeder Teilnehmer an jeden Teilnehmer seine Entscheidung. Beim **hierarchischen 2PC** werden Sub-Koordinatoren definiert insbesondere um die Kosten von teuren Verbindungen zu minimieren (Siehe [34]).

Drei-Phasen-Commit-Protokoll

Das Drei-Phasen-Commit **(3PC, *three-phase-commit)*** verhindert blockierende Zustände durch Einführung einer ***Pre-Commit***-Phase.

Die drei Phasen sind:

1. Wahlphase: Der Koordinator sendet an alle beteiligten DBMS die Nachricht ***Prepare***.

   Jeder Teilnehmer meldet dann an den Koordinator "OK" ***(Vote-Commit)***, wenn er ein Commit durchführen kann (bzw "NOT OK" ***(Vote-Abort)***, falls das nicht möglich ist).
2. Entscheidungsvorbereitungsphase: Der Koordinator sammelt von allen beteiligten DBMS die Antworten. Wenn alle Antworten "OK" ***(Vote-Commit)*** waren, sendet er an alle Teilnehmer ***Prepare-To-Commit***. Ansonsten sendet er an alle Teilnehmer ein "rollback" ***(Abort)***.

   Jeder Teilnehmer, der mit ***Vote-Commit*** gestimmt hat, wartet auf ein ***Prepare-To-Commit*** oder ein ***Abort***. Bei ***Abort*** entscheidet er auf Abbruch, ansonsten bestätigt er mit ***Ready-To-Commit***.
3. Entscheidungsphase: Der Koordinator sammelt alle Bestätigungen und entscheidet gegebenenfalls auf ***Commit***, das allen Teilnehmern mitgeteilt wird.

   Die Teilnehmer warten auf die Entscheidung des Koordinators und führen dann die entsprechende Operation aus.

Fällt irgendjemand vor der Entscheidungsvorbereitungsphase aus, dann wird die Transaktion abgebrochen.

Hat ein Teilnehmer schon ein ***Prepare-To-Commit*** erhalten, dann übernimmt dieser Teilnehmer bei Ausfall des Koordinators dessen Rolle und die Transaktion kann beendet werden.

29.5.2 Transaktionen auf Replikaten

Für replizierte Datenbanken muss man sich auf eine Definition von Serialisierbarkeit festlegen.

Definition Serialisierbarkeit: Eine verschachtelte Durchführung von Transaktionen auf einer replizierten Datenbank ist **1-Kopie-serialisierbar**, wenn es eine serielle Durchführung auf einer nicht-replizierten Datenbank gibt, die den gleichen Effekt erzeugt wie auf dem replizierten Datenbestand.

Das bedeutet, dass Replikate nicht unbedingt den identischen Inhalt haben. Folgende Verfahren werden eingesetzt:

ROWA-Verfahren *(Read One, Write All)*: Bei ändernden Operationen sollen alle Replikate synchron geändert werden. Das ist ein hoher Aufwand. Eine Variante ist das ROWAA-Verfahren ***(Read One, Write All Available)***, bei dem nur die erreichbaren Replikate verändert werden.

Abstimmungsverfahren: Die schon besprochenen Abstimmungsverfahren gibt es in verschiedenen Varianten (Gewichtung der Stimmen, statische und dynamische Quoren).

Absolutistische Verfahren: Hierzu gehört die Primärkopie-Methode.

29.5.3 Verteilte Verklemmungs-Erkennung

Gerade bei verteilten Systemen ist das Vermeiden oder Erkennen von **Verklemmungen *(deadlock)*** eine wichtige Aufgabe. Ein **verklemmungsfreier** Betrieb kann dadurch erreicht werden, dass alle für eine Transaktion benötigten Resourcen vor Beginn der Transaktion belegt werden. Die Realisierung ist im verteilten Fall allerdings sehr problematisch, da ein Synchronisationsprotokoll benötigt wird. Außerdem ist die atomare Belegung aller Resourcen nur für bestimmte Transaktionen realisierbar. **Verklemmungsprävention** ist möglich, wenn alle DB-Objekte total angeordnet werden. Der Zugriff erfolgt dann nur in der totalen Reihenfolge.

Verklemmungen können in verteilten Systemen auf viele Arten erkannt werden:

Time-Out-Mechanismus: Diese Methode bricht eine wartende Transaktion nach einer bestimmten Wartezeit ab. Das ist einfach zu implementieren, aber wenig flexibel.

Globaler Deadlock-Graph: Ein zentraler Koordinator vereinigt die lokalen Deadlock-Graphen zu einem globalen Deadlock-Graphen. Ein Zyklus zeigt eine Verklemmung an. Problematisch hierbei ist ein Ausfall des Koordinators oder die Aktualisierung des globalen Graphen, wobei Deadlocks erkannt werden, die in Wirklichkeit bereits aufgelöst sind **(Phantom-Deadlock)**.

Zeitmarken als Anforderungs-Ordnung: Bei diesem Verfahren wird die totale Ordnung der Zeitmarken der Transaktionen verwendet. Eine zu spät gekommene Transaktion bricht ab, anstatt auf die Aufhebung der Sperre zu warten.

Globale Deadlock-Erkennung: Wenn eine Transaktion blockiert wird, dann verschickt sie eine Nachricht, die eine Transaktions-Identifikation enthält, an die blockierende Transaktion. Wenn die empfangende Transaktion ihrerseits blockiert wird, schickt sie diese Nachricht weiter an die sie blockierende Transaktion. Diese Nachrichten werden Knotenübergreifend verschickt. Wenn eine Transaktion eine Nachricht mit ihrer Identifikation erhält, dann liegt ein Zyklus, d.h. eine Verklemmung vor.

Kapitel 30

Datenbankanbindung – ODBC

Eine Alternative zu embedded SQL ist die Benutzung einer Bibliothek mit Funktionen, die vom Anwendungsprogramm aus aufgerufen werden können. Eine Standardisierung dieser Funktionen für C als Hostsprache stellt ***Open Database Connectivity*** **(ODBC)** von Microsoft dar.

Die Standardisierung umfaßt eine Bibliothek von Funktionsaufrufen, mit denen eine Verbindung zu einer Datenbank hergestellt werden kann, SQL-Statements abgesetzt und Anfrageresultate erhalten werden können. Ferner ist die Menge der Fehler-Codes standardisiert.

Wir begnügen uns hier mit einem Beispiel in C:

```
#include "SQL.H"
#include <stdio.h>
#include <stdlib.h>
#define  MAX_ANW_LAENGE 256
                        /* Maximale Anweisungslänge      */

main ()
   {
   HENV  umg;           /* Umgebungszeiger               */
   HDBC  con;           /* Datenbankverbindungszeiger    */
   HSTMT anw;           /* Anweisungszeiger              */
   RETCODE  rck;        /* Rückgabecode                  */
   UCHAR sel[MAX_ANW_LAENGE]; /* SELECT-String           */
   UCHAR snr[2];        /* Lieferantennummer             */
   UCHAR name[20];      /* Lieferantenname               */
   UCHAR stadt[20]      /* Lieferantenstadt              */
   INT   status;        /* Lieferantenstatus             */
   SDWORD   snrlg, namelg, stadtlg;

   SQLAllocEnv (&umg);  /* Allokiere Umgebungszeiger     */
   SQLAllocConnect (umg, &con);  /* Allokiere con        */
   rck = SQLConnect (con,
      "SPDatenbank", SQL_NTS,    /* Datenbankname        */
```

```
      "Meyer", SQL_NTS,          /* Benutzername         */
      "Hochgeheim", SQL_NTS      /* Paßwort              */
      );
      /* SQL_NTS sorgt dafür, daß \0 als Stringende erkannt wird. */

   if (rck == SQL_SUCCESS || rck == SQL_SUCCESS_WITH_INFO)
      {
      SQLAllocStmt (con, anw);   /* Allokiert anw        */

      lstrcpy (sel, "SELECT SNR, SNAME, STATUS, CITY FROM S");
                           /* Erstellung der Anfrage     */

      if (SQLExecDirect (anw, sel, SQL_NTS) != SQL_SUCCESS)
         exit (-1);
                           /* Durchführung der Anfrage   */

      SQLBindCol (anw, 1, SQL_C_CHAR, snr, (SDWORD) sizeof (snr), &snrlg);
      SQLBindCol (anw, 2, SQL_C_CHAR, name, (SDWORD) sizeof (name), &namelg);
      SQLBindCol (anw, 3, SQL_C_INT, status);   /* ? */
      SQLBindCol (anw, 4, SQL_C_CHAR, city, (SDWORD) sizeof (city), &citylg);

      while (rck == SQL_SUCCESS || rck == SQL_SUCCESS_WITH_INFO)
                     /* Nun werden die Resultate geholt: */
         {
         rck = SQLFetch (anw);
         if (rck == SQL_SUCCESS || rck == SQL_SUCCESS_WITH_INFO)
            {
            printf (
               "Resultat: SNR = %s, SNAME = %s, STATUS = %d, CITY = %s\n",
               snr, name, status, city);
            }
         }
      SQLFreeStmt (anw, SQL_DROP);
      SQLDisconnect (con);
      }
   SQLFreeConnect (con);
   SQLFreeEnv (umg);
   }
```

Kapitel 31

Datenbankanbindung – JDBC

31.1 Einführung

JDBC^{TM} ***(Java Database Connectivity)*** ist eine Plattform-unabhängige Schnittstelle zwischen Java und Datenbanken, die SQL verstehen. Es ist ein Java-API zur Ausführung von SQL-Anweisungen. Damit ist JDBC ein ***low-level*** API, womit dann ***high-level*** oder benutzerfreundliche Oberflächen geschrieben werden können.

Von JDBC aus gibt es im wesentlichen zwei Entwicklungsrichtungen:

- ***embedded*** SQL für Java: SQL-Anweisungen können mit Java-Code gemischt werden. Java-Variable können in SQL-Anweisungen verwendet werden. Ein Produkt dieser Art ist SQLJ.
- direkte Abbildung von Tabellen als Java-Klassen: Jede Zeile einer Tabelle wird zur Instanz der Klasse der Tabelle ***(object/relational mapping)***. Jede Spalte entspricht einem Attribut der Klasse. Der Programmierer kann dann direkt mit den Java-Objekten und ihren Attributen arbeiten. Die benötigten SQL-Anweisungen werden automatisch generiert.

Das ODBC API *(Open DataBase Connectivity)* von Microsoft gibt es für beinahe jede Datenbank. Eine direkte Übersetzung von ODBC in ein Java-API ist nicht wünschenswert, da ODBC z.B. reichlichen Gebrauch von Pointern macht. Daher gibt es eine JDBC-ODBC-Bridge, die eine Übersetzung von ODBC in JDBC ist. Ferner ist ODBC schwierig zu lernen, indem einfache und komplexe Konzepte vermischt werden. Z.B. ist für einfache Anfragen die Angabe komplizierter Optionen notwendig. JDBC ist viel leichter zu lernen als ODBC. Beide Schnittstellen basieren auf X/Open SQL CLI (Call Level Interface). Die JDBC-ODBC-Bridge wird mit dem Java 2 SDK mitgeliefert.

`ADO.NET` bietet allerdings ähnlich bequeme Mechanismen, um auf Datenquellen zuzugreifen. Statt eine gemeinsame Schnittstelle für alle datenbanken anzubieten, werden für jedes Datenbanksystem spezielle Klassen angeboten.

JDBC unterstützt einstufige und zweistufige Modelle **(2-Tier**, ***two-tier***, **3-Tier**, ***three-tier models***, allgemein **N-Tier-Modelle oder Architekturen)**. Im einstufigen Modell kommuniziert ein Applet oder eine Anwendung direkt mit dem Datenbank-Management-System, das auf

einer anderen Machine laufen kann. Im zweistufigen Modell werden die SQL-Anfragen an einen Anwendungsserver geschickt, der dann über JDBC mit dem DBMS verkehrt. Das hat den Vorteil, dass man mehr Kontrolle über den Zugang zur Datenbank hat und dass man benutzerfreundliche Schnittstellen anbieten kann, die vom Anwendungsserver in JDBC-Aufrufe übersetzt werden.

Wie geht JDBC mit SQL um? Man kann *jedes* SQL-Kommando absetzen. Damit läuft man aber Gefahr, dass der Code von einer Datenbank zur anderen nicht mehr ohne weiteres portierbar ist, da die verschiedenen Datenbanken vom SQL-Standard abweichen. Eine Möglichkeit ist, ODBC-ähnliche Escape-Sequenzen zu benutzen. JDBC bietet noch eine andere Möglichkeit, indem deskriptive Informationen über das DBMS verwendet werden, wozu eine `DatabaseMetaData`-Schnittstelle zu implementieren ist.

Die neueste Information über die Verfügbarkeit von Treibern erhält man über:

```
http://java.sun.com/products/jdbc
```

Die JDBC-Klassen befinden sich im Paket `java.sql` .

31.2 Verbindung `Connection`

Ein Objekt der Klasse `Connection` repräsentiert eine Verbindung mit der Datenbank. Eine Anwendung kann mehrere Verbindungen mit derselben oder verschiedenen Datenbanken haben. Normalerweise wird eine Verbindung mit der statischen Methode `getConnection ()` der Klasse `DriverManager` erzeugt, wobei die Datenbank durch eine URL repräsentiert wird. Der Treibermanager versucht, einen Treiber zu finden, der auf die gewünschte Datenbank passt. Der Treibermanager verwaltet eine Liste registrierter Treiber-Klassen (Klasse `Driver`). Wenn ein Treiber gefunden ist, dann wird mit der Methode `connect` die Verbindung zur Datenbank aufgebaut.

Beispiel:

```
String   url = "jdbc:mysql://rechner:3306/Spielbank";
Connection  verbindung = DriverManager.getConnection
                            (url, "Benutzername", "Passwort");
```

Mit einer JDBC-URL werden Datenbanken identifiziert, wobei der Protokollname `jdbc` verwendet wird. Der Treiberhersteller muss angeben, wie die Datenbank nach dem `jdbc:` identifiziert wird.

Die empfohlene Standard-Syntax einer JDBC-URL besteht aus drei durch Doppelpunkte getrennten Teilen:

`jdbc:<Subprotokoll>:<Subname>`

1. Der Protokollname `jdbc` ist zwingend.
2. Der `<Subprotokoll>`-Name ist der Name des Treibers oder des Datenbank-Verbindungs-Mechanismus. Berühmte Beispiele dafür sind `odbc` oder `mysql`. Es kann aber auch der Name eines Namensdienstes sein. JavaSoft registriert informell die JDBC-Subprotokollnamen. Wenn man einen Treiber geschrieben hat, dann kann man den Namen des Protokolls registrieren lassen, indem man Email an

```
jdbc@wombat.eng.sun.com
```

sendet.

3. Der `<Subname>` identifiziert irgendwie die Datenbank. Er kann z.B. die Form

   ```
   //hostname:port/kfg/db/Spielbank
   ```

   haben.

Das `odbc`-Subprotokoll hat die spezielle Syntax:

`jdbc:odbc:<Datenquellname>[;<Attributname>=<Attributwert>]*`

Damit können beliebig viele Attribute als Parameter übergeben werden. Z.B.:

```
jdbc:odbc:Spielbank;UID=kfg;PWD=Ratatui;CacheSize=20;
```

Weitere Methoden von `DriverManager` sind `getDriver`, `getDrivers` und `registerDriver`.

Ein Treiber kann auf zwei Arten geladen werden:

1. `Class.forName ("TreiberKlassenname");`
   Diese Anweisung lädt die Treiberklasse explizit. Wenn die Klasse so geschrieben ist (Verwendung von statischen Initialisatoren), dass beim Laden der Klasse ein Objekt der Klasse erzeugt wird und dafür die Methode

   ```
   DriverManager.registerDriver (objektDerTreiberKlasse)
   ```

   aufgerufen wird, dann steht dieser Treiber für die Erzeugung einer Verbindung zur Verfügung. Falls das Treiberobjekt nicht automatisch erzeugt wird, ist der Aufruf

   ```
   Class.forName ("TreiberKlassenname").newInstance ();
   ```

   sicherer.

2. Man kann auch den Treiber der `System`-Eigenschaft `jdbc.drivers` hinzufügen. Diese Eigenschaft ist eine durch Doppelpunkt getrennte Liste von Treiber-Klassennamen. Wenn die Klasse `DriverManager` geladen wird, dann wird versucht alle Treiberklassen zu laden, die in der Liste `jdbc.drivers` stehen.

Methode 1) wird empfohlen, da sie unabhängig von einer Umgebungsvariablen ist.

Aus Sicherheitsgründen weiß JDBC, welcher Klassenlader welche Treiberklasse zur Verfügung stellt. Wenn eine Verbindung hergestellt wird, dann wird nur der Treiber benutzt, der vom selben Klassenlader geladen wurde wie der betreffende (die Verbindung anfordernde) Code.

Die Reihenfolge der Treiber ist signifikant. Der `DriverManager` nimmt den *ersten* Treiber, der die Verbindung aufbauen kann. Die Treiber in `jdbc.drivers` werden zuerst probiert.

Beispiel:

```
Class.forName ("org.gjt.mm.mysql.Driver");
String   url = "jdbc:mysql://rechner:3306/Spielbank";
Connection  verbindung
   = DriverManager.getConnection (url, "Bozo", "hochgeheim");
```

31.3 SQL-Anweisungen

Nach dem Verbindungsaufbau wird das `Connection`-Objekt benutzt, um SQL-Anweisungen abzusetzen. JDBC macht bezüglich der Art der SQL-Statements keine Einschränkungen. Das erlaubt große Flexibilität. Dafür ist man aber selbst dafür verantwortlich, dass die zugrunde liegende Datenbank mit den Statements zurechtkommt. JDBC fordert, dass der Treiber mindestens das ANSI SQL2 Entry Level unterstützt, um JDBC COMPLIANT zu sein. Damit kann der Anwendungsprogrammierer sich mindestens auf diese Funktionalität verlassen.

Es gibt drei Klassen um SQL-Anweisungen abzusetzen.

- `Statement`
- `PreparedStatement`
- `CallableStatement`

31.3.1 Klasse `Statement`

Die Klasse `Statement` dient zur Ausführung einfacher SQL-Anweisungen. Ein `Statement`-Objekt wird mit der `Connection`-Methode `createStatement` erzeugt. Mit der Methode `executeQuery` werden dann SQL-Anfragen abgesetzt:

```
Statement   anweisung = verbindung.createStatement ();
ResultSet   ergebnis
   = anweisung.executeQuery ("SELECT a, b, c FROM tabelle");
while (ergebnis.next ())
   {
   int   x = ergebnis.getInt ("a");
   String   y = ergebnis.getString ("b");
   float z = ergebnis.getFloat ("c");
   // Tu was mit x, y, z ...
   }
```

Die Methode `executeQuery` gibt eine einzige Resultatmenge zurück und eignet sich daher für `SELECT`-Anweisungen.

Die Methode `executeUpdate` wird verwendet, um `INSERT`-, `UPDATE`- oder `DELETE`-Anweisungen und Anweisungen von SQL-DDL wie `CREATE TABLE` und `DROP TABLE` auszuführen. Zurückgegeben wird die Anzahl der modifizierten Zeilen (evtl. 0).

Die Methode `execute` bietet fortgeschrittene Möglichkeiten (mehrere Resultatmengen), die selten verwendet werden und daher kurz in einem eigenen Abschnitt behandelt werden.

Nicht benötigte `ResultSet`-, `Statement`- und `Connection`-Objekte werden vom GC automatisch deallokiert. Dennoch ist es gute Programmierpraxis gegebenenfalls ein explizites

```
ergebnis.close (); anweisung.close (); verbindung.close ();
```

aufzurufen, damit Resourcen des DBMS unverzüglich freigegeben werden, womit man potentiellen Speicherproblemen aus dem Weg geht.

Auf die Escape-Syntax, um Code möglichst Datenbank-unabhängig zu machen, wird hier nicht eingegangen.

Ein Objekt der Klasse `ResultSet` enthält alle Zeilen, die das Ergebnis einer `SELECT`-Anfrage sind. Mit

`get<Typ> ("Spaltenname")`-Methoden
oder
`get<Typ> (Spaltennummer)`-Methoden

kann auf die Daten der jeweils *aktuellen* Zeile zugegriffen werden. Mit

```
ergebnis.next ()
```

wird die nächste Zeile zur aktuellen Zeile. `next ()` gibt `true` oder `false` zurück, jenachdem ob die nächste Zeile existiert oder nicht. Der erste Aufruf von `next` macht die erste Zeile zur aktuellen Zeile. (Das ist praktisch für die Gestaltung von Schleifen, da man so die erste Zeile nicht gesondert behandeln muss.)

Benannte Kursoren und positionierte Manipulationen sind auch möglich.

Spaltennummern beginnen bei "1". Verwendung von Spaltennummern ist etwas effektiver und notwendig bei Namenskonflikten. Mit

```
findColumn (Name)
```

kann die Nummer einer Spalte gefunden werden.

Es gibt empfohlene und erlaubte SQL-Java-Datentyp-Wandlungen.

Es ist möglich, beliebig lange Datenelemente vom Typ `LONGVARBINARY` oder `LONGVARCHAR` als Stream zu lesen, da die Methode

```
InputStream ein1 = ergebnis.getBinaryStream (spaltennummer);
InputStream ein2 = ergebnis.getAsciiStream (spaltennummer);
Reader   ein3 = ergebnis.getUnicodeStream (spaltennummer);
```

einen Stream liefert.

Mit der Sequenz

```
ergebnis.get<Typ> (spaltennummer);
ergebnis.wasNull ();
```

kann auf ein NULL-Eintrag in der Datenbank geprüft werden. Dabei gibt `get<Typ>`

- `null` zurück, wenn Objekte zurückgegeben werden,
- `false` bei `getBoolean (spaltennummer)`
- und Null, wenn ein anderer Standardtyp zurückgegeben wird.

Die Methode `wasNull ()` gibt `boolean` zurück.

31.3.2 Methode `execute`

Es kann vorkommen, dass eine gespeicherte Datenbankroutine mehrer SQL-Statements absetzt und damit mehrere Resultatmengen oder Update-Counts erzeugt. Die Einzelheiten des Gebrauchs der verschiedenen beteiligten Methoden sind langwierig. Daher beschränken wir uns darauf, nur für den kompliziertesten Fall, nämlich dass man überhaupt nicht weiß, was die verschiedenen Resultate sind, im folgenden einen Beispiel-Code anzugeben.

```
import   java.sql.*;
public class   JDBCexecute
   {
   public static void   werteAus (Statement anweisung,
      String anfrageStringMitUnbekanntemResultat)
      throws SQLException
      {
      anweisung.execute (anfrageStringMitUnbekanntemResultat);
      while (true)
         {
         int   zeilen = anweisung.getUpdateCount ();
         if (zeilen > 0)
            // Hier liegt ein Update-Count vor.
            {
            System.out.println ("Geänderte Zeilen = " + zeilen);
            anweisung.getMoreResults ();
            continue;
            }
         if (zeilen == 0)
            // DDL-Kommando oder kein Update
            {
            System.out.println (
               "DDL-Kommando oder keine Zeilen wurden geändert.");
            anweisung.getMoreResults ();
            continue;
            }
         // zeilen < 0:
         // Wenn man soweit ist, dann gibt es entweder ein ResultSet
         // oder keine Resultate mehr.
         ResultSet   rs = anweisung.getResultSet ();
         if (rs != null)
            {
            // Benutze ResultSetMetaData um das rs auszuwerten
            // ...
            anweisung.getMoreResults ();
            continue;
            }
         break;   // Es gibt keine Resultate mehr.
         }
```

```
    }
}
```

31.3.3 Klasse `PreparedStatement`

Objekte der Klasse `PreparedStatement` repräsentieren eine kompilierte SQL-Anweisung mit ein oder mehreren `IN`-Parametern. Der Wert eines `IN`-Parameters wird nicht spezifiziert, sondern mit einem Fragezeichen "`?`" als Platzhalter offengelassen. Bevor das Statement ausgeführt wird, muss für jedes Fragezeichen die Methode

```
set<Typ> (parameternummer, parameterWert)
```

aufgerufen werden, z.B. `setInt`, `setString`. `parameternummer` beginnt bei 1. Gesetzte Parameter können für wiederholte Ausführungen der Anweisung verwendet werden.

Die drei Methoden `execute`, `executeQuery` und `executeUpdate` sind so modifiziert, dass sie keine Argumente haben. Die Formen mit Argumenten sollten für Objekte von `PreparedStatement` nie benutzt werden.

Ein Objekt von `PreparedStatement` wird folgendermaßen erzeugt:

```
PreparedStatement prepAnw = verbindung.prepareStatement (
   "UPDATE SP SET QTY = ? WHERE SNR = ?");
```

Weiterführende Hinweise: Mit `setNull` können `NULL`-Werte in der Datenbank gesetzt werden. Mit `setObject` kann im Prinzip jedes Java-Objekt akzeptiert werden und erlaubt generische Anwendungen. Auch gibt es Methoden, mit denen ein `IN`-Parameter auf einen Eingabe-Strom gesetzt werden kann.

31.3.4 Klasse `CallableStatement`

Objekte vom Typ `CallableStatement` erlauben den Aufruf von in einer Datenbank gespeicherten Prozeduren.

Die Syntax für einen Aufruf einer Prozedur ohne Rückgabewert lautet:

```
{call Prozedurname[ (?, ?, ...)]}
```

Die `[]`-Klammern beinhalten optionale Elemente der Syntax. Die Fragezeichen können `IN`- oder `OUT`- oder `INOUT`-Argumente sein. Mit Rückgabewert ist die Syntax:

```
{? = call Prozedurname[ (?, ?, ...)]}
```

Über die Klasse `DatabaseMetaData` kann man die Datenbank bezüglich ihrer Unterstützung von gespeicherten Prozeduren untersuchen (Methoden `supportsStoredProcedures ()` und `getProcedures ()`).

`CallableStatement` erbt von `Statement` alle Methoden, die generell mit SQL-Anweisungen zu tun haben. Von `PreparedStatement` werden Methoden geerbt, die mit `IN`-Argumenten zu tun haben. Neu hinzu kommen Methoden, die mit `OUT`- oder `INOUT`-Argumenten arbeiten.

Objekte werden erzeugt mit:

```
CallableStatement prozAnw = verbindung.prepareCall (
   "{call prozedurName (?, ?)}");
```

Ob die Fragezeichen `IN` oder `OUT` oder `INOUT` sind, hängt von der Definition der Prozedur ab. `OUT`- und `INOUT`-Parameter müssen registriert werden, bevor die Anweisung ausgeführt wird. Nach Ausführung der Anweisung sollten erst alle `ResultSet`s abgearbeitet werden, ehe mit `get<Typ>`-Methoden die `OUT`-Parameter geholt werden. Folgendes Code-Fragment soll das illustrieren, wobei der erste Paramter ein `INOUT`-Parameter, der zweite ein `OUT`-Parameter ist:

```
CallableStatement prozAnw = verbindung.prepareCall (
   "{call prozedurName (?, ?)}");
prozAnw.setInt (1, 25);
prozAnw.registerOutParameter (1, java.sql.Types.INTEGER);
prozAnw.registerOutParameter (2, java.sql.Types.DECIMAL, 3);
ResultSet   rs = prozAnw.executeQuery ();
// Verarbeite rs
int i = prozAnw.getInt (1);
java.math.BigDecimal d = prozAnw.getBigDecimal (2, 3);
```

31.3.5 SQL- und Java-Typen

Für eine professionelle Arbeit mit JDBC sollte man sich die Einzelheiten bezüglich der Entsprechung von SQL-Typen und Java-Typen in der JDBC-Referenz ansehen. Diese darzustellen führt hier zu weit. Es soll nur der Hinweis gegeben werden, dass da einiges zu beachten ist.

31.4 Beispiel `JDBC_Test`

```
import   java.lang.*;
import   java.io.*;
import   java.sql.*;
public class   JDBC_Test
   {
   public static void   main (String[] argument)
      {
      String dbURL = argument [0];
      try
         {
         Class.forName ("postgresql.Driver");
         }
         catch (ClassNotFoundException e)
            {
            System.err.println ("Fehler in JDBC_Test.main: " + e);
            }
```

```
    try
       {
       Connection  v = DriverManager.getConnection (dbURL, "kfg", "");
       try
          {
          Statement   s = v.createStatement ();
          try
             {
             s.executeUpdate ("DROP TABLE SUPPLIER");
             }
             catch (SQLException e)
                {
                System.err.println ("Fehler in JDBC_Test.main: ");
                System.err.println ("   DROP TABLE SUPPLIER: " + e);
                }
          s.executeUpdate ("CREATE TABLE SUPPLIER ("
             + "SNR CHAR (6) NOT NULL,"
             + "SNAME CHAR (12),"
             + "STATUS INTEGER,"
             + "CITY CHAR (12))");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S1', 'Smith', 20, 'London')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S2', 'Jones', 10, 'Paris')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S3', 'Blake', 30, 'Paris')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S4', 'Clark', 20, 'London')");
          s.executeUpdate ("INSERT INTO SUPPLIER (SNR, SNAME, STATUS, CITY)"
             + "VALUES ('S5', 'Adams', 30, 'Athens')");
          ResultSet   r =
             s.executeQuery (
                "SELECT SNAME FROM SUPPLIER WHERE CITY = 'Paris'");
          System.out.println ("SUPPLIER.SNAME");
          System.out.println ("--------------");
          while (r.next ())
             {
             System.out.println (r.getString ("SNAME"));
             }
          System.out.println ("--------------");
          s.close ();
          }
          catch (SQLException e)
             {
             System.err.println ("Fehler in JDBC_Test.main: " + e);
             }
       v.close ();
       }
       catch (SQLException e)
          {
          System.err.println ("Fehler in JDBC_Test.main: ");
          System.err.println ("   DriverManager.getConnection: " + e);
          }
    }
}
```

31.5 Transaktionen

Eine Transaktion besteht aus einem oder mehreren SQL-Anweisungen, die nacheinander vollständig ausgeführt werden und entweder mit einem Commit oder Rollback abgeschlossen werden.

Eine neu erzeugte Verbindung ist im Auto-Commit Mode, was bedeutet, dass nach jedem

`executeUpdate` automatisch die Methode `commit` aufgerufen wird. Jede Transaktion besteht aus einer SQL-Anweisung.

Wenn der Auto-Commit Mode abgeschaltet ist (`setAutoCommit (false)`), dann läuft eine Transaktion solange, bis explizit `commit ()` oder `rollback ()` für die Verbindung aufgerufen wird.

Wenn JDBC eine Exception wirft, dann wird das Rollback nicht automatisch gegeben. D.h. die ProgramiererIn muss in solchen Fällen ein Rollback ins `catch` aufnehmen.

Mit z.B.

```
verbindung.setTransactionIsolation (TRANSACTION_READ_UNCOMMITTED)
```

können verschiedene Isolations-Niveaus eingestellt werden.

Bemerkung: Anstatt dieser Transaktionsmechanismen empfiehlt es sich direkt die Mechanismen des verwendeten Datenbanksystems zu verwenden. Hier gibt es sehr starke Unterschiede.

31.6 JDBC 2.0

Diese neue JDBC-Version unterstützt erweiterte Möglichkeit `ResultSet`-Objekte zu behandeln. Dazu kommen BLOB-Felder und andere kleinere Verbesserungen. JDBC 2.0 unterstützt scrollbare und aktualisierbare Ergebnis-Mengen. Wegen weiterführender Literatur verweisen wir auf das Buch von Flanagan et al. [15].

31.7 Beispiel `SQLBefehle`

Die Applikation in der Klasse `SQLBefehle` führt alle in einer Datei angegebenen, mit Semikolon abgeschlossenen SQL-Befehle aus und schreibt eventuelle Resultate nach Standard-Ausgabe.

Aufruf: `$ java SQLBefehle <` *Dateiname*
Abbruch nach dem ersten fehlgeschlagenen SQL-Statement.

Beim Aufruf sind noch folgende Optionen angebbar:

```
-u userid
-p password
-b dburl
-d driver
-f filename
```

Beispiel:

```
$ java SQLBefehle -u mck -p mckpw -b jdbs:mysql://localhost/mck -f sqldatei
$ java SQLBefehle -u mck -p mckpw -b jdbs:mysql://localhost/mck < sqldatei
```

Bei diesem Beispiel ist die Verwendung der Methode `getMetaData ()` zu beachten, die ein Objekt der Klasse `ResultSetMetaData` liefert, das man über die Eigenschaften des `ResultSet`-Objekts ausfragen kann, um etwa das `ResultSet` geeignet darzustellen.

```
import   java.lang.*;
import   java.io.*;
import   java.sql.*;
import   java.util.*;
public class   SQLBefehle
   {
   public static void   main (String[] argument)
      // Option: -u userid z.B. "-u kfg"
      // Option: -p password z.B. "-p hochgeheim"
      // Option: -b DatenbankURL z.B. "-b jdbc:postgresql://spielberg/it95"
      // Option: -d Datenbanktreiber z.B. "-d postgresql.Driver"
      // Option: -f SQL-File z.B. "-f selAlles"
      {
      String   userID = "mck";
      String   password = "mckpw";
      String   dbURL = "jdbc:mysql://localhost/mck";
      String   dbDriver = "org.gjt.mm.mysql.Driver";
      String   sqlFile = null;
      for (int i = 0; i < argument.length; i+=2)
         {
         if (argument[i].equals ("-u")) userID = argument[i + 1];
         else if (argument[i].equals ("-p")) password = argument[i + 1];
         else if (argument[i].equals ("-b")) dbURL = argument[i + 1];
         else if (argument[i].equals ("-d")) dbDriver = argument[i + 1];
         else if (argument[i].equals ("-f")) sqlFile = argument[i + 1];
         }
      try
         {
         BufferedReader f = null;
         if (sqlFile != null)
               {
               f = new BufferedReader (new FileReader (sqlFile));
               }
         else
            {
            f = new BufferedReader (new InputStreamReader (System.in));
            }
         Class.forName (dbDriver);
         String   sw = "";
         try
            {
            String   s = null;
            int   j;
            while ( (s = f.readLine ()) != null)
               {
               if ((j = s.indexOf ("--")) >= 0)
                  {
                  if (j == 0) s = "";
                  else s = s.substring (0, j);
                  }
               sw = sw + s;
               }
            }
            catch (IOException e)
               {
               System.err.println ("Fehler bei f.readLine (): " + e);
               }
         StringTokenizer   st
            = new StringTokenizer (sw, ";", false);
         Connection  v = DriverManager.getConnection (dbURL, userID,
            password);
         Statement   s = v.createStatement ();
         try
            {
            while (st.hasMoreElements ())
               {
               String   sql = ( (String)st.nextElement ()).trim ();
               if (sql.equals ("")) continue;
```

```
System.out.println ();
System.out.println (sql + ';');
if (sql.toLowerCase ().indexOf ("select") < 0
   && sql.toLowerCase ().indexOf ("show") < 0
   && sql.toLowerCase ().indexOf ("describe") < 0)
   {
   s.executeUpdate (sql);
   }
else
   {
   ResultSet r = s.executeQuery (sql);
   ResultSetMetaData rm = r.getMetaData ();
   int spalten = rm.getColumnCount ();
   int[] w = new int[spalten + 1];
   for (int i = 1; i <= spalten; i++)
      {
      /*
      w[i] = rm.getColumnDisplaySize (i);
      if (w[i] < rm.getColumnLabel (i).length ())
         w[i] = rm.getColumnLabel (i).length ();
      */
      w[i] = rm.getColumnLabel (i).length ();
      }
   String   rs;
   while (r.next ())
      {
      for (int i = 1; i <= spalten; i++)
         {
         rs = r.getString (i);
         if (rs == null) rs = "NULL";
         if (w[i] < rs.length ()) w[i] = rs.length ();
         }
      }
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("--");
      for (int j = 0; j < w[i]; j++)
         System.out.print ('-');
      }
   System.out.println ("-");
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("| ");
      System.out.print (rm.getColumnLabel (i));
      for (int j = 0;
         j < w[i] - rm.getColumnLabel (i).length (); j++)
         System.out.print (' ');
      }
   System.out.println ("|");
   for (int i = 1; i <= spalten; i++)
      {
      System.out.print ("|-");
      for (int j = 0; j < w[i]; j++)
         System.out.print ('-');
      }
   System.out.println ("|");
   r = s.executeQuery (sql);
   while (r.next ())
      {
      for (int i = 1; i <= spalten; i++)
         {
         System.out.print ("| ");
         rs = r.getString (i);
         if (rs == null) rs = "NULL";
         for (int j = 0;
            j < w[i] - rs.length (); j++)
            System.out.print (' ');
         System.out.print (rs);
```

```
                    }
                 System.out.println ("|");
                 }
              for (int i = 1; i <= spalten; i++)
                 {
                 System.out.print ("--");
                 for (int j = 0; j < w[i]; j++)
                    System.out.print ('-');
                 }
              System.out.println ("-");
              }
           System.out.println ();
           }
        }
        catch (SQLException e)
           {
           System.err.println ("Fehler in JDBC_Test.main: " + e);
           e.printStackTrace ();
           }
     s.close ();
     v.close ();
     }
     catch (ClassNotFoundException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: " + e);
        }
     catch (FileNotFoundException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: " + e);
        }
     catch (SQLException e)
        {
        System.err.println ("Fehler in JDBC_Test.main: ");
        System.err.println ("   DriverManager.getConnection: " + e);
        }
  }
}
```

Anhang A

Hashing

Eine häufige Problemstellung ist die Suche von Objekten oder Elementen e über einen Schlüssel $k \in K$ ***(key, hash field)*** in einem "durchnummerierten" Speicherbereich (Tabelle, Datei). Bei den Hashverfahren wird versucht, aus dem Schlüssel direkt die Position $i \in I$ in einem Speicherbereich zu bestimmen. Auf diese Weise gelingt es, eine Suche mit annähernd konstantem Aufwand zu realisieren.

Schlüssel eignen sich normalerweise nicht direkt als Positionsangabe, weil es sich oft nicht um Zahlen handelt, sondern um Zeichenketten und weil die Menge der möglichen Schlüssel (K, ***hash field space)*** i.a. astronomisch groß ist, die Menge der möglichen Positionen aber (I, ***address space)*** begrenzt ist.

Daher wird der Schlüssel mit Hilfe einer **Hash-Funktion** ***(hash function, randomizing function)***

$$h : K \to I$$

in einen **Primär-Index** transformiert, der ein Intervall der ganzen Zahlen ist.

Wenn alle Schlüssel auf **verschiedene** Primär-Indizes abgebildet werden, dann spricht man von **perfektem Hashing**. Dazu muss aber die Menge der Schlüssel von vornherein bekannt sein. Außerdem darf es nicht mehr Schlüssel geben als in die Tabelle passen. Das sind Voraussetzungen, die in der Praxis selten erfüllt sind. Man muss damit rechnen, dass unterschiedliche Schlüssel auf **gleiche** Primär-Indizes abgebildet werden:

$$h(k_1) = h(k_2) \quad \text{bei} \quad k_1 \neq k_2$$

Das wird als **Kollision** ***(collision)*** bezeichnet.

Die Hash-Funktion ist so zu wählen, dass einerseits Kollisionen selten sind, andererseits die Berechnung effizient ist (nicht zu lange dauert). Z.B. ist die Gewichtung der einzelnen Zeichen eines Schlüssels mit Primzahlen hinsichtlich Kollisionsvermeidung sehr wirksam, aber sehr aufwendig. Typischerweise hat eine Hash-Funktion folgende Form:

$$h(k) = h_c(k) \bmod n$$

Dabei transformiert eine sogenannte **Hashcode**-Funktion h_c den Schlüssel in eine ganze Zahl, die durch mod n auf das Intervall $[0, n-1]$ abgebildet wird. Dabei wird der Schlüssel typischerweise "zerhackt". Die Tabellengröße n sollte eine Primzahl und etwa 30% größer als nötig sein, um die Anzahl der Kollisionen gering zu halten. Ziel einer guten Hash-Funktion ist die möglichst gleichmäßige Verteilung der Elemente im Speicherbereich.

Die Behandlung der Kollisionen ***(collision resolution)*** kann **extern (offenes Hashing)** oder **intern (geschlossenes Hashing)** erfolgen.

Externe Kollisions-Behandlung:

1. Jeder Tabelleneintrag zeigt auf einen **Hash-Eimer** ***(hash bucket)***, der mehrere Schlüssel/Element-Paare (k/e) aufnehmen kann. Die Eimer werden gegebenenfalls linear durchsucht.
2. Kollidierende (wenn ein Eimer voll ist) Schlüssel/Element-Paare werden in einen dafür reservierten Bereich geschrieben.

Multiples Hashing: Im Falle einer Kollision wird eine zweite oder dritte usw. Hash-Funktion versucht.

Interne Kollisions-Behandlung: Ein Tabelleneintrag besteht aus einem k/e-Paar. Im Fall einer Kollision wird vom Kollisionsort aus nach einem fest vorgegebenen Verfahren **(Sondieren)** der nächste freie Platz gesucht, wobei es zu sogenannten **Sekundärkollisionen** kommen kann. Es entstehen **Sondierketten**.

- Lineares Sondieren in Einerschritten: Einfach, aber die Wahrscheinlichkeit der Sekundärkollisionen ist sehr groß.
- Lineares Sondieren mit schlüsselabhängiger Schrittweite: Komplizierte Sondierfunktion.
- Quadratisches Sondieren: Keine Neigung zu Sekundärkollisionen, aber die Tabelle wird nur zu 50% ausgenutzt.

Eine i.a. aufwendige Reorganisation der Tabelle ***(rehashing)*** wird notwendig, wenn die Tabelle voll ist und erweitert werden soll.

Grundoperationen:

Suchen: Gegeben sei der Schlüssel k. Falls die Position $h(k)$ leer ist, wird Fehler angezeigt. Andernfalls wird k im Eimer bzw. in der Sondierkette gesucht.

Einfügen: Gegeben sei das Paar k/e. Falls die Position $h(k)$ leer ist, wird sie mit k/e gefüllt bzw. ein Eimer mit k/e angelegt. Andernfalls wird k/e im Eimer bzw. in der Sondierkette eingefügt.

Löschen: Beim Löschen von Einträgen muss man beachten, dass bei interner Kollisionsbehandlung Sondierketten unterbrochen werden. Das hat entweder Reorganisation oder die Verwaltung eines "Löschstatus" mit dann unnötig langen Sondierketten zur Folge.

Die Methode

```
public native int hashCode ()
```

der Java-Klasse `Object` ist so implementiert, dass die Adresse des Objekts geliefert wird. Das ist selten nützlich. Für die Klasse `String` ist die Methode allerdings sinnvoll überschrieben. Die Klasse

```
java.util.HashMap
```

repräsentiert eine erweiterbare Hash-Tabelle mit externer Kollisionsbehandlung. Im Konstruktor kann man eine Anfangsgröße und einen Füllungsgrad *(load factor)* angeben. Wenn der Füllungsgrad überschritten wird, wird die Tabellengröße verdoppelt mit folgender Reorganisation. Ein kleiner Füllungsgrad hat wenige Kollisionen, aber häufige Reorganisation zu Folge. Default ist eine Anfangsgröße von 101 und ein Füllungsgrad von 75%.

Algorithmen und ausführlichere Diskussionen zu diesem Thema sind bei Ludewig[29] und Wirth[41] zu finden.

A.1 Übungen

Gegeben sei folgende Hash-Funktion:

$h(k)$
$=$((Nummer des ersten Buchstabens von k im Alphabet) -1)mod11

Fügen Sie in folgende Tabelle die Schlüssel `Bernd`, `Martina`, `Monika`, `Peter`, `Helga`, `Heike`, `Barbara`, `Marcel`, `Sarah`, `Frieder` und `Torsten` nach den in den Spaltenköpfen angegebenen Verfahren ein.

Pos	Heap-Datei (FIFO)	Sortierte Datei	internes Hashing		externes Hashing	
			Lineares Sondieren	Quadratisches Sondieren	Nr.	Bucket
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
10						

Bemerkung: Es ist etwas übersichtlicher, wenn man zuerst das externe Hashing macht.

Überlegen Sie, wie Sie einen beliebigen Schlüssel wieder finden und was getan werden muss, wenn ein Element gelöscht wird, insbesondere, wenn die Anzahl der Daten groß wird.

Diskutieren Sie Vor- und Nachteile der verschiedenen Speichermethoden.

Anhang B

Baumstrukturen

Bäume eignen sich, geordnete, sich häufig ändernde Information aufzunehmen. Der Suchaufwand verhält sich logarithmisch.

Algorithmen und ausführlichere Diskussionen zu diesem Thema sind bei Ludewig[29] und Wirth[41] zu finden.

B.1 Definitionen

Ein **Baum** ***(tree)*** ist

- entweder eine leere Struktur (leerer Baum)
- oder die **Wurzel** ***(root)***, der d **(Grad, Ordnung,** ***degree)*** weitere Bäume **(Teil-** oder **Unterbäume,** ***subtree)*** zugeordnet sind.

Ein **Knoten** ***(node)*** ist entweder die Wurzel des Baums oder die Wurzel eines darin enthaltenen Unterbaums.

Ein **Blatt** ***(leaf)*** ist ein Knoten, dessen Unterbäume sämtlich leere Bäume sind.

Ein **innerer Knoten** ***(internal node)*** ist ein Knoten, der mindestens einen nicht leeren Unterbaum hat.

Die Wurzel ist das **Elter (Vater,** ***parent)*** oder der **direkte Vorgänger** aller d Wurzeln seiner Unterbäume. Die Wurzeln der Unterbäume sind die **Kinder (Söhne,** ***child)*** oder **direkten Nachkommen** der Wurzel.

Ein **geordneter** ***(ordered)*** Baum ist ein Baum, bei dem die Unterbäume aller Knoten eine geordnete Menge bilden.

Die **Höhe** ***(height)*** des Baums ist die Höhe des höchsten Unterbaums plus Eins. Ein leerer Baum hat die Höhe Null.

Die **Stufe** ***(level)*** eines Knotens ist die Stufe seines Elters plus Eins. Die Wurzel hat die Stufe Null.

Ein **Binärbaum (B-Baum, *binary tree, B-Tree)*** ist ein Baum des Grades 2 ($d = 2$).

Die Knoten eines Binärbaums können auf verschiedene Arten sequentiell durchlaufen **(traversiert, *traverse)*** werden:

- **Vorordnung *(preorder)***:
  1. Besuche Wurzel.
  2. Traversiere linken Unterbaum.
  3. Traversiere rechten Unterbaum.
- **Zwischenordnung *(inorder)***:
  1. Traversiere linken Unterbaum.
  2. Besuche Wurzel.
  3. Traversiere rechten Unterbaum.
- **Nachordnung *(postorder)***:
  1. Traversiere linken Unterbaum.
  2. Traversiere rechten Unterbaum.
  3. Besuche Wurzel.

Ein **binärer Suchbaum *(binary searchtree)*** ist ein Binärbaum, dessen Knoten unter der Zwischenordnung eine strenge Ordnung bilden.

Ein **vollständiger (perfekt, *perfect)*** Baum der Höhe h hat d^{h-1} Blätter. Alle anderen Knoten (d.h. außer den Blättern) haben d nichtleere Unterbäume. Insgesamt hat der vollständige Baum $\sum_{i=0}^{h-1} d^i$ Knoten.

Die **Weglänge *(depth)*** (Distanz zur Wurzel, Tiefe) eines Knotens ist gleich seiner Stufe. Die maximale Weglänge ist die Höhe minus Eins. Als **innere Weglänge *(internal distance)*** eines Baumes bezeichnet man die Summe der Weglängen aller Knoten. Die **mittlere Weglänge** ist gleich der inneren Weglänge geteilt durch die Zahl der Knoten.

B.2 Heap-Baum

Ein Heap-Baum oder kurz **Heap** ist ein Binärbaum, bei dem jedes Elter größer (kleiner) als seine Kinder ist, sofern es welche hat.

Dieser Baum spielt beim Heapsort eine Rolle. Wir gehen hier nicht näher darauf ein.

B.3 Binäre Suchbäume

Die einfachste Baumstruktur, mit der eine geordnete Menge gespeichert werden kann, ist der binäre Suchbaum oder einfach nur Binärbaum, wobei die Ordnung als selbstverständlich vorausgesetzt wird.

Grundoperationen:

Suchen: Das Suchen ist einfach und geht nach folgendem rekursiven Algorithmus: Der Suchschlüssel wird mit dem Schlüssel der Wurzel verglichen. Entweder passt er, dann ist man fertig, oder er passt nicht, dann wird entweder der linke oder der rechte Unterbaum durchsucht, jenachdem ob der Suchschlüssel kleiner oder größer als der Schlüssel der Wurzel ist. Wenn ein leerer Unterbaum angetroffen wird, dann wurde das Element nicht gefunden.

Einfügen: Auch das Einfügen ist einfach. Man sucht mit dem Schlüssel des einzufügenden Elements solange, bis man auf einen leeren Unterbaum trifft. Dort wird das Element dann als Blatt angehängt. Der Fehlerfall tritt ein, wenn der Schlüssel gefunden wird. Beim Einfügen sind strukturelle Änderungen des Baums nicht nötig.

Löschen: Das Löschen eines Blattes ist einfach, da die Struktur des Baums sich nicht ändert.

Das Löschen eines Knotens mit nur *einem* Unterbaum ist auch einfach. Der eine Unterbaum tritt einfach an die Stelle des zu löschenden Knotens.

Das Löschen eines Knotens K mit beiden nichtleeren Unterbäumen U_l und U_r ist komplizierter. Im Sinne der Zwischenordnung muss zunächst ein linker (oder rechter) Nachbar gefunden werden. Nehmen wir den linken Nachbarn N. Dieser hat die Eigenschaft, dass er der erste Knoten ohne rechten Unterbaum ist. Der zu löschende Knoten wird nun durch diesen linken Nachbarn N ersetzt. Dessen linker Unterbaum tritt an seine Stelle. (D.h. der Nachbar N wird durch seinen linken Unterbaum ersetzt.)

E_K sei das Elter von K, K sei die Wurzel des Unterbaums U_x von E_K und E_N sei das Elter von N. Dann kann man das formal so ausdrücken:

$$
\begin{array}{lcl}
\text{Hilf} & = & K \\
K & = & N \\
N & = & N.U_l \\
K.U_l & = & \text{Hilf}.U_l \\
K.U_r & = & \text{Hilf}.U_r
\end{array}
$$

B.4 Vollständig ausgeglichene Bäume

Das Ziel des logarithmischen Aufwands wird verfehlt, wenn die Bäume sehr ungleichmäßig wachsen. Das kann durch Randbedingungen verhindert werden.

Ein Binärbaum ist **vollständig ausgeglichen** ***(fully balanced)***, wenn an jedem Knoten die Differenz der Knotenzahlen der beiden Unterbäume höchstens Eins beträgt. Damit sind auch die Stufen der Blätter höchstens um Eins verschieden und der logarithmische Suchaufwand ist gesichert.

Allerdings ist beim Einfügen der Aufwand zur Reorganisation in ungünstigen Fällen sehr groß.

Daher kommen vollständig ausgeglichene Bäume nur in Frage, wenn selten oder nie eingefügt wird. Der Baum und die Unterbäume werden dann jeweils von der Mitte der geordneten Schlüssel aus aufgebaut. Man kann den Baum auch bezüglich der Zugriffshäufigkeiten optimieren.

B.5 Höhenbalancierte Bäume

Wenn Einfüge- und Löschoperationen eine Rolle spielen, dann kann man die Ausgleichsbedingung lockern. Die Bedingung von Adelson-Velskii und Landis **(AVL-Bäume)** lautet:

> *Die zwei Unterbäume eines Knotens dürfen sich in der Höhe höchstens um Eins unterscheiden.*

L_h sei die minimale Knotenzahl eines AVL-Baums der Höhe h. Dann gilt:

$$\begin{aligned} L_0 &= 0 \\ L_1 &= 1 \\ L_h &= 1 + L_{h-1} + L_{h-2} \quad \text{für} \quad h > 1 \end{aligned}$$

AVL-Bäume mit minimaler Knotenzahl heißen **Fibonacci-Bäume**. Die Zahlen L_i heißen **Leonardo-Zahlen**. Ihr Zusammenhang mit den Fibonacci-Zahlen ergibt sich zu $L_k = \sum_{i=1}^{k} F_i$.

Grundoperationen:

Suchen: Das Suchen funktioniert wie bei den binären Suchbäumen.

Einfügen: Das Einfügen erfordert eventuell Reorganisationsmaßnahmen. D.h durch den Zuwachs entsteht an einem Knoten ein Ungleichgewicht. Dieses Ungleichgewicht wird durch eine sogenannte **Rotation** beseitigt. Bei einem Rechts-Zuwachs ist das eine RR-Rotation, bei einem Links-Zuwachs eine LL-Rotation. Bei einer RR-Rotation wechseln Wurzel und rechter Unterknoten die Position. (D.h. der rechte Unterknoten wird die Wurzel und der Knoten wird der linke Unterknoten.) Bei einer LL-Rotation wechseln Wurzel und linker Unterknoten die Position.

Ein Rechts-Rechts-Zuwachs erfordert eine sogenannte RR-Rotation.

Entsprechend resultiert ein Links-Links-Zuwachs in einer LL-Rotation.

Diese Rotationen können wieder andere Ungleichgewichte zur Folge haben, die durch weitere Rotationen auszugleichen sind.

Ein Rechts-Links-Zuwachs erfordert eine sogenannte RL-Rotation. Das ist eine LL-Rotation am rechten Zuwachs gefolgt von einer RR-Rotation an der Wurzel.

Entsprechend resultiert ein Links-Rechts-Zuwachs in einer LR-Rotation.

Löschen: Das Löschen in einem AVL-Baum ist recht kompliziert. Zunächst wird der Knoten wie bei einem gewöhnlichen Suchbaum gelöscht. Ein anschließender Balanciervorgang hat eventuell das Schrumpfen von Teilbäumen zur Folge. Dies muss nach oben weitergemeldet werden mit der Folge, dass eventuell weitere Balanciervorgänge fällig werden. Trotzdem bleibt der Aufwand logarithmisch.

B.6 Übungen

B.6.1 Binärer Suchbaum

Fügen Sie nacheinander Elemente mit in folgender Tabelle angegebenen Schlüsseln in einen Binärbaum ein. Geben Sie in der Tabelle mit (+, –) an, ob die Eigenschaft "vollständig ausgeglichen" (VA) bzw "Adelson-Velskii-Landis" (AVL) nach der Einfügung erhalten bleibt.

	VA	AVL
7		
3		
4		
17		
2		
18		
22		
9		
11		

Und dasselbe mit einem weiteren Beispiel:

	VA	AVL
308		
52		
600		
987		
51		
4		
222		
412		
999		
577		
578		

Anhang C

MySQL

In diesem Anhang werden die wichtigsten Dinge über das Datenbanksystem MySQL zusammengefasst. Weitergehende Informationen findet man in dem Buch von Kofler [28].

Wir beschränken uns hier auf Angaben, die für das Linux-Betriebssystem gelten. Unter Windows sehen die Details etwas anders aus. Dazu verweisen wir wieder auf das Buch von Kofler [28].

Die wichtigste Quelle ist das Handbuch von MySQL, das unter

```
http://www.mysql.com/documentation/
```

zu finden ist. Die mitgelieferte Dokumentation gibt es irgendwo unter:

```
......../mysql/manual.html
```

C.1 Installation

C.1.1 SuSE-Linux

MySQL ist in fünf Pakete aufgeteilt, von denen üblicherweise die ersten drei mit `YaST` installiert werden:

`mysql.rpm`	(MySQL-Server)
`mysqclnt.rpm`	(MySQL-Clients)
`mysqllib.rpm`	(Shared Libraries)
`mysqbnch.rpm`	(Benchmark-Tests)
`mysqldev.rpm`	(C-Entwicklungsumgebung)

Nach der Installation kann MySQL manuell mit

```
root# /sbin/init.d/mysql start
```
oder
```
root# rcmysql start
```

gestartet werden. Damit MySQL automatisch beim Booten des Rechners startet, muss in `/etc/rc.config`

```
START_MYSQL="yes"
```

eingetragen werden. Als Datenbankverzeichnis wird `/var/mysql` verwendet.

Überprüfen Sie mit

```
$ps -axu | grep mysql
```

ob der MySQL-Server `mysqld` unter einem anderen User als `root` läuft. Er sollte **nicht** unter `root` laufen. Wenn das der Fall ist, dann hilft in der Datei `/etc/my.cnf` der Eintrag:

```
user = mysql
```

C.2 Sicherheit

MySQL überprüft beim Zugang zu einer Datenbank drei Informationen – Username, Passwort und Hostname:

- **Username**: Das ist der Username der Datenbank. Er hat grundsätzlich nichts zu tun mit dem Login-Namen des Betriebssystems. Natürlich steht es dem Benutzer frei, dieselben Namen zu verwenden. Aber es gibt keine Mechanismen zur Synchronisation dieser Namen.

  Der Name kann 16 Zeichen lang sein und ist case-sensitiv. Sonderzeichen sollten vermieden werden.

- **Passwort**: Hier gilt das gleiche wie für den Usernamen. MySQL speichert die Passwörter nur verschlüsselt. Anders als beim Username sollte man für Betriebssystem und Datenbank nie dasselbe Passwort verwenden.

- **Hostname**: Der Hostname ist der Name des Rechners, von dem aus eine Verbindung zur Datenbank geöffnet werden soll. Wenn es sich um den lokalen Rechner handelt, kann man als Hostnamen `localhost` angeben. Ansonsten ist der volle Internetname z.B. `bacall.informatik.ba-stuttgart.de` oder die IP-Nummer `141.31.11.82` notwendig. Beim Hostnamen sind als Platzhalter die Zeichen `_` (genau ein beliebiges Zeichen) und `%` (beliebig viele Zeichen) erlaubt. Auch die Hostnamen sind case-sensitiv.

Als Default-Werte werden für diese drei Größen verwendet:

Betriebssystem-Login-Name, "", "localhost"

Ein Verbindunsgaufbau zur Datenbank sieht dann folgendermaßen aus:

```
$ mysql -u Username@Hostname -p
```

Zur Eingabe des Passworts wird man dann aufgefordert.

Nach der Installation ist MySQL möglicherweise "offen wie ein Scheunentor". Im Abschnitt "Einrichten einer Datenbank" wird dies behandelt.

Testdatenbanken: Jeder Benutzer darf Datenbanken einrichten, deren Name mit `test_` beginnt. Diese Datenbanken sind vollkommen ungeschützt. Das kann man folgendermaßen abstellen:

```
root# mysql -u root -p
Enter password: xxx
mysql> USE mysql;
mysql> DELETE FROM user WHERE user='';
mysql> FLUSH PRIVILEGES;
mysql> exit
```

MySQL kennt folgende Privilegien:

Privileg	Erklärung
	für Tabellen und eventuell Spalten
SELECT	Erlaubt Daten zu lesen.
INSERT	Erlaubt Datensätze einzufügen.
UPDATE	Erlaubt Datensätze zu ändern.
DELETE	Erlaubt Datensätze zu löschen.
INDEX	Erlaubt Indexe anzulegen oder zu löschen.
ALTER	Erlaubt die Tabellenstruktur zu ändern.
	für Datenbanken, Tabellen, Indexe und eventuell Spalten
USAGE	Erlaubt nichts.
CREATE	Erlaubt Anlegen von Datenbanken bzw Tabellen.
DROP	Erlaubt Löschen von Datenbanken bzw Tabellen.
WITH GRANT OPTION	Erlaubt das Weitergeben von Privilegien.
REFERENCES	noch nicht in Verwendung
	für Dateizugriff
FILE	Erlaubt lokale Dateien zu lesen und zu schreiben.
	für MySQL-Administration
PROCESS	Erlaubt `processlist` und `kill`.
RELOAD	Erlaubt `reload, refresh, flush-....`
SHUTDOWN	Erlaubt MySQL herunterzufahren.

Das Zugriffssystem von MySQL wird in insgesamt sechs Tabellen der Datenbank `mysql` verwaltet. Die Attribute von fünf Tabellen – (Auf die Tabelle `func` wird später eingegangen.) – sind in folgendem Schema zusammenfassend dargestellt:

Tabelle: / Attribut ↓	user	db	host	tables_priv	columns_priv
Host	abc	abc	abc	abc	abc
User	abc	abc	—	abc	abc
Password	123	—	—	—	—
Db	—	abc	abc	abc	abc
Select_priv	Y/N	Y/N	Y/N	—	—
Insert_priv	Y/N	Y/N	Y/N	—	—
Update_priv	Y/N	Y/N	Y/N	—	—
Delete_priv	Y/N	Y/N	Y/N	—	—
Create_priv	Y/N	Y/N	Y/N	—	—
Drop_priv	Y/N	Y/N	Y/N	—	—
Grant_priv	Y/N	Y/N	Y/N	—	—
References_priv	Y/N	Y/N	Y/N	—	—
Index_priv	Y/N	Y/N	Y/N	—	—
Alter_priv	Y/N	Y/N	Y/N	—	—
Reload_priv	Y/N	—	—	—	—
Shutdown_priv	Y/N	—	—	—	—
Process_priv	Y/N	—	—	—	—
File_priv	Y/N	—	—	—	—
Table_name	—	—	—	abc	abc
Column_name	—	—	—	—	abc
Grantor	—	—	—	abc	—
Timestamp	—	—	—	456	456
Table_priv	—	—	—	pri1	—
Column_priv	—	—	—	pri2	pri2

— : Die Tabelle hat das Attribut nicht.
abc: Zeichenkette
123: Verschlüsseltes Passwort
456: Datum-Zeit-Stempel-Wert
Y/N: Wert kann `Y` oder `N` sein.
pri1: `Select, Insert, Update, Delete, Create, Drop, Grant, References, Index, Alter`
pri2: `Select, Insert, Update, References`

Bei `pri2` gibt es eine Redundanz. Aus der Dokumentation geht nicht hervor, welche Bedeutung dies hat.

In MySQL kann man die Granularität der Zugriffsprivilegien steuern von "global" bis spaltenspezifisch. D.h man kann Privilegien vergeben auf folgenden Ebenen:

MySQL-Ebene: Privileg gilt für *alle* Datenbanken und deren Tabellen und Spalten **(global)**.
SQL: `GRANT/REVOKE ... ON *.* ...`

Datenbank-Ebene: Privileg gilt für *eine* Datenbank und ihre Tabellen und Spalten.
SQL: `GRANT/REVOKE ... ON datenbankname.* ...`
SQL: `GRANT/REVOKE ... ON * ...`
(alle Tabellen der aktiven Datenbank)

Tabellen-Ebene: Privileg gilt für *eine* Tabelle und ihre Spalten.
SQL: `GRANT/REVOKE ... ON datenbankname.tabellenname ...`
SQL: `GRANT/REVOKE ... ON tabellenname ...`
(aktive Datenbank)

Spalten-Ebene: Privileg gilt für *angegebene* Spalten einer Tabelle.
SQL: `GRANT/REVOKE privilegx (spaltey, ...), ... ON tabellenname ...`

Diese vier Ebenen (von global bis spaltenspezifisch) sind insofern hierarchisch geordnet, als ein auf höherer Ebene gewährtes Privileg ("`Y`") auf einer tieferen Ebene nicht zurückgenommen werden kann. Allerdings kann eine höhere Ebene durch ein "`N`" nicht verhindern, dass eine tiefere Ebene ein Privileg gewährt.

Im folgenden werden diese Tabellen und ihre Bedeutung für die Privilegien kurz beschrieben, da sie wichtig für ein Verständnis des Zugriffskonzeptes sind. Eine direkte Manipulation dieser Tabellen mit SQL-Anweisungen ist äußerst fehleranfällig. Man sollte dazu die `GRANT`- und `REVOKE`-Anweisungen verwenden. (Die Beispiele sind für einen Usernamen `Oskar` mit Passwort `geheim` vom Host `Rechner`, die Datenbank `spielbank`, deren Tabelle `geld` und deren Spalte `betrag` ausgeführt.)

`user`: Steuert den Zugang zu MySQL und verwaltet die globalen Privilegien. Für `root` sind diese defaultmäßig auf "`Y`" gestellt. Mit

```
mysql> GRANT USAGE ON *.* TO Oskar@Rechner IDENTIFIED BY 'geheim';
```

wird ein Benutzer ohne Privilegien angelegt. Das ist meist vernünftig. Später kann er dann spezifische Privilegien bekommen. Mit

```
mysql> GRANT ALL ON *.* TO Oskar@Rechner WITH GRANT OPTION;
```

bekommt der Benutzer alle Rechte. Das ist normalerweise nicht sinnvoll. Wenn ein Passwort für einen Benutzer einmal eingetragen ist, dann muss es nicht mehr angegeben werden, es sei denn man will es ändern. (`ALL` umfasst nicht das Privileg `GRANT`.) Mit

```
mysql> REVOKE DELETE ON *.* FROM Oskar@Rechner;
```

oder

```
mysql> REVOKE GRANT OPTION ON *.* FROM Oskar@Rechner;
```

werden einem Benutzer wieder Rechte entzogen. Allerdings kann man mit `REVOKE` einen Benutzer nicht völlig entfernen. Dazu muss man dann ein direktes `DELETE` in der Tabelle `user` ausführen:

```
mysql> DELETE FROM user where User='Oskar';
```

`db`: Steuert den Zugang zu einer Datenbank und verwaltet die Privilegien für die einzelnen Datenbanken. Mit

```
mysql> GRANT ALL ON etoto.* TO Oskar WITH GRANT OPTION;
```

bekommt der Benutzer `Oskar` alle Rechte für die Datenbank `etoto`, die er dann auch weitergeben darf. `Oskar` kann von jedem Rechner aus zugreifen.

`host:` Ergänzt die Tabelle `db` bezüglich des Zugriffs von anderen Rechnern aus. Die Tabelle `host` kommt nur zum Einsatz, wenn das Feld `Host` in der Tabelle `db` **leer** ist. Dann wird in `host` gesucht, ob es ein passendes `Host/Db`-Paar gibt, und, falls es ein Paar gibt, werden die Privilegien aus `db` und `host` logsich mit AND verknüpft. Das Resultat ergibt dann die erlaubten Privilegien.

D.h.man kann für einen Benutzer Datenbank-Privilegien unabhängig vom Zugriffshost setzen und diese dann für einzelne Hosts (aber unabhängig vom Benutzer) einschränken.

Die Tabelle `host` wird selten verwendet und ist defaultmäßig leer. Sie wird auch nicht durch die Kommandos `GRANT` und `REVOKE` berührt. Mit

```
mysql> GRANT ALL ON etoto.* TO Oskar@amadeus;
mysql> UPDATE db SET Host='' WHERE User='Oskar' AND Host='amadeus';
mysql> INSERT INTO host (Host, Db, Select_priv)
     >     VALUES ('amadeus', 'etoto', 'Y');
```

bekommt der Benutzer `Oskar` alle Rechte für die Datenbank `etoto`. Aber er kann nur von `amadeus` aus zugreifen, und zwar nur lesend (`Select_priv`). Die anderen Privilegien wurden beim `INSERT` defaultmäßig auf `N` gesetzt.

`tables_priv:` Steuert den Zugriff auf einzelne Tabellen.

```
mysql> GRANT SELECT, UPDATE ON etoto.geld TO Oskar;
```

Die Tabelle `tables_priv` verwaltet die Rechte als Mengen (`SET`).

`columns_priv:` Steuert den Zugriff auf einzelne Spalten.

```
mysql> GRANT SELECT (betrag), UPDATE (betrag) ON etoto.geld TO Oskar;
```

Die Tabelle `cloumns_priv` verwaltet die Rechte ebenfalls als Mengen (`SET`).

`func:` Ermöglicht die bisher noch nicht dokumentierte Verwaltung von benutzerdefinierten Funktionen (**UDF,** ***user defined function***).

Die Zugriffsrechte eines Benutzers kann man betrachten mit

```
mysql> SHOW GRANTS FOR Oskar;
mysql> SHOW GRANTS FOR Oskar@amadeus;
```

Ausführlichere Möglichkeiten bietet:

```
$ mysqlaccess Oskar etoto
```
und `$ mysql_setpermission -u root`

C.2.1 Systemsicherheit

Eine Datenbank kann überhaupt nur sicher sein, wenn das zugrundeliegende Betriebssystem sicher ist oder entsprechende Sicherheits-Mechanismen zur Verfügung stellt. Folgende Problembereiche müssen hier untersucht werden:

- Kann sich jemand unberechtigterweise als `root` oder System-Administrator auf dem System einloggen?
- Sind die Logging-Dateien abgesichert? Dort finden sich Passwörter im Klartext.
- Sind Script-Dateien, in denen Passwörter im Klartext stehen, ausreichend abgesichert?
- Der MySQL-Server darf nicht unter `root` laufen, sondern unter einem speziellen Benutzer, also etwa `mysql`.
- Oft ist es nicht nötig, dass MySQL über den Port `3306` von außen ansprechbar ist. Dann kann man diesen Port mit Firewall-Mechanismen sperren.

C.3 Einrichten einer Datenbank

Öffnen Sie als `root` eine Verbindung zu MySQL und erzeugen Sie mit

```
mysql> CREATE DATABASE Datenbankname;
```

eine Datenbank. Es ist zweckmäßig für jede Datenbank einen eigenen Datenbankadministrator einzurichten:

```
mysql> GRANT ALL ON Datenbankname.* TO DatenbanknameAdmin
     >    IDENTIFIED BY 'Passwort' WITH GRANT OPTION;
```

Zum Beispiel:

```
mysql> CREATE DATABASE etoto;
mysql> GRANT ALL ON etoto.* TO etotoadmin
     >    IDENTIFIED BY 'geheim' WITH GRANT OPTION;
```

und möglicherweise:

```
mysql> GRANT ALL ON etoto.* TO etotoadmin@localhost
     >    IDENTIFIED BY 'geheim' WITH GRANT OPTION;
```

Offenbar muss `localhost` extra abgegeben werden. Das ist merkwürdig.

C.4 Anlegen von Benutzern

Wenn eine Datenbank erzeugt und ein Datenbankadministrator eingerichtet ist, dann kann man weitere Benutzer für diese Datenbank einrichten. Mit

```
mysql> GRANT USAGE ON etoto.* TO Oskar@Rechner IDENTIFIED BY 'geheim';
```

wird ein Benutzer `Oskar` für die Datenbank `etoto` eingerichtet, der nur vom Rechner `Rechner` aus zugreifen darf. Der Benutzer hat dann noch keinerlei Rechte. Deren Vergabe ist im Abschnitt "Sicherheit" beschrieben.

Für einen bestimmten Rechner möchten Sie für *jeden* Benutzer den Zugang zu MySQL (oder auch einer Datenbank und einigen Rechten) öffnen:

```
mysql> GRANT USAGE ON *.* TO ''@Rechner;
```

C.4.1 Ändern des Passworts

Wenn gleichzeitig Zugriffsrechte geändert werden, dann bietet sich

```
mysql> GRANT ... IDENTIFIED BY neuesPasswort
```

an. Sonst ist die einfachere Alternative:

```
$ mysqladmin -u Oskar -p password neuesPasswort
Enter password:  altesPasswort
```

Rootpasswort Vergessen

```
root# /sbin/init.d/mysql stop
root# startproc /usr/bin/safe_mysqld --user=mysql \
          --datadir=/var/mysql --skip-grant-tables
root# mysql -u root
mysql> USE mysql;
mysql> UPDATE user SET password=''
     > WHERE user='root' AND host='localhost';
     > quit
root# /sbin/init.d/mysql restart
```

Und nun das Passwort neu definieren.

Unter Windows:
Erst muss MySQL gestoppt werden:

```
   Start > Systemsteuerung > Verwaltung > Computerverwaltung
      > Dienste und Anwendungen > Dienste > MySQL > beenden
```

oder über den Taskmanager stoppen.

```
$ mysqld --skip-grant-tables
$ mysql -u root
> use mysql
> update user set password = PASSWORD('neu') where user = 'root';
> exit
```
MySQL muss wieder gestoppt werden. Dann als Dienst durch Reboot starten oder direkt:
```
$ mysqld
```

C.5 Backup

C.5.1 Backup einer Tabelle

Mit

```
mysql> BACKUP TABLE Tabelle TO 'Verzeichnis';
```

wird eine Tabelle als zwei Dateien (`.MYD` bzw `.db` und `.frm`) in das angegebene Verzeichnis geschrieben. Von dort kann man die Tabelle wieder holen. Mit

```
mysql> RESTORE TABLE Tabelle FROM 'Verzeichnis';
```

kann man die Tabelle wiederherstellen. (Dazu darf die Tabelle aber nicht existieren.)

Beispiele:

```
mysql> BACKUP TABLE S TO '/tmp/supaBackups';
mysql> DROP TABLE S;
mysql> RESTORE TABLE S FROM '/tmp/supaBackups';
```

C.5.2 Backup einer Datenbank

Eine ASCII-Backupdatei einer ganzen Datenbank kann mit dem Programm `mysqldump` erstellt werden.

```
$ mysqldump -u DBloginname -p --opt Datenbankname > Backupdatei
```

Das Gegenstück dazu erfordert die Verwendung des SQL-Monitors `mysql`:

```
$ mysql -u DBloginname -p Datenbankname < Backupdatei
```
oder
```
mysql> CREATE DATABASE Datenbankname;
mysql> USE Datenbankname;
mysql> SOURCE Backupdatei;
```

Bemerkung: Unter Windows wird bei `mysqldump` nicht nach dem Passwort gefragt. Man muss also das Passwort unaufgefordert eintippen! Ferner darf das `>` bzw `<` nicht durch Blanks eingerahmt werden.

```
$ mysqldump -u  DBloginname -p --opt  Datenbankname>Backupdatei
```

C.6 Datentypen

MySQL stellt folgende Datentypen zur Verfügung:

Datentyp	Erklärung
`TINYINT`	8-Bit-Integer
`SMALLINT`	16-Bit-Integer
`MEDIUMINT`	24-Bit-Integer
`INT, INTEGER`	32-Bit-Integer
`BIGINT`	64-Bit-Integer
`FLOAT`	8-stellige Fließkommazahl
`DOUBLE, REAL`	16-stellige Fließkommazahl
`DECIMAL(p,s)` `NUMERIC, DEC`	Fließkommazahl als Zeichenkette
`DATE`	Datum in der Form `2002-07-27`
`TIME`	Zeit in der Form `14:31:45`
`DATETIME`	Kombination in der Form `2002-07-27 14:31:45`
`YEAR`	Jahreszahl 1900 bis 2155
`TIMESTAMP`	Kombination in der Form `20020727143145`
`CHAR(n)`	Zeichenkette mit vorgegebener Länge, maximal n = 255
`VARCHAR(n)`	Zeichenkette mit variabler Länge n < 256
`TINYTEXT`	wie `VARCHAR(255)`
`TEXT`	Zeichenkette mit variabler Länge, maximal $2^{16} - 1$
`MEDIUMTEXT`	Zeichenkette mit variabler Länge, maximal $2^{24} - 1$
`LONGTEXT`	Zeichenkette mit variabler Länge, maximal $2^{32} - 1$
`TINYBLOB`	Binärdaten mit variabler Länge, maximal $2^{8} - 1$
`BLOB`	Binärdaten mit variabler Länge, maximal $2^{16} - 1$
`MEDIUMBLOB`	Binärdaten mit variabler Länge, maximal $2^{24} - 1$
`LONGBLOB`	Binärdaten mit variabler Länge, maximal $2^{32} - 1$
`ENUM`	Aufzählung von maximal 65535 Zeichenketten (1 oder 2 Byte)
`SET`	Menge von maximal 63 Zeichenketten (1 bis 8 Byte)

C.7 Kurzreferenz

Hier werden die wichtigsten MySQL-Kommandos beschrieben (oder nur genannt, damit man sie nachschlagen kann):

`mysql> CREATE DATABASE` **Datenbankname;**
Erzeugt eine neue Datenbank.

`mysql> DROP DATABASE` **Datenbankname;**
Löscht eine ganze Datenbank.

`mysql> USE` **Datenbankname;**
Ändert die aktive Datenbank.

`mysql> DESCRIBE` **Tabellenname;**
Zeigt die Struktur einer Tabelle.

`mysql> SHOW ...;`
Zeigt alles mögliche.

- `mysql> SHOW COLUMNS FROM` Tabelle `LIKE [FROM` Datenbank`] [`Muster`]`;
  Liefert Informationen über Spalten, deren Name zum Muster passt.
- `mysql> SHOW CREATE TABLE` Tabelle;
  Zeigt das SQL-Kommando, mit dem die Tabelle erzeugt wurde.
- `mysql> SHOW DATABASES [LIKE` Muster`]`;
  Liefet eine Liste aller Datenbanken, die einem eventuell angegebenen Muster entsprechen.
- `mysql> SHOW GRANTS FOR` User`@`Host;
  Zeigt die Zugriffsrechte eines Benutzers.
- `mysql> SHOW INDEX FROM` Tabelle;
  Zeigt die Indexe einer Tabelle.
- `mysql> SHOW TABLE STATUS`;
  Liefert Status-Informationen über alle Tabellen einer Datenbank.
- `mysql> SHOW TABLES [FROM` Datenbank`] [LIKE` Muster`]`;
  Liefert eine Liste der Tabellen einer Datenbank.

`mysql> \.` **Dateiname** (Ohne Semikolon!)
Datei mit SQL-Statements wird eingelesen und ausgeführt.

C.8 Absichern von MySQL

In diesem Abschnitt wird zusammenfassend schrittweise gezeigt, wie eine MySQL-Installation abgesichert werden kann.

Wir gehen davon aus, dass MySQL frisch installiert worden ist und noch keine Sicherheitseinstellungen vorgenommen wurden.

1. Damit man wieder von vorn beginnen kann, ist es ratsam die Datenbank `mysql` vorher zu sichern:

   ```
   mysqldump -u root --opt mysql > mysql.bck
   ```

   (Wiederherstellung erfolgt gegebenenfalls mit:

   ```
   mysql -u root mysql < mysql.bck
   ```
   oder `mysql -u root -p mysql < mysql.bck`

   falls für `root` schon ein Passwort vergeben wurde.)

   Wenn etwas mit den Berechtigungen des Superusers schief geht, dann kann man oft auch das Laden des Backup-Files nicht durchführen.

   Daher legen wir noch einen Benutzer `spezial` an, der alles darf:

   ```
   mysql> USE mysql;
   mysql> GRANT ALL ON *.* TO spezial@localhost
                           IDENTIFIED BY 'pwspezial' WITH GRANT OPTION;
   ```

   Mit diesem Benutzer können wir dann direkt die Tabellen manipulieren. Man sollte diesen Benutzer dann auch "ausprobieren".

2. Der MySQL-Server `mysqld` darf nicht unter dem Betriebssystem-Administrator laufen. Unter Windows läuft er automatisch unter dem Benutzer `ODBC`. Unter Linux muss man aber aufpassen, dass er nicht ausversehen unter `root` gestartet wird.

3. Es muss einen Super-Administrator geben, dem alles – allerdings nur vom Datenbankserver (`localhost`) aus – erlaubt ist. Er darf zur Datenbank nur einen Verbindungsaufbau mit Passwort machen. Der Datenbank-Super-Administrator soll auch `root` heißen und das Passwort `psswrd` haben.

   ```
   $ mysql -u root
   mysql> GRANT ALL ON *.* TO root@localhost IDENTIFIED BY 'psswrd'
               WITH GRANT OPTION;
   mysql> FLUSH PRIVILEGES;
   ```

   Jetzt sollte überprüft werden, ob das funktioniert hat:

   ```
   mysql> SELECT Host, User, Password, Select_priv FROM user;
   ```

   In der Zeile `localhost`, `root` sollte es jetzt einen Passwort-Eintrag geben.

   Wenn hier noch andere Einträge für `root` erscheinen, dann müssen diese entfernt werden. Z.B. sollte ja `%` für `root` nicht eingetragen sein. Also löschen wir das heraus:

   ```
   mysql> DELETE FROM user WHERE User = 'root' AND Host = '%';
   ```

   Entsprechend sollten andere `root`-Einträge außer für `localhost` entfernt werden.

   Nach den Änderungen müssen wir ein

   ```
   mysql> FLUSH PRIVILEGES;
   ```

   geben, damit der MySQL-Server die Sicherheitstabellen wieder neu einliest.

4. Jetzt gibt es immer noch namenlose Benutzer, die sich auf der Datenbank einloggen können. Diese streichen wir ganz heraus:

```
mysql> DELETE FROM user WHERE User = '';
mysql> FLUSH PRIVILEGES;
```

   Von nun an sind administrative Arbeiten nur noch als `root` von `localhost` aus mit Passwort möglich (abgesehen von unserem Benutzer `spezial`, den wir noch in der Hinterhand behalten).

5. Wenn WinMySQLadmin gestartet wurde, dann gibt es einen WinMySQLadmin-Benutzer. Sein Name sei `wab`. Wir sollten ihnen bestehen lassen, aber mit eingeschränkten Rechten:

```
mysql> REVOKE INSERT, UPDATE, DELETE, DROP, FILE, ALTER
          ON *.* FROM wab@localhost;
mysql> REVOKE GRANT OPTION ON *.* FROM wab@localhost;
mysql> FLUSH PRIVILEGES;
```

6. Nun legen wir beispielhaft eine Anwendungs-Datenbank an mit Namen `masp`:

```
mysql> CREATE DATABASE masp;
```

   Jede Datenbank sollte einen eigenen Administrator haben, der sich nur lokal und mit Passwort einloggen kann. Diese Datenbank bekommt den Administrator `maspadmin` mit Passwort `masppw`:

```
mysql> GRANT ALL ON masp.* TO maspadmin@localhost
             IDENTIFIED BY 'masppw' WITH GRANT OPTION;
mysql> FLUSH PRIVILEGES;
```

   Das überprüfen wir wieder mit:

```
mysql> SELECT Host, User, Password, Select_priv FROM user;
mysql> SELECT Host, User, Db FROM db;
```

   Damit wäre die Arbeit von `root` abgeschlossen. Allerdings benötigen wir `root` immer wieder, wenn ein neuer Benutzer angelegt werden soll. Das geschieht im folgenden Schritt.

7. Anlegen eines Benutzers, der nur auf die Datenbank `masp` mit Passwort zugreifen kann, dafür aber von jedem Rechner aus. Weitere Rechte wird der Administrator von `masp` vergeben.

```
mysql> GRANT USAGE ON masp.* TO maspus
             IDENTIFIED BY 'maspuspw';
mysql> FLUSH PRIVILEGES;
```

8. Die weiteren Rechte vergibt der Administrator von `masp`. Dazu müssen wir uns als solcher einloggen:

```
$ mysql -u maspadmin -p masp
mysql> GRANT SELECT, UPDATE, INSERT, DELETE ON masp.* TO maspus;
```

9. Der Benutzer `maspus` kann sich nun von allen Rechnern aus auf dem Rechner einloggen, wo MySQL mit der Datenbank `masp` läuft, also etwa der Rechner `dbserver`:

```
$ mysql -u maspus -h dbserver -p masp
```

Anhang D

Objektorientiertes Datenbanksystem db4o

Das objektorientierte Datenbanksystem db4o gibt es unter `.NET` und `Java`. Der folgende Anhang enthält eine kurze Einführung in die Verwendung der Datenbank db4o unter Java JDK 5.

Als Beispiel werden Taxen und ihre Chauffeure in der Datenbank verwaltet. Alle Beispiele werden über die Klasse `Anwendung` gestartet.

D.1 Installation

Die db4o-Datenbank-Maschine besteht aus einer einzigen `.jar`-Datei.

Wenn man von db4o die Java-zip-Datei für die Datenbank heruntergeladen und entpackt hat, findet man im Verzeichnis `lib` eine Datei mit etwa folgendem Namen:

```
db4o-8.0.184.15484-all-java5.jar
```

Es genügt, diese Datei in den CLASSPATH für `javac` und `java` aufzunehmen. Unter Eclipse wird die Datei in das `/lib/`-Verzeichnis Ihres Projekts kopiert und als Bibliothek zum Projekt addiert.

Damit ist das Datenbanksystem installiert. Die Größe der `jar`-Datei beträgt nur etwa 2,7 MByte!

D.2 Öffnen, Schließen

Unter db4o ist eine Datenbank eine Datei an beliebiger Stelle mit beliebigem Namen. Es ist allerdings Konvention, dass als Datei-Erweiterung `.db4o` verwendet wird.

Als `import`-Anweisung genügt zunächst:

```
import   com.db4o.*;
```
und/oder
```
import   com.db4o.query.*
```

Unsere Beispieldatenbank soll `taxi.db4o` heißen. Sie wird neu angelegt oder geöffnet, indem

```
com.db4o.ObjectContainer   db
   = com.db4o.Db4oEmbedded.openFile (
      Db4oEmbedded.newConfiguration (),
      "taxi.db4o");
```

aufgerufen wird.

Nach Verwendung der Datenbank, muss sie wieder geschlossen werden:

```
db.close ();
```

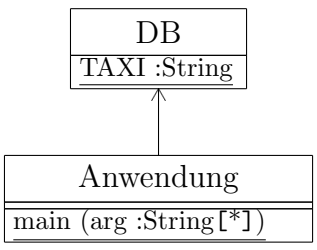
Um das Schließen der Datenbank zu garantieren, werden wir einen Konstrukt

```
try { ... } finally { db.close (); }
```

verwenden.

Bemerkungen:

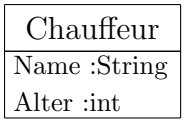
1. Ein `ObjectContainer` repräsentiert im Single-User-Mode die Datenbank, im Multi-User-Mode eine Client-Verbindung zu einem db4o-Server für eine Datenbank. Wir werden zunächst im Single-User-Mode arbeiten.
2. Jeder `ObjectContainer` besitzt eine Transaktion. Jeder Datenbank-Vorgang ist transaktional. Wenn ein `ObjectContainer` geöffnet wird, beginnt sofort eine Transaktion. Mit `commit ()` oder `rollback ()` wird die nächste Transaktion gestartet. Die letzte Transaktion wird durch `close ()` beendet.
3. Jeder `ObjectContainer` verwaltet seine eigenen Referenzen zu gespeicherten Objekten.
4. `com.db4o.ObjectContainer` hat sehr wenige Methoden. Jeder `ObjectContainer` kann nach `com.db4o.ext.ExtObjectContainer` gecasted werden. Dort sind wesentlich mehr Methoden definiert.



Verzeichnis: `../code/oeffnen/`

D.3 Speichern von Objekten

Wir schreiben eine Klasse `Chauffeur`, deren Konstruktor den Namen und das Alter des Chauffeurs erwartet:



Folgender Code legt einen Chauffeur neu an und speichert ihn in der Datenbank:

```
Chauffeur   chauffeur = new Chauffeur ("Ballack", 29);
db.store (chauffeur);
```

Verzeichnis: `../code/speichern/`

Bemerkung: Primitive Datenelemente (`boolean`, `byte`, `char`, `int`, `long`, `float`, `double`) und auch `String`- und `Date`-Objekte können nicht als eigene Objekte in der Datenbank gespeichert werden.

D.4 Finden von Objekten

Es gibt drei Möglichkeiten Objekte durch Anfragen an die Datenbank zu ermitteln:

- QBE: Query by Example, Anfrage durch Angabe eines Beispielobjekts
- NQ: Native Query
- SODA: Query API Simple Object Database Access

Es wird empfohlen NQ zu verwenden. Aber in diesem Abschnitt behandeln wir zunächst QBE.

Wir erzeugen ein prototypisches Objekt, z.B.:

```
Chauffeur   chauffeur = new Chauffeur (null, 29);
```

Damit können wir uns alle Chauffeure im Alter von 29 Jahren aus der Datenbank holen:

```
List<Chauffeur>   resultat = db.queryByExample (chauffeur);
```

Die resultierenden Chauffeure zeigen wir uns an mit:

```
for (Object o : resultat)
   {
   System.out.println (o);
   }
```

Da wir diesen Code öfters benötigen, schreiben wir ihn in die Methode

```
static void zeigeResultat (List<?> aList)
```

der Klasse `DB`.

Wenn wir alle Objekte der Klasse `Chauffeur` benötigen, dann schreiben wir

```
List<Chauffeur>   resultat = db.query (Chauffeur.class);
```

Leider funktioniert auch

```
List<Chauffeur>   resultat = db.queryByExample (new Chauffeur (null, 0));
```

genauso, weil "0" der Defaultwert eines `int` ist.

Wenn wir den Namen angeben, erhalten wir alle Chauffeure, die diesen Namen tragen:

```
List<Chauffeur>   resultat = db.queryByExample (new Chauffeur ("Kahn", 0));
```

Wir können auch nach Objekten von Klassen suchen, die eine bestimmte Schnittstelle realisieren oder eine abstrakte Klasse erweitern.

Bemerkungen:

1. Wenn nichts gefunden wird, dann kann das daran liegen, dass man nach unterschiedlichen `Chauffeur`-Klassen sucht! Die `Chauffeur`-Klassen könnten in unterschiedlichen Packages liegen (bei Eclipse)!

2. Die verschiedenen Query-Methoden liefern als Resultat immer ein `ObjectSet`. Es wird aber empfohlen `List` zu verwenden, das von `ObjectSet` erweitert wird.

Verzeichnis: `../code/finden/`

D.5 Aktualisierung von Objekten

Ballack wird ein Jahr älter:

```
List<Chauffeur>   resultat = db.queryByExample (new Chauffeur ("Ballack", 0));
Chauffeur   gefunden = resultat.get (0);
gefunden.setAlter (gefunden.getAlter () + 1);
db.store (gefunden); // Aktualisierung!
```

Dabei ist zu beachten, dass wir ein zu aktualisierendes Objekt erst aus der Datenbank holen müssen, bevor wir es aktualisieren können. Wenn man nicht aufpasst, setzt man ausversehen ein neues Objekt in die Welt!

Verzeichnis: `../code/aktualisieren/`

D.6 Loeschen von Objekten

Objekte werden aus der Datenbank mit der Methode `delete` entfernt:

```
List<Chauffeur>   resultat
   = db.queryByExample (new Chauffeur ("Ballack", 0));
Chauffeur   gefunden = resultat.get (0);
db.delete (gefunden);
```

Mit dem Code

```
List<Object>   resultat
   = db.queryByExample (new Object ());
for (Object o : resultat))
   {
   db.delete (o);
   }
```

loeschen wir alle Objekte in der Datenbanken.

Verzeichnis: `../code/loeschen/`

D.7 NQ – Native Anfragen

QBE-Anfragen sind sehr eingeschränkt verwendbar, insbesondere wenn nach Defaultwerten gesucht werden soll. NQ-Anfragen dagegen bieten weitergehende Möglichkeiten unter Beibehaltung des Vorteils, dass sie in der Hostsprache formuliert werden können. Damit werden diese Anfragen typsicher, Compilezeit-prüfbar und refactorierbar.

Syntax: Der `ObjectContainer`-Methode `query (...)` wird ein Objekt einer – normalerweise anonymen – Klasse übergeben, die die Schnittstelle `Predicate<T>` realisiert, indem die Methode `match (...)` implementiert wird. Das sei an folgendem Beispiel verdeutlicht:

```
List<Chauffeur>   chauffeure = db.query (new Predicate<Chauffeur> ()
   {
   public boolean match (Chauffeur chauffeur)
      {
      return chauffeur.getAlter () > 30;
      }
   });
```

Die Anfrage wird gegen alle Objekte vom Typ `T`, hier im Beispiel also `Chauffeur`, durchgeführt. Die Methode `match` kann irgendwelchen Code enthalten, solange sie ein `boolean` zurückgibt.

Bemerkung: In Eclipse kann man sich für diesen Anfragetyp ein Template erstellen:
`Window -> Preferences -> Java -> Editor -> Templates -> New`
(Rechts muss `Java` als Kontext eingestellt sein.)
Als Name könnte man `nq` verwenden und dann folgenden Text in das Musterfeld eingeben:

```
List<${extent}> list = db.query (new Predicate<${extent}> ()
   {
   public boolean match (${extent} candidate)
      {
      return true;
      }
   });
```

Der folgende Code zeigt einige Beispiele:

Verzeichnis: `../code/nqQueries/`

D.8 SODA Query API

SODA ***(simple object database access)*** ist die low-level Anfrage-API. Mit SODA können Anfragen dynamisch generiert werden.

Mit der `ObjectContainer`-Methode `query ()` wird ein `Query`-Objekt erzeugt, für das Einschränkungen definiert werden können.

```
// Finde alle Chauffeure jünger als 30 Jahre:
Query query = db.query ();
query.constrain (Chauffeur.class);
query.descend ("alter")
   .constrain (new Integer (30))
```

```
      .smaller ();
List  resultat = query.execute ();
```

Ein sogenannter Query-Graph besteht aus Query-Knoten und Einschränkungen. Ein Query-Knoten ist ein Platzhalter für ein mögliches Ergebnis-Objekt. Eine Einschränkung entscheidet, ob ein Objekt tatsächlich ins Ergebnis aufgenommen wird.

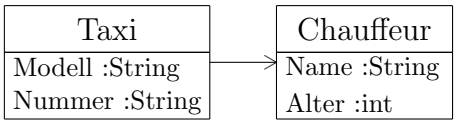
Die Syntax von SODA wird hier nicht präsentiert. Stattdessen sollen einige Beispiele grob zeigen, wie es funktioniert. Wenn man tiefer einsteigen will, muss man die (leider spärliche) Dokumentation von db4o oder das Internet zu Rate ziehen. Ein *gute* ausführliche Darstellung gibt es offenbar noch nicht.

Der folgende Code zeigt einige Beispiele:

Verzeichnis: `../code/sodaQueries/`

D.9 Strukturierte Objekte

Jetzt geben wir dem Chauffeur ein Taxi



und legen ein paar Taxen und Chauffeure an:

```
Chauffeur   chauffeur = new Chauffeur ("Ballack", 31);
db.store (chauffeur);
   // Chauffeur ist jetzt schon in der Datenbank.
Taxi  taxi = new Taxi ("BMW", "13", chauffeur);
db.store (taxi);

chauffeur = new Chauffeur ("Kahn", 39);
taxi = new Taxi ("VW", "1", chauffeur);
db.store (taxi);
      // Chauffeur ist hiermit auch in der Datenbank.

db.store (new Taxi ("Mercedes", "32", new Chauffeur ("Gomez", 22)));
```

Wenn ein Objekt auf andere Objekte verweist, werden diese mit dem Objekt gespeichert, falls sie noch nicht in der Datenbank sind.

Anfragen: Wir wollen alle Taxen haben, die Gomez fährt.

QBE:

```
Chauffeur   prototypChauffeur = new Chauffeur ("Gomez", 0);
Taxi  prototypTaxi = new Taxi (null, null, prototypChauffeur);
List<Taxi>  resultat = db.queryByExample (prototypTaxi);
```

NQ:

```
List<Taxi>  resultat = db.query (
   new Predicate<Taxi> ()
      {
      public boolean match (Taxi taxi)
         {
         return taxi.getChauffeur ().getName ().equals ("Gomez");
         }
      });
```

SODA:

```
Query query = db.query ();
query.constrain (Taxi.class);
query.descend ("aChauffeur").descend ("name").constrain ("Gomez");
List  resultat = query.execute ();
```

Eine interessante SODA-Anfrage ist: "Alle Chauffeure, die ein BMW-Taxi fahren." (Interessant, weil die Chauffeure ihr Taxi nicht kennen.)

```
Query taxiQuery   = db.query ();
taxiQuery.constrain (Taxi.class);
taxiQuery.descend ("modell").constrain ("BMW");
Query chauffeurQuery = taxiQuery.descend ("aChauffeur");
List  resultat = chauffeurQuery.execute ();
```

Aktualisierung:

1. Ein Taxi bekommt einen anderen Chauffeur.
   ```
   List<Taxi>  resultat = db.queryByExample (new Taxi ("BMW", null, null));
   Taxi  taxi = resultat.get (0);
   List<Chauffeur>   res = db.queryByExample (new Chauffeur ("Gomez", 0));
   Chauffeur   chauffeur = res.get (0);
   taxi.setChauffeur (chauffeur);
   db.store (taxi);  // Taxi wird zurückgespeichert.
   ```

2. Der Name des Chauffeurs eines Taxis wird geändert.
```
List<Taxi>  resultat = db.queryByExample (new Taxi ("BMW", null, null));
Taxi  taxi = resultat.get (0);
taxi.getChauffeur ().setName ("Strauss-Kahn");
db.store (taxi);  // Taxi wird zurückgespeichert.
```
Wenn man das mit einer zweiten Datenbank-Verbindung überprüft, dann sieht man, dass der Name nicht geändert ist. Hier stoßen wir auf das – i.a. große –

   Problem der Aktualisierungstiefe
   (update depth).

   Bei db4o ist die Aktualisierungstiefe defaultmäßig eins, d.h. die Operation `db.store (...)` aktualisiert nur primitive Datenelemente. In diesem Fall hätten wir den Chauffeur gesondert speichern müssen.

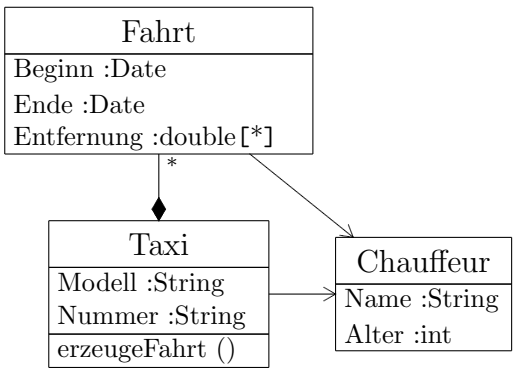
   Ähnlich verhält sich das System beim Löschen von Objekten, das defaultmäßig nur "flach" ausgeführt wird.

   Man kann die Datenbank-Verbindung bezüglich Update- und Delete-Verhalten konfigurieren. Darauf wird in einem späteren Abschnitt eingegangen. Aber insgesamt ist das ein Problem, für das es bei vielen DBMS schließlich keine wirklich befriedigende allgemeine Lösung gibt.

Verzeichnis: `../code/taxi/`

D.10 Felder und Collections

Jedes Taxi verwaltet alle seine Fahrten (Klasse `Fahrt`). Eine Fahrt beginnt (`Beginn`), endet (`Ende`) und hat zwei Entfernungen (Hin- und Rückfahrt), die – um ein Beispiel für ein Feld zu haben – in einem Feld von `double` gespeichert werden.



- Felder sind keine Objekte der Datenbank.
- Collections sind eigene Objekte der Datenbank.

Anfrage QBE: Alle Fahrten mit Klose:

```
Chauffeur   prototypChauffeur = new Chauffeur ("Klose", 0);
Fahrt prototypFahrt = new Fahrt (null, null, null, prototypChauffeur);
List<Fahrt> resultat = db.queryByExample (prototypFahrt);
```

Anfrage NQ: Alle Fahrten mit Geschwindigkeit kleiner 40 oder größer 80 km/h:

```
List<Fahrt> resultat = db.query (new Predicate<Fahrt> ()
   {
   public boolean match (Fahrt fahrt)
      {
      double   v = fahrt.getEntfernung ()[0]
         / ((fahrt.getEnde ().getTime () - fahrt.getBeginn ().getTime ())
            /1000/60/60);
      return v < 40.0 || v > 80.0;
      }
   });
```

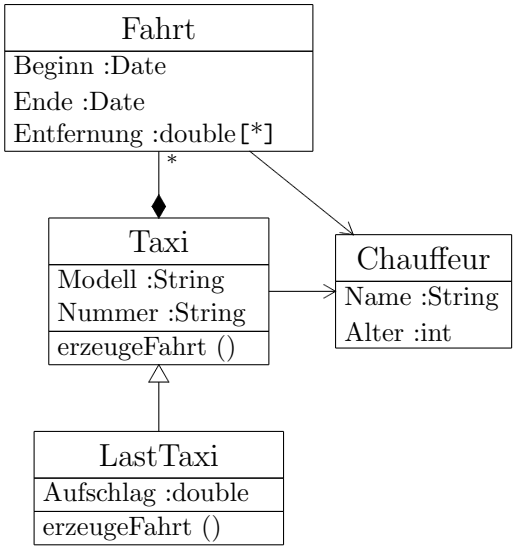
Anfrage SODA: Alle Fahrten mit Klose:

```
Query fahrtQuery = db.query ();
fahrtQuery.constrain (Fahrt.class);
fahrtQuery.descend ("aChauffeur")
   .descend ("name")
      .constrain ("Klose");
List  resultat = fahrtQuery.execute ();
```

Verzeichnis: `../code/fctaxi/`

D.11 Vererbung

Neben den normalen Taxen gibt es auch Lasttaxen (Klasse `LastTaxi`), die sich wie Taxen verhalten, allerdings fahren sie etwas langsamer und haben noch einen Preisaufschlagsfaktor (Attribut `Aufschlag`):



Die Vererbung funktioniert, wie erwartet. Es ist nur noch zu ergänzen, dass wir mit

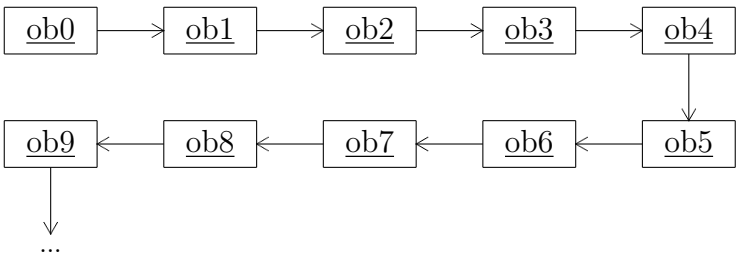
```
db.query (X.class)
public boolean match (X x) { }
```

auch nach Objekten von Klassen suchen können, die eine bestimmte Schnittstelle `X` realisieren oder eine Klasse `X` erweitern.

Verzeichnis: `../code/lasttaxi/`

D.12 Tiefe Graphen und Aktivierung

Betrachten wir folgende Objektstruktur mit einem tiefen Graphen, d.h. die Struktur ist sehr tief und weit verzeigert.

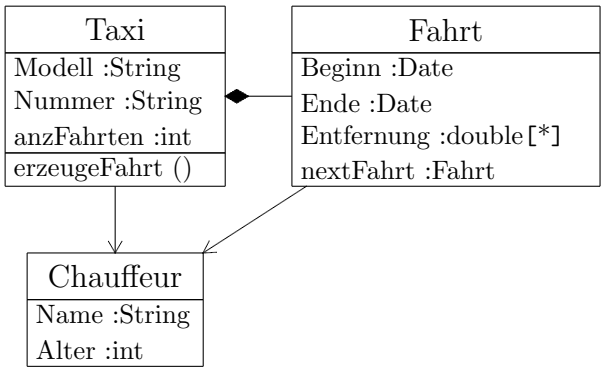


Wenn wir Objekt `ob0` von der Datenbank holen, sollen dann alle anderen über den Graphen an ihm hängenden Objekte auch mitgeholt werden? Nein, denn man würde damit ja eventuell die ganze "Welt" laden, was eventuell sehr lange dauern und unseren Speicherplatz sprengen würde.

D.12.1 Defaultverhalten

Bei db4o heißt der Term anstatt "Laden" "Aktivieren". In der Tat werden defaultmäßig nur Objekte bis zur Tiefe 5 aktiviert, also `ob0` bis `ob4`. Der Zeiger von `ob4` auf `ob5` ist zwar noch ein gültiger Zeiger, aber das `ob5` wird mit Defaultwerten initialsiert (Standard- und Referenztypen), d.h. alle Zeiger von `ob5` aus bleiben `null`.

Das wollen wir demonstrieren, indem wir die Fahrten eines Taxis nicht in einem `ArrayList`-Objekt, sondern in einer verketteten Liste verwalten, die wir selbst schreiben. Die Klasse `Fahrt` bekommt einfach einen Zeiger `nextFahrt` auf die nächste Fahrt. Die Klasse `Taxi` kennt direkt nur die erste Fahrt.



Der folgende Code legt einen Chauffeur und ein Taxi mit 9 Fahrten an. Wenn wir nach Neueröffnung der Datenbank das Taxi wieder von der Datenbank holen, dann wird zwar noch die fünfte Fahrt aktiviert, nicht mehr aber deren Referenz auf ihren Chauffeur, obwohl der Chauffeur schon längst wieder aktiv ist. Auch bleiben die Entfernung und die `Date`-Objekte `null`.

Verzeichnis: `../code/tiefgraph/`

D.12.2 Dynamische Aktivierung

Wir können Objekte **dynamisch** aktivieren durch:

```
db.activate (objekt, tiefe);
```

Dazu muss allerdings `db` zur Verfügung stehen.

Im folgenden Code wurde das in der `toString ()`-Methode der Klasse `Taxi` mit

```
Anwendung.getDb ().activate (fahrt, 1);
```

gemacht:

Verzeichnis: `../code/tiefgraph2/`

D.12.3 Kaskadierte Aktivierung

Man kann für einen Typ die Aktivierung **kaskadieren:**

```
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.common ().objectClass (Fahrt.class).cascadeOnActivate (true);
db = Db4oEmbedded.openFile (conf, "taxi.db4o");
```

(Damit wird eine geladene Fahrt auch sofort aktiviert, was das Laden der nächsten Fahrt zur Folge hat.)

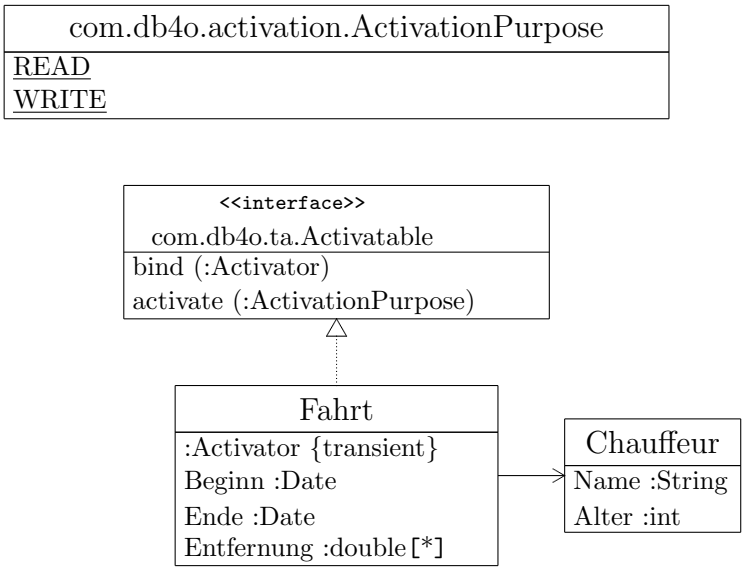
Siehe dazu: Verzeichnis: `../code/tiefgraph3/`

D.12.4 Transparente Aktivierung

Wir können allerdings auch dafür sorgen, dass alle Objekte immer aktiviert werden mit:

```
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.common ().add (new TransparentActivationSupport ());
db = Db4oEmbedded.openFile (conf, "taxi.db4o");
```

Dabei riskieren wir aber wieder, dass eventuell die ganze Datenbank geladen wird. Objekte sollten eigentlich nur dann aktiviert werden, wenn wir sie brauchen. Genau das ist jetzt (bei transparenter Aktivierung) möglich für Objekte von Klassen, die die Schnittstelle `Activatable` realisieren.



Die `get`-Methoden werden – nach Bedarf – folgendermaßen implementiert:

```
public Typ  getX ()
   {
   activate (ActivationPurpose.READ);
   return x;
   }
```

Im folgenden Beispiel machen wir die `Klasse` Fahrt aktivierbar.

Verzeichnis: `../code/tiefgraph4/`

Es sei darauf hingewiesen, dass db4o noch wesentlich mehr Möglichkeiten bezüglich Aktivierung anbietet. Dennoch: Mit der Aktivierung von Objekten muss man vorsichtig sein. Offenbar kann man leicht Fehler machen, die sich meistens entweder in `NullPointerException`s oder in der Performanz äußern oder den Speicher voll laufen lassen.

D.12.5 Transparente Persistenz

Bei den strukturierten Objekten hatten wir schon über das Problem der Aktualisierungstiefe ***update depth*** gesprochen. Wenn sie zu klein ist, vergessen wir eventuell aktualisierte Objekte zurückzuspeichern. Wenn sie zu groß ist, kann es zu viele unnötige Aktualisierungen geben.

Mit ***transparent persistence*** kann das Problem gelöst werden. Der Mechanismus ist ganz ähnlich der transparenten Aktivierung. Die Datenbank wird entsprechend konfiguriert:

```
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.common ().add (new TransparentPersistenceSupport ());
db = Db4oEmbedded.openFile (conf, "taxi.db4o");
```

Ab jetzt ist die Aktualisierungstiefe für jede Klasse unendlich. Das kann jetzt aber wieder für jede Klasse individuell gesteuert werden, indem die Schnittstelle `Activatable` realisiert wird und in den `set`-Methoden

```
activate (ActivationPurpose.WRITE)
```

aufgerufen wird.

Verzeichnis: `../code/tiefgraph5/`

D.13 Indexe

Bemerkung: Indexe wirken sich insbesondere bei QBE- und SODA-Anfragen aus, inzwischen aber auch bei NQ-Anfragen, allerdings nicht besonders drastisch. Das ist auch verständlich, da man dem Code von `match (...)` i.a. nur schwer ansehen kann, welcher Index zu verwenden wäre.

Bei geschlossener Datenbank kann mit den Anweisungen

```
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.common ().objectClass (Chauffeur.class)
   .objectField ("name").indexed (true);
db = Db4oEmbedded.openFile (conf, "taxi.db4o");
```

ein Index über dem Attribut `name` der Klasse `Chauffeur` erzeugt werden, der beim ersten Gebrauch der Klasse Chauffeur erzeugt wird.

Wenn der Index erzeugt ist, dann genügt zur Öffnung der Datenbank:

```
db = Db4oEmbedded.openFile (Db4oEmbedded.newConfiguration (), "taxi.db4o");
```

Mit

```
db.close ();
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.common ().objectClass (Chauffeur.class)
   .objectField ("name").indexed (false);
db = Db4oEmbedded.openFile (conf, "taxi.db4o");
```

wird der Index wieder gelöscht.

Verzeichnis: `../code/index/`

Indexe können auch auf Referenzen angelegt werden. Bei der Suche muss man allerdings darauf achten, dass man mit Objekten aus der Datenbank sucht! Im Beispiel Verzeichnis: `../code/indexRef/` wird nach Chauffeuren mit einer gewissen Adresse gesucht. Die Adresse muss daher erst einmal aus der Datenbank geholt werden.

Wenn man eine neues Adress-Objekt nimmt, dann wird es – bei SODA und QBE – offenbar als Prototyp verwendet, um die Adresse in der Datenbank zu finden, die dann für die Suche verwendet wird. Also man findet schließlich auch die gewünschten referenzierenden Objekte. Allerdings wirkt sich der Index nicht aus.

Bei NQ wird tatsächlich mit der Hauptspeicher-Adresse des neuen Adress-Objekts verglichen, die es in der Datenbank nicht gibt. Also findet man hier keine Chauffeure.

Bemerkung: Das nachträgliche Anlegen eines Index ist insgesamt schneller, kostet aber mehr, eventuell zu viel Speicher.

D.14 Transaktionen

Schon die ganze Zeit hatten wir implizit mit einer Transaktion gearbeitet, die mit dem Öffnen (`openFile`) eines `ObjectContainer`s beginnt und mit dessen Schließen (`close`) mit einem Commit endet.

Mit

```
db.commit ();
```

beenden wir eine Transaktion vor dem Schließen des Objektcontainers. Die nächste Transaktion beginnt mit dem nächsten Zugriff auf den Objektcontainer.

Mit

```
db.rollback ();
```

kann die laufende Transaktion rückgängig gemacht werden. Dabei wird zwar der Datenbankinhalt rückgängig gemacht, nicht aber Objekte, die für die Transaktion aus der Datenbank geholt wurden. Diese sogenannten **Live**-Objekte müssen wir selbst wieder aus der Datenbank holen und zwar mit:

```
db.ext ().refresh (objekt, Integer.MAX_VALUE);
```

Ein erneutes `db.query... (...)` nützt hier nichts, da hierbei offenbar nur das Live-Objekt geholt wird.

Bemerkung: `ext`-Funktionalität läuft zwar stabil, könnte aber noch geändert werden.

Das Isolations-Niveau einer Transaktion ist `READ COMMITTED`.

Wie stehts mit Locks? Es gibt offenbar alle Möglichkeiten (über Semaphore und Callbacks).

Verzeichnis: `../code/transaction/`

D.15 Client/Server

In diesem Abschnitt behandeln wir nun Transaktionen, die parallel laufen. Das kann innerhalb einer JVM oder über mehrere JVMs verteilt stattfinden. Konzeptionell gibt es da keinen Unterschied. "`ObjectContainer`" und "Datenbankverbindung" sind unter db4o synonym. Eine Transaktion bezieht sich immer auf eine(n) `ObjectContainer`/Datenbankverbindung.

Bei nur einer JVM wird ein Datenbank-Server unter Port 0 geöffnet. Das bedeutet, dass kein Netz verwendet wird. Diesen Fall – Embedded Server – behandeln wir zunächst.

D.15.1 Embedded Server

Bisher hatten wir einen `ObjectContainer` direkt durch Öffnen der Datenbankdatei erhalten. Wenn wir mehrere Container (d.h. Transaktionen) gleichzeitig benötigen, dann müssen wir über einen Server gehen. Folgender Code zeigt die Vorgehensweise:

```
import   com.db4o.cs.*;

// ...

ObjectServer   server = Db4oClientServer.openServer (
   Db4oClientServer.newServerConfiguration (), "taxi.db4o", 0);   // Port 0
try
```

```
{
ObjectContainer   ct1 = server.openClient ();
ObjectContainer   ct2 = server.openClient ();
ObjectContainer   ct3 = server.openClient ();
// tu was mit den Containern bzw. Clients
ct1.close ();
ct2.close ();
ct3.close ();
}
finally
   {
   server.close ();
   }
```

Jeder Client-Container hat seinen eigenen Cache von schwachen Referenzen auf die ihm schon bekannten Objekte. Um alle ***committed*** Änderungen durch andere Transaktionen sichtbar zu machen, muss eine Transaktion die ihr bekannten Objekte explizit mit der Datenbank abgleichen ***(refresh)***:

```
container.ext ().refresh (objekt, tiefe);
```

Die `tiefe` gibt an, wie weit bei einem Objekt-Graphen gegangen werden soll.

Im folgenden Programm legt die Transaktion `ct1` einen neuen Chauffeur an und gibt diesen einem existierenden Taxi. Das aktualisierte Taxi wird wieder zurückgespeichert. Ein zweite Transaktion `ct2` "schaut dabei zu". Was diese zweite Transaktion vor und nach einem Commit bzw. Refresh sieht, soll das oben Ausgeführte exemplarisch verdeutlichen.

Verzeichnis: `../code/embeddedServer/`

D.15.2 Verteiltes System

Um das System über ein TCP/IP-Netz zu verteilen, wird der Server mit einer Port-Nummer größer Null geöffnet. Die prinzipielle Vorgehensweise zeigt folgender Code:

```
ObjectServer   server = Db4oClientServer.openServer (
   Db4oClientServer.newServerConfiguration (), "taxi.db4o", 0xdb40); // Port > 0
try
   {
   server.grantAccess ("user1", "password1");
   server.grantAccess ("user2", "password2");

   ObjectContainer   ct1
      = Db4oClientServer.openClient ("localhost", 0xdb40, "user1", "password1");

   ObjectContainer   ct2
      = Db4oClientServer.openClient ("localhost", 0xdb40, "user2", "password2");
```

```
        // tu was mit den Containern bzw. Clients

        ct1.close ();
        ct2.close ();
        }
        finally
           {
           server.close ();
           }
```

Verzeichnis: `../code/clientServer/`

Server und Clients laufen normalerweise auf unterschiedlichen JVMs. Wir präsentieren ohne weiteren Kommentar den Code für die verschiedenen Aktionen, die dann auf unterschiedlichen Hosts durchgeführt werden können, wobei "localhost" entsprechend zu ändern ist:

Starten des Servers:
Verzeichnis: `../code/startServer/`

```
public class   Anwendung
   {
   // Constructors and Operations:
   public static void   main (String[] aString)
      {
      new StartServer (false).runServer ();
      }
   }  // end Anwendung
```

```
import   com.db4o.*;
import   com.db4o.cs.*;
import   com.db4o.messaging.*;
public class   StartServer
   implements  MessageRecipient
   {
   private boolean   stop;
   // Constructors and Operations:
   public final boolean isStop ()
      {
      return stop;
      }
   private final void   setStop (boolean stop)
      {
      this.stop = stop;
      }
   public   StartServer (boolean stop)
      /**
      *  working constructor
      **/
      {
      setStop (stop);
      }
   public synchronized void   runServer ()
      {
      ObjectServer   server = Db4oClientServer.openServer (
         Db4oClientServer.newServerConfiguration (), DB.TAXI, DB.PORT);
```

```
      try
         {
         server.grantAccess ("stopUser", "stopPassword");
         server.grantAccess ("user1", "password1");
         server.grantAccess ("user2", "password2");
         server.ext ().configure ().clientServer ().setMessageRecipient (this);
            // this.processMessage erhält die Botschaften (insbesondere stop)
         Thread.currentThread ().setName (this.getClass ().getName ());
            // um den Thread in einem Debugger zu identifizieren
         Thread.currentThread ().setPriority (Thread.MIN_PRIORITY);
            // server hat eigenen Thread. Daher genügt hier ganz niedrige Prio.
         try
            {
            System.out.println ("db4o-Server wurde gestartet.");
            while (!isStop ())
               {
               this.wait ();
                  // Warte, bis du gestoppt wirst.
               }
            }
            catch (InterruptedException e) { e.printStackTrace (); }
         }
         finally
            {
            server.close ();
            System.out.println ("db4o-Server wurde gestoppt.");
            }
      }
   public void processMessage
      (
      MessageContext aMessageContext,
      Object   message
      )
      {
      if (message instanceof StopServer)
         {
         close ();
         }
      }
   public synchronized void   close ()
      {
      setStop (true);
      this.notify ();
      }
   }  // end StartServer
```

```
public class   StopServer
   {
   // Constructors and Operations:
   public   StopServer ()
      /**
      *  working constructor
      **/
      {
      }
   }  // end StopServer
```

```
import   com.db4o.*;
import   java.util.*;
public class   DB
   {
```

```
   public static final String TAXI = System.getProperty ("user.home") + "/tmp/taxi.db4o";
   public static final String HOST = "localhost";
   public static final int PORT = 0xdb40;
   // Constructors and Operations:
   public static void   zeigeResultat (List<?> aList)
      {
      System.out.println ("Anzahl Objekte: " + aList.size ());
      for (Object o : aList)
         {
         System.out.println ("   " + o);
         }
      }
   }  // end DB
```

Stoppen des Servers:

Verzeichnis: `../code/stopServer/`

```
import   com.db4o.*;
import   com.db4o.cs.*;
import   com.db4o.messaging.*;
public class   Anwendung
   {
   // Constructors and Operations:
   public static void   main (String[] aString)
      {
      ObjectContainer   cont = null;
      try
         {
         cont = Db4oClientServer.openClient (
            DB.HOST, DB.PORT, "stopUser", "stopPassword");
         }
         catch (Exception e ) { e.printStackTrace (); }
      if (cont != null)
         {
         MessageSender  sender = cont.ext ().configure ().clientServer ()
            .getMessageSender ();
         sender.send (new StopServer ());
         //cont.close ();
         while (!cont.ext ().isClosed ());
         }
      }
   }  // end Anwendung
```

Füllen der Datenbank:

Verzeichnis: `../code/fillClient/`

```
import   com.db4o.*;
import   com.db4o.query.*;
import   java.util.*;
public class   Anwendung
   {
   private static ObjectContainer   db;
   // Constructors and Operations:
   public static final ObjectContainer getDb ()
      {
      return db;
      }
   private static final void  setDb (ObjectContainer db)
      {
```

```
      Anwendung.db = db;
      }
   public static void   erzeugeChauffeureUndTaxen ()
      {
      Chauffeur   chauffeur = new Chauffeur ("Ballack", 31);
      getDb ().store (chauffeur);   // Chauffeur ist jetzt schon in der Datenbank.
      Taxi  taxi = new Taxi ("BMW", "13", chauffeur);
      getDb ().store (taxi);

      chauffeur = new Chauffeur ("Kahn", 39);
      taxi = new Taxi ("VW", "1", chauffeur);
      getDb ().store (taxi);  // Chauffeur ist hiermit auch in der Datenbank.

      getDb ().store (new Taxi ("Mercedes", "32", new Chauffeur ("Gomez", 22)));

      System.out.println ("Alle Taxen in der Datenbank:");
      List<Taxi>  resultat = getDb ().query (Taxi.class);
      DB.zeigeResultat (resultat);

      System.out.println ();
      System.out.println ("und alle Objekte in der Datenbank:");
      resultat = getDb ().queryByExample (new Object ());
      DB.zeigeResultat (resultat);
      }
   public static void   alleObjekteLoeschen ()
      {
      List<Object>   resultat
         = getDb ().queryByExample (new Object ());
      for (Object o : resultat)
         {
         getDb ().delete (o);
         }
      resultat = getDb ().queryByExample (new Object ());
      DB.zeigeResultat (resultat);
      }
   public static void   main (String[] arg)
      {
      try
         {
         setDb (Db4o.openClient (DB.HOST, DB.PORT, "user1", "password1"));
         try
            {
            alleObjekteLoeschen ();
            erzeugeChauffeureUndTaxen ();
            }
            finally
               {
               getDb ().close ();
               }
         }
         catch (Exception e) { e.printStackTrace (); }
      }
   }  // end Anwendung
```

Zwei Clients:

Verzeichnis: `../code/zweiClients/`

```
import   com.db4o.*;
import   com.db4o.cs.*;
import   com.db4o.query.*;
import   java.util.*;
public class   Anwendung
   {
   // Constructors and Operations:
   public static void   main (String[] arg)
      {
      ObjectContainer   ct1 = null;
```

```
    ObjectContainer   ct2 = null;
    try
       {
       ct1 = Db4oClientServer.openClient (
          DB.HOST, DB.PORT, "user1", "password1");
       ct2 = Db4oClientServer.openClient (
          DB.HOST, DB.PORT, "user2", "password2");
       // Zunächst ein paar Daten in die Datenbank:
       ct1.store (new Taxi ("BMW", "13", new Chauffeur ("Ballack", 31)));
       ct1.commit ();
       System.out.println ();
       System.out.println ("Wir legen einen neuen Chauffeur Lahm an ");
       System.out.println ("   und geben ihn dem Taxi, das Ballack fährt,");
       System.out.println ("   unter Verwendung des Clients ct1.");
       Chauffeur   lahm = new Chauffeur ("Lahm", 27);
       List<Taxi> rs = ct1.queryByExample (new Taxi ("BMW", "13",
          new Chauffeur ("Ballack", 31)));
       Taxi  taxi = rs.get (0);
       System.out.println ("Taxi: " + taxi);
       System.out.println ();
       taxi.setChauffeur (lahm);
       ct1.store (taxi);
       System.out.println ("Datenbank-Inhalt von ct1 aus gesehen:");
       List<Object> rso = ct1.queryByExample (new Object ());
       DB.zeigeResultat (rso);
       System.out.println ("Datenbank-Inhalt von ct2 aus gesehen:");
       rso = ct2.queryByExample (new Object ());
       DB.zeigeResultat (rso);
       System.out.println ();
       if (true)   // Umschalten zwischen commit und rollback
          {
          System.out.println ("ct1 macht commit!");
          ct1.commit ();
          }
       else
          {
          System.out.println ("ct1 macht rollback!");
          ct1.rollback ();
          }
       System.out.println ();
       System.out.println ("Datenbank-Inhalt von ct1 aus gesehen:");
       rso = ct1.queryByExample (new Object ());
       DB.zeigeResultat (rso);
       System.out.println ("Datenbank-Inhalt von ct2 aus gesehen:");
       rso = ct2.queryByExample (new Object ());
       DB.zeigeResultat (rso);
       System.out.println ();
       System.out.println ("Datenbank-Inhalt von ct1 aus gesehen"
          + " nach Refresh:");
       rso = ct1.queryByExample (new Object ());
       DB.zeigeRefreshedResultat (ct1, rso, 2);
       System.out.println ("Datenbank-Inhalt von ct2 aus gesehen"
          + " nach Refresh:");
       rso = ct2.queryByExample (new Object ());
       DB.zeigeRefreshedResultat (ct2, rso, 2);
       }
       catch (Exception e) { e.printStackTrace (); }
       finally
          {
          ct1.close ();
          ct2.close ();
          }
    }
 }  // end Anwendung
```

D.15.3 Spezialitäten

Ohne auf Einzelheiten einzugehen nennen wir hier Probleme und auch Möglichkeiten, die man sich gegebenenfalls mit Hilfe der db4o-Dokumentation erarbeiten sollte.

- Native Queries als anonyme innere Klassen
- Out-of-band signalling: Wird eingesetzt, um dem Server besondere Botschaften zu senden (z.B. Defragmentierung. Als ein Beispiel haben wir das Stoppen des Servers gezeigt.).

D.16 Identifikatoren

db4o verwaltet Objekt-Identifikatoren (ID) transparent. IDs sollten von der AnwendungsprogrammiererIn nie verwendet werden. Daher ist dieser Abschnitt eigentlich überflüssig. Allerdings können IDs für die Fehlersuche hilfreich sein, wenn z.B. nicht klar ist, wie oft ein und dasselbe Objekt gespeichert wurde.

IDs spielen eventuell eine Rolle in zustandslosen Anwendungen, wenn Objekt und Datenbank getrennt wurden.

Unter db4o gibt es zwei ID-Systme.

D.16.1 Interne IDs

Die interne ID eines Objekts `ob` erhält man mit:

```
long  id = objectContainer.ext ().getID (ob);
```

Über die ID ist der schnellste Zugriff auf ein Objekt möglich:

```
Object   ob = objectContainer.ext ().getByID (id);
```

Vor Verwendung muss das Objekt allerdings noch aktiviert werden:

```
objectContainer.ext ().activate (ob, tiefe);
```

Die interne ID ist nur eindeutig bezüglich eines `ObjectContainer`s.

D.16.2 Eindeutige universelle ID (UUID)

Die UUID ***(unique universal ID)*** ist die eigentliche, unveränderliche, über Container eindeutige ID. Sie wird allerdings von db4o nicht automatisch generiert, da sie Platz und Performanz kostet.

Man kann die UUIDs global oder für einzelne Klassen generieren lassen (oder auch nicht).

```
Configuration  conf = Db4o.newConfiguration ();
conf.generateUUIDs (ConfigScope.DISABLED);
oder
conf.generateUUIDs (ConfigScope.GLOBALLY);
oder
conf.generateUUIDs (ConfigScope.INDIVIDUALLY);
conf.objectClass (Taxi.class).generateUUIDs (true);
```

Die UUID eines Objekts `ob` erhält man mit:

```
Db4oUUID uuid = objectContainer.ext ().getObjectInfo (ob).getUUID ();
```

Über die UUID ist ein Zugriff auf ein Objekt möglich:

```
Object   ob = objectContainer.ext ().getByUUID (uuid);
```

D.17 Probleme

In diesem Abschnitt nennen oder diskutieren wir Probleme, die uns bei der Arbeit mit db4o aufgefallen sind.

D.17.1 Viele Objekte, Lazy Query

Wenn man sehr viele Objekte, d.h. unter Windows etwa 1 000 000 Chauffeure, unter FreeBSD etwa 10 000 000 Chauffeure in der Datenbank speichert und dann versucht, einige dieser Objekte durch eine Anfrage wieder zu finden, dann kommt es zum Absturz, da offenbar der Java Heap Space überschritten wird. Dieses Problem kann man lösen, indem man die Anfrage als ***lazy query*** durchführt:

```
conf.common ().queries ().evaluationMode (QueryEvaluationMode.LAZY);
```

Verzeichnis: `../code/lazy/`

Im Lazy Querying Mode werden Objekte nicht ausgewertet. Stattdessen wird ein Iterator gegen den besten Index erzeugt. Damit erhält man die ersten Anfrageresultate beinahe sofort, und es wird kaum Speicher verbraucht. Die gefundenen Objekte werden erst dann in den ObjectContainer geladen, wenn sie wirklich benötigt werden. Bei nebenläufigen Transaktionen bedeutet das, dass committed Änderungen an der Datenbank durch andere Transaktionen (oder uncomitted durch die eigene) während der Anfrage Einfluss auf die ermittelten Objekte haben.

D.17.2 Defragmentierung

Auch wenn Objekte gelöscht werden, wächst die Datenbank-Datei trotzdem immer weiter. Daher muss gelegentlich – bei geschlossener Datenbank – defragmentiert werden mit:

```
com.db4o.defragment.Defragment.defrag (Datenbankdateiname);
```

Die alte Version wird nach Datenbankdateiname.backup kopiert.

Verzeichnis: `../code/defragment/`

Es gibt zahlreiche Optionen für die Defragmentierung.

D.17.3 Maximale Datenbankgröße

Defaultmäßig ist die maximale Datenbankgröße 2GB. Das entspricht einer "Blockgröße" ***(block size)*** von 1 byte. Die Blockgröße kann eingestellt werden zwischen 1 byte und 127 byte, was einer maximalen Datenbankgröße von 254GB entspricht.

Diese Einstellung ist möglich bei der Neuanlage einer Datenbank oder beim Defragmentieren mit:

```
EmbeddedConfiguration   conf = Db4oEmbedded.newConfiguration ();
conf.file ().blockSize (8);
```

Anhang E

Übung ODB

Mit der Übung ODB sollen erlernt werden:

- Einsatz einer objekt-orientierten Datenbank
- Einsatz einer relationalen Datenbank in objekt-orientierter Entwicklungsumgebung
- Einsatz eines ORM-Frameworks *(object relational mapping)*
- Einsatz eines EJB-Frameworks
- Zwei Beispiele: "Bank" und "Malen"

  "Bank" hat relativ wenige Objekte, die sich allerdings oft ändern können.
  "Malen" generiert sehr viele Objekte, die dauernd benötigt werden, sich aber nicht ändern.
- `odbueb.zip`

```
odbueb
   odbueb.pdf
   bank
      code
      musterRDB
         meta
            tabellen
         code
   malen
      code
      musterRDB
         meta
            tabellen
         code
   db4o.jar
   mjt.jar
```

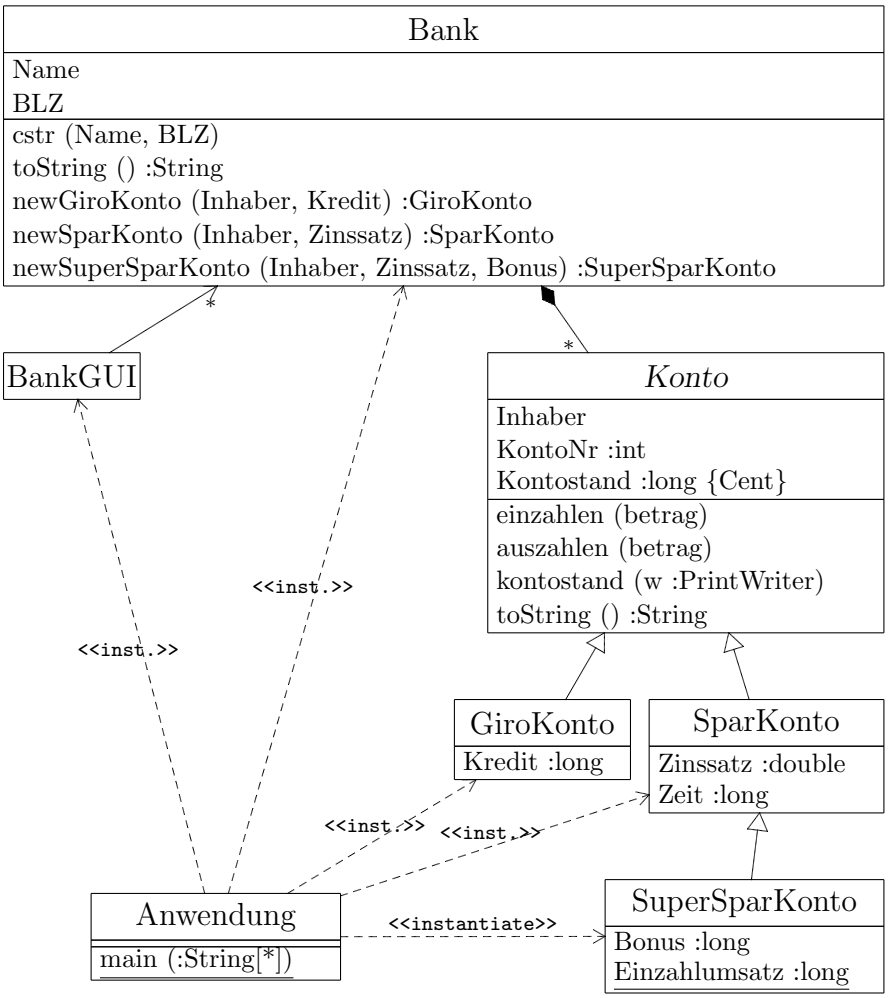
E.1 Das Beispiel "Bank"

Das folgende Java-Programm modelliert Banken mit ihren verschiedenartigen Konten.

Aufgabe: Machen Sie sich mit dem Programm vertraut. (Schauen Sie aber möglichst nicht so tief in die GUI. Das ist – wie aber fast jede GUI – ein ziemlicher Hack.)

Das `main` befindet sich in der Klasse `Anwendung`.

Klassendiagramm:



Code: Verzeichnis: `odbueb/bank/code`

E.2 Das Beispiel "Malen"

Mit dem folgenden Java-Programm `Malen` kann man mit der Maus in verschiedenen Farben auf ein `JPanel` malen. Bei gedrücktem Mausknopf werden zwischen der letzten und der aktuellen Mausposition sogenannte "Grafle" (Strich, Doppelstrich usw.) gezeichnet.

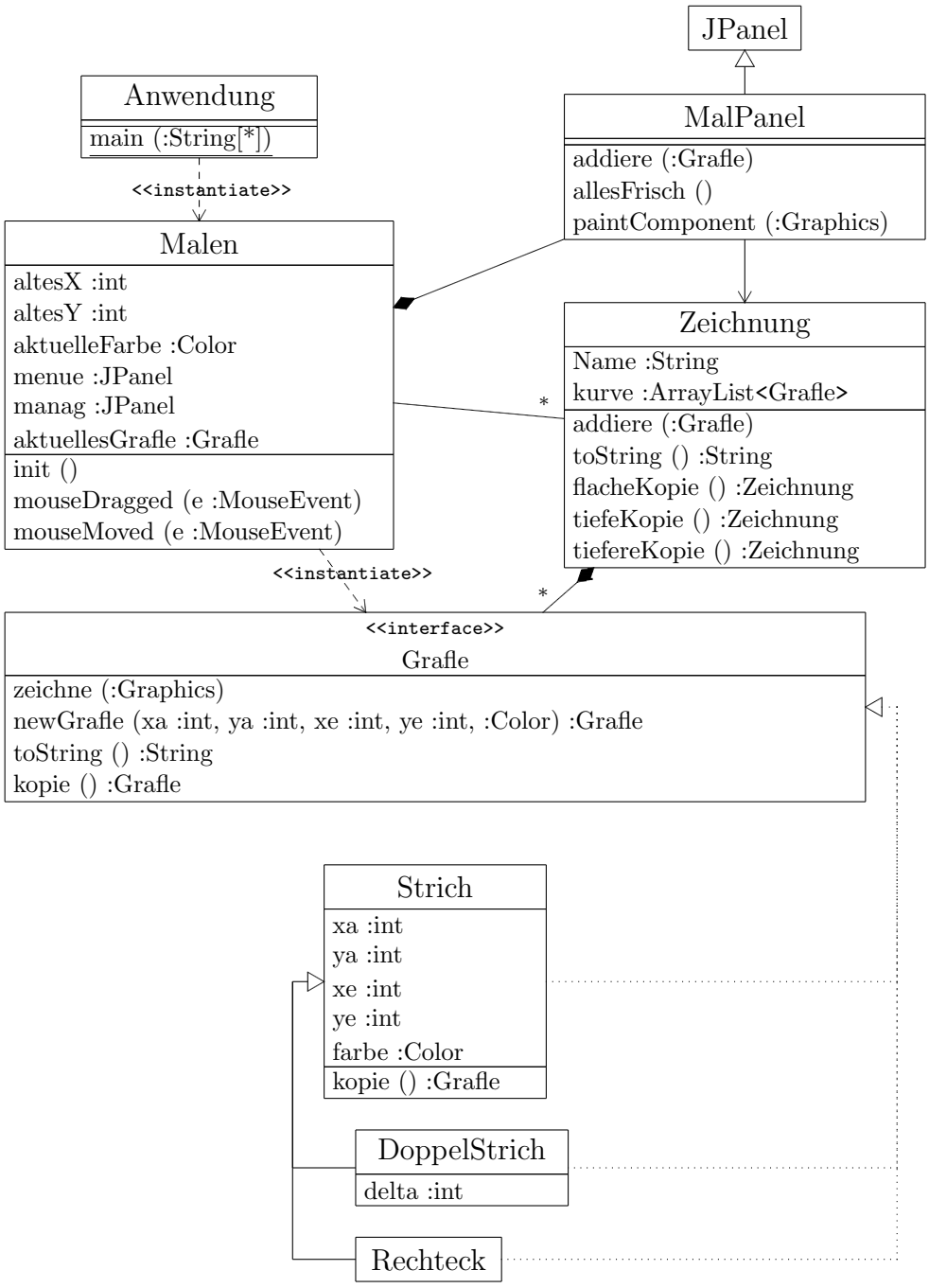
Die Menüzeile unten enthält GUI-Komponenten, die für die Datenbank-Anwendung vorbereitet sind und im Moment weitgehend ohne Funktion sind.

Aufgabe: Machen Sie sich mit dem Programm vertraut. (Welches Design-Pattern wird hier z.B. verwendet?)

Das `main` befindet sich in der Klasse `Anwendung`.

Aufgabe: Erstellen Sie eine weiteres "Grafle". Die Klasse `java.awt.Graphics` kann Ihnen dazu Ideen liefern.

Klassendiagramm:



Code: Verzeichnis: `odbueb/malen/code`

E.3 Objekt-orientierte Datenbank

Aufgabe: "Persistieren" Sie das Beispiel "Bank" **und** das Beispiel "Malen" unter Verwendung der objekt-orientierten Datenbank db4o.

Jede Sprache, die von db4o unterstützt wird, kann verwendet werden. Ein Coaching ist nur bei Java möglich.

Solange man entwickelt, ist es sicher nützlich, Code zu haben, der den Datenbank-Inhalt löscht.

Die GUI-Klassen sollen nicht persistiert werden.

E.3.1 Beispiel "Bank"

Wenn das Programm beendet wird, dann sind alle Ihre Banken und Konten wieder weg. Sie sollen jetzt dafür sorgen, dass beim nächsten Programmstart alles wieder da ist.

Vielleicht nicht ganz selbstverständlich: Alle Banken sollen in **einer** Datenbank zu finden sein.

Fragen:

- Wird die Datenbank ordentlich geschlossen? (`EXIT_ON_CLOSE`?, siehe `setDefaultCloseOperation`, `WindowListener`, `WindowAdapter`)
- Erweiterung des Beispiels durch richtige Konten-Inhaber-Objekte?
- Erweiterung des Beispiels um Transaktionen?
- Schreiben Sie in einem eigenen Programm verschiedene, möglichst auch verrückte Queries. Zum Beispiel:
  1. Geht nur mit Transaktionen: Von welchem Konto wurde innerhalb einer gewissen Zeitspanne eine gewisse Summe abgehoben.

E.3.2 Beispiel "Malen"

Wenn das Programm beendet wird, dann ist Ihre hübsche Zeichnung weg. Sie sollen jetzt dafür sorgen, dass jede Zeichnung unter einem Namen (und evtl. Datum) abgespeichert werden kann und beim nächsten Start wieder unter diesem Namen gefunden werden kann.

Eigentlich selbstverständlich: Alle Zeichnungen sollen in **einer** Datenbank zu finden sein.

Fragen:

- Kann man alte Zeichnungen verändern?
- Was passiert mit den Grafle von gespeicherten Zeichnungen, wenn man "Alles frisch" macht?
- Kann man die veränderte Zeichnung als neue Kopie speichern?
- Werden immer wieder neue Objekte der Klasse `Color` angelegt?

- Überlegen Sie, ob man Grafle eventuell wiederverwenden sollte. Was müsste man dazu tun?
- Wird die Datenbank ordentlich geschlossen? (`EXIT_ON_CLOSE`?, siehe `setDefaultCloseOperation`, `WindowListener`, `WindowAdapter`)
- Schreiben Sie in einem eigenen Programm verschiedene, möglichst auch verrückte Queries. Zum Beispiel:
  1. Alle Grafle einer gewissen Farbe
  2. Alle Grafle einer gewissen Länge
  3. Alle Grafle an einer gewissen Position
  4. Alle Zeichnungen, die ...
  5. Alle "ersten" Grafle einer Grafleserie.
  6. Experimentieren Sie mit Indexen für die Grafle. D.h. z.B. legen Sie Indexe für Koordinaten des Grafle an und suchen Sie nach Grafle in einem gewissen Bereich. Zeigen Sie Performanzunterschiede mit und ohne Index.

E.4 Relationale Datenbank

Idee: Bei den persistenten Objekten soll jeder Zugriff auf ein Datenelement sofort als Zugriff auf die relationale Datenbank realisiert werden. Jede `set/get`-Methode wird durch einen Datenbankzugriff realisiert. D.h. die Klassen der persistenten Objekte haben außer einer Objekt-ID (OID) **keine** Datenelemente (außer eventuell noch transienten Datenelementen).

Aufgabe: Wenden Sie dieses Konzept auf das Beispiel "Bank" und "Malen" an.

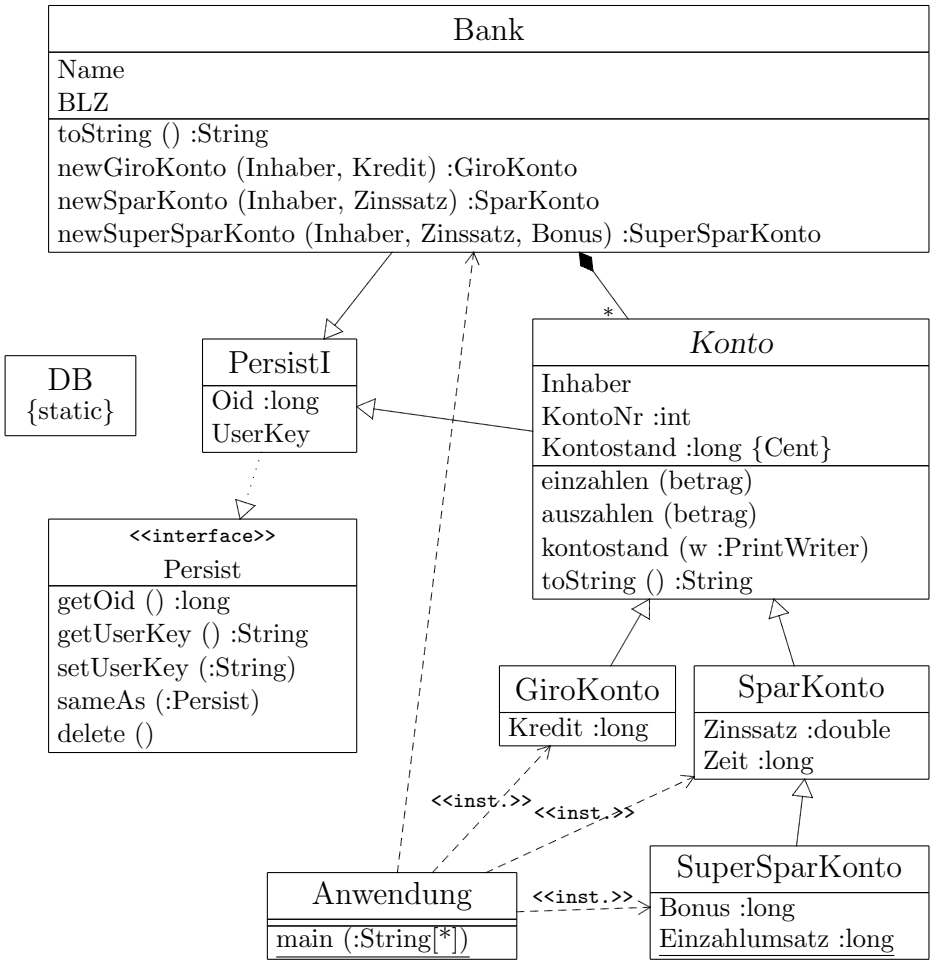
Jede Sprache-Datenbank-Kombination kann verwendet werden. Ein Coaching ist nur bei Java und MySQL möglich.

Fragen:

- Wie beurteilen Sie bei dieser Lösung die Trennung von Business-Code und Datenbank-Code?
- Diskutieren Sie sämtliche Beziehungen (d.h. Wie sind sie in den Tabellen realisiert und welche Konsequenzen hat das?). Beziehungen mit ein- oder beidseitiger Navigation: one-to-one, one-to-many, many-to-many, Vererbung.
- Welche Erfahrungen machen Sie mit der Performanz?

E.4.1 Musterlösung Beispiel Bank

Bei den persistenten Objekten muss eine systemweit eindeutige Objekt-ID (OID) verwaltet werden, die wir als Datenelement `oid` in einer Klasse `PersistI` anlegen, die die Schnittstelle `Persist` realisiert. Jede Klasse persistenter Objekte muss von `PersistI` erben oder mindestens `Persist` realisieren.



Das Attribut `oid` ist das einzige Datenelement und es ist nur in der Klasse `PersistI` angelegt. Hinter allen anderen Attributen sind nur `set/get`-Methoden zu sehen, die unter Verwendung von `oid` auf die jeder Klasse zugeordneten Tabellen zugreifen. (Das Attribut `UserKey` der Klasse `Persist` wurde angelegt, um den Empfehlungen von ODMG gerecht zu werden. Es kann z.B. durchaus nützlich sein, um Wurzel-Objekte zu finden.)

Schon in der Datenbank gespeicherte Objekte müssen mit ihrer `Oid` konstruiert werden. Dazu muss aber der Klassenname bekannt sein. Daher verwaltet die Tabelle `persist`, die die Klasse `PersistI` repräsentiert, zu jeder `oid` auch den Klassennamen. Jede Klasse muss einen Konstruktor `cstr (oid :long)` haben, der die OID schließlich an die Klasse `PersistI` hinunterreicht.

Bei allen anderen Konstruktoren werden neue Datenbankobjekte erstellt. Dabei generiert der Default-Konstruktor von `PersistI` eine neue OID. Die Konstruktoren müssen dann für die Eintragung einer neuen Zeile in die jeweilige zur Klasse gehörenden Tabelle sorgen.

Es folgen die SQL-Statments, um die Tabellen anzulegen:

Datei: `../odbueb/bank/musterRDB/meta/tabellen`

```
drop table lastoid;
create table lastoid
   (
   oid   bigint
   );
insert into lastoid(oid) values (0);
drop table persist;
create table persist
   (
   oid   bigint,
   klassenname varchar (255),
   userkey  varchar (255)
   );
drop table bank;
create table bank
   (
   oid   bigint,
   name  varchar (255),
   blz   varchar (255)
   );
drop table konto;
create table konto
   (
   oid   bigint,
   inhaber  varchar (255),
   kontonr  int,
   kontostand  bigint
   );
drop table bankkonto;
create table bankkonto
   (
   ooid  bigint, -- oid of one-side
   moid  bigint, -- oid of many-side
   ordnung  int
   );
drop table girokonto;
create table girokonto
   (
   oid   bigint,
   kredit   bigint
   );
drop table sparkonto;
create table sparkonto
   (
   oid   bigint,
   zinssatz float,
   zeit  bigint
   );
drop table supersparkonto;
create table supersparkonto
   (
   oid   bigint,
   bonus bigint
   );
```

Die Tabelle `lastoid` verwaltet nur die letzte vergebene OID. Wir verwenden kein vorhandenes oder selbstgemachtes Autoinkrement, da dabei die letzten OODs oft wiederverwendet werden. OIDs sollten in der Regel nie wiederverwendet werden, um nicht Beziehungen wieder zufällig zu kitten, wenn diese nicht sauber beim Löschen eines Objekts abgebaut wurden.

Die Tabelle `bankkonto` repräsentiert die one-to-many-Beziehung zwischen `Bank` und `Konto`, wobei `ooid` (one-side) ein Fremdschlüssel ist, der `oid` in Tabelle `bank` referenziert, und `moid` (many-side) ein Fremdschlüssel ist, der `oid` in Tabelle `konto` referenziert. Da sehr oft die Reihenfolge erhalten bleiben soll, wird sie mit `ordnung` verwaltet. Diese Beziehung wird nur von der Bank (one-side) aus verwaltet. Navigation ist nur von Bank nach Konto möglich. Wenn wir eine Navigation in umgekehrter Richtung benötigen, dann hätte `Konto` ein Datenelement vom Typ `Bank` und die

Tabelle `konto` eine Spalte `abank`. Das ist zwar redundant zur Tabelle `bankkonto`, aber eine Redundanz, die leicht zu beherrschen ist und sicherlich einen Performanz-Gewinn bedeutet.

Bei einem im wesentlichen navigationalen System benötigt man eventuell nur für die OIDs Indexe. Hier haben wir bisher ganz auf Indexe verzichtet.

Eine many-to-many-Beziehung mit beidseitiger Navigation würde mit diesem Konzept am einfachsten mit zwei one-to-many-Beziehungen realisiert werden. Das hätte etwas Redundanz zur Folge, die aber sehr leicht zu beherrschen ist, das Design modularer macht und das System wahrscheinlich performanter macht!

Redundanz, die nur OIDs betrifft, ist unschädlich und damit erlaubt!

Falls gefordert würde, dass ein Konto seine Bank kennt, würde man das durch ein Attribut `aBank:Bank` in `Konto` und einer Spalte `abank` in der Tabelle `konto` realisieren. Obwohl das wieder wegen der Tabelle `bankkonto` redundant ist, ist es aus denselben Gründen vernünftig.

Die Klasse `DB` stellt statische Methoden zur Verfügung, um über JDBC auf die relationale Datenbank zuzugreifen:

DB
HostDB User Password :Connection :Statement
getConnection () :Connection setStmtNull () getStmt () :Statement close () newOid (:Class) :long newZeile (p :Persist, tabelle :String) delZeile (p :Persist, tabelle :String) delZeilen (p :Persist, tabelle :String) findePersist (oid :long) :Persist findePersist (userkey :String) :Persist getString (p :Persist, tabelle :String, spalte :String) :String setString (p :Persist, tabelle :String, spalte :String, :String) getInt (p :Persist, tabelle :String, spalte :String) :int setInt (p :Persist, tabelle :String, spalte :String, :int) getLong (p :Persist, tabelle :String, spalte :String) :long setLong (p :Persist, tabelle :String, spalte :String, :long) getDouble (p :Persist, tabelle :String, spalte :String) :double setDouble (p :Persist, tabelle :String, spalte :String, :double) getPersist (p :Persist, tabelle :String, spalte :String) :Persist setPersist (p :Persist, tabelle :String, spalte :String, p2 :Persist) getNumPersists (p :Persist, tabelle :String) :int getPersists (p :Persist, tabelle :String) :ArrayList<Persist> getPersistsUnordered (p :Persist, tabelle :String) :ArrayList<Persist> getPersist (p :Persist, tabelle :String, ordnung :int) :Persist setPersists (p :Persist, tabelle :String, pp :Persist[*]) setPersists (p :Persist, tabelle :String, pp :ArrayList<Persist>) setPersist (p :Persist, tabelle :String, i :int, p2 :Persist) addPersists (p :Persist, tabelle :String, pp :Persist[*]) addPersist (p :Persist, tabelle :String, p2 :Persist) rmvPersist (p :Persist, tabelle :String, p2 :Persist) rmvPersist (p :Persist, tabelle :String, i :int) getAlle (klasse :Class) :ArrayList<Persist> getAlle (userkey :String) :ArrayList<Persist> beginTA () commitTA () rollbackTA ()

Hier sollte man wahrscheinlich ein Singleton-Pattern einsetzen, um DBS-Varianten abdecken zu können.

Wenn Objekte geloescht werden, dann geht das von Subklasse zu Superklasse bis man bei `PersistI` landet, die die OID auf -1 setzt. Die Methode `delete ()` hat dann immer die Struktur:

```
public void delete ()
   {
   // eventuell Einträge in Beziehungstabellen löschen
   // oder überhaupt Beziehungen vernünftig abbauen
   DB.delZeile (this, tabelle);
   super.delete ();  // Muss letzte Anweisung sein.
   }
```

Es gibt keine direkte Möglichkeit, die gelöschten Objekte aus dem Hauptspeicher zu entfernen. Man muss auf jeden Fall dafür sorgen, dass sie nicht mehr referenziert werden, da es sonst beim nächsten Zugriff auf das Objekt zum Absturz kommt. Ohne Referenz auf das gelöschte Objekt kann dann auch der GC schließlich seine Arbeit tun.

Bemerkungen:

1. Die persistenten Klassen (Entitäten) unterscheiden sich sehr stark von den bisherigen Klassen, allerdings in einer relativ leicht automatisierbaren Form.

2. Wenn man von den Löschoperationen absieht, dann unterscheidet sich das Anwendungsprogramm (gemeint ist `BankGUI`) nur in zwei Zeilen vom nicht persistierten Programm: Beim Ausstieg wird noch `DB.close ()` aufgerufen. Ferner gibt es eine zusätzliche Zeile, um auch den eingegebenen Userkey zu speichern.

3. Der Code zum Löschen eines Kontos oder einer Bank (`machKontoLoeschknopf ()`, `machBankLoeschknopf ()`) unterscheidet sich ziemlich vom nicht-persistierten Code. Man muss natürlich die Methode `delete ()` irgendwann aufrufen. Ferner muss man einige `remove`-Operationen nachprogrammieren. Denn die Objektidentität muss anstatt mit "`==`" mit `sameAs (:Persist)` geprüft werden. Das tun die Operationen von Collections natürlich nicht. Außerdem muss man sehr aufpassen, dass schon gelöschte Objekte nicht irgendwo noch in der GUI verwendet werden. Wegen gewisser Swing-Automatismen hat es hier etwas gedauert, bis der Code lief.

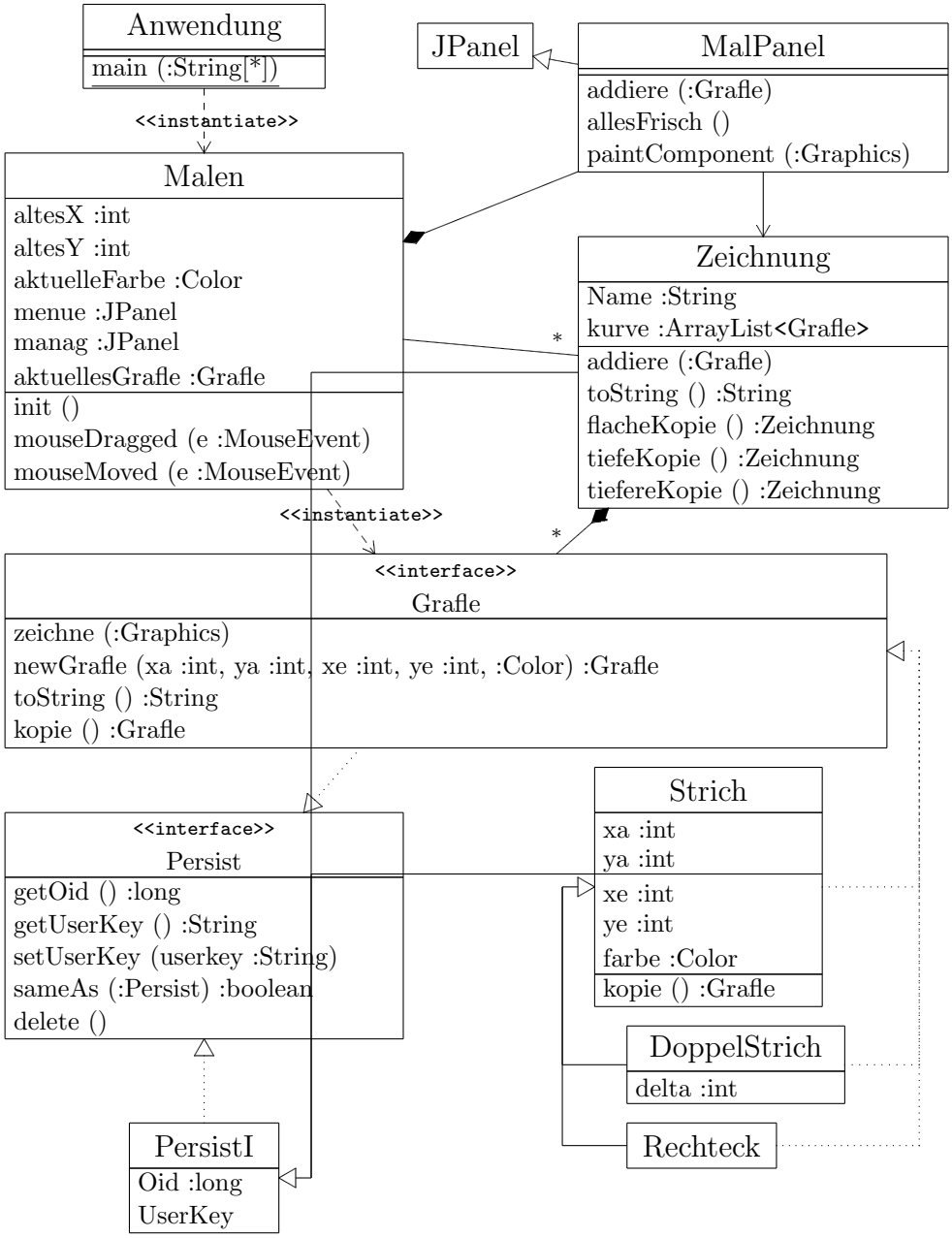
4. Nur bei Überweisungen wurden Transaktionen verwendet. Man müsste überprüfen, an welchen Stellen das sonst noch wichtig wäre. Wahrscheinlich ist das bei jeder Manipulation von Konten notwendig.

5. An der Performanz merken wir nichts davon, dass wir immer auf die Datenbank zugreifen. Wir arbeiten bisher lokal. Wie das übers Netz aussieht, muss noch getestet werden.

Der Code der Musterlösung findet sich in:
Datei: `../odbueb/bank/musterRDB/meta/bank`

E.4.2 Musterlösung Beispiel Malen

Wir gehen hier genauso vor wie beim Beispiel "Bank".



Es folgen die SQL-Statments, um die Tabellen anzulegen:

Datei: `../odbueb/malen/musterRDB/meta/tabellen`

```
drop table lastoid;
create table lastoid
   (
   oid   bigint
   );
insert into lastoid (oid) values (0);
drop table persist;
create table persist
   (
   oid   bigint primary key,
   klassenname varchar (255),
   userkey  varchar (255)
   );
drop table farbe;
create table farbe
   (
   oid   bigint primary key,
   deutsch  varchar (255)
   );
drop table strich;
create table strich
   (
   oid   bigint primary key,
   xa int,
   ya int,
   xe int,
   ye int,
   farbe bigint
   );
drop table doppelstrich;
create table doppelstrich
   (
   oid   bigint primary key,
   delta int
   );
drop table rechteck;
create table rechteck
   (
   oid   bigint primary key
   );
drop table zeichnung;
create table zeichnung
   (
   oid   bigint primary key,
   name  varchar (255)
   );
drop table zeichnunggrafle;
create table zeichnunggrafle
   (
   ooid  bigint,
   moid  bigint,
   ordnung  int
   );
create index ooidindex on zeichnunggrafle(ooid);
```

Die Tabelle `zeichnunggrafle` repräsentiert die one-to-many-Beziehung zwischen `Zeichnung` und `Grafle`, wobei `ooid` (one-side) ein Fremdschlüssel ist, der `oid` in Tabelle `zeichnung` referenziert, und `moid` (many-side) ein Fremdschlüssel ist, der `oid` in Tabelle `strich`, `doppelstrich` oder `rechteck` referenziert.

Bei einem im wesentlichen navigationalen System benötigt man eventuell nur für die OIDs Indexe. Hier haben wir für alle `oid` und `ooid` Indexe angelegt, da wir schnell sehr viele Objekte generieren.

Bemerkungen:

1. Die Lösung lässt noch vieles zu wünschen übrig, was die Tiefe der Löschoperationen angeht.

2. Sieh auch die Bemerkungen beim Beispiel "Bank"
3. Diese Anwendung läuft mit diesem Persistenz-Konzept nicht besonders gut, da dauernd sehr viele Objekte benötigt werden, was ungheuer viele Datenbankzugriffe erzeugt. Übers Netz dürfte das unerträglich werden.
4. Das Anlegen von Indexen hat einen Performanzgewinn gebracht. Ohne Indexe ruckelt das gewaltig. Mit Index ist es erträglich. Um das zu testen, muss man einfach die `PRIMARY KEY`-Einschränkungen und die eine `CREATE INDEX`-Anweisung herausnehmen.
5. Wenn eine Zeichnung gelöscht wird, so werden momentan ihre Grafle nicht mitgelöscht. Das ergibt ein *database memora leak*. Das hätte natürlich so gemacht werden können, allerdings gibt es dann ein Problem mit den flachen Kopien einer Zeichnung, die dann ihre Grafle verlieren würden.

Der Code der Musterlösung findet sich in:
Datei: `../odbueb/malen/musterRDB/meta/malen`

E.5 ORM-Framework Hibernate

Aufgabe: Wenden Sie Hibernate auf das Beispiel "Malen" und "Bank" an.

E.6 XML-Datenbank

Aufgabe: Wenden Sie eine XML-Datenbank auf das Beispiel "Malen" und "Bank" an.

E.7 EJB-Framework

Aufgabe: Wenden Sie ein EJB-Framework auf das Beispiel "Malen" und "Bank" an.

Literaturverzeichnis

[1] Carlo Batini, Stefano Ceri und Shamkant B. Navathe, "Conceptual Database Design", The Benjamin/Cummings Publishing Company 1992

[2] Christian Bauer und Gavin King, "Java Persistence with Hibernate", Manning Publications 2006

[3] Grady Booch, "Object-Oriented Analysis and Design", The Benjamin/Cummings Publishing Company 1994

[4] Rainer Burkhardt, "UML – Unified Modeling Language", Addison-Wesley 1997

[5] Stephen Cannan und Gerard Otten, "SQL — The Standard Handbook", McGraw-Hill International 1993

[6] Jeff Carpenter und Eben Hewitt, "Cassandra – The Definitive Guide", O'Reilly

[7] Rick G. G. Cattell und Douglas K. Barry, "The Object Data Standard: ODMG 3.0" Morgan Kaufmann Publishers 2000

[8] Peter Pin-Shan Chen, "The Entity-Relationship Model: Toward a Unified View of Data", ACM Transactions on Database Systems **1**, 9-36 (1976)

[9] Peter Coad und Edward Yourdon, "Object-Oriented Analysis", Prentice-Hall 1991

[10] Peter Coad und Edward Yourdon, "Object-Oriented Design", Prentice-Hall 1991

[11] Edgar Frank Codd, "The Relational Model for Database Management Version 2", Addison-Wesley 1990

[12] Thomas Connolly und Carolyn Begg, "Database Systems", Addison-Wesley 2002

[13] C. J. Date, "An Introduction to Database Systems", Addison-Wesley

[14] C. J. Date und Hugh Darwen, "A Guide to the SQL Standard", Addison-Wesley 1993

[15] David Flanagan, Jim Farley, William Crawford und Kris Magnusson, "Java Enterprise in a Nutshell", O'Reilly & Associates 1999

[16] Stefan Edlich, Achim Friedland, Jens Hampe, Benjamin Brauer und Markus Brückner, "NoSQL – Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken", Hanser

[17] Ramez Elmasri und Shamkant B. Navathe, "Fundamentals of Database Systems", The Benjamin/Cummings Publishing Company

[18] Volker Gaede und Oliver Günther, "Multidimensional Access Methods", ACM Computing Surveys **30**, 170-231 (1998)

[19] Hector Garcia-Molina, Jeffrey D. Ullman und Jennifer Widom, "Database Systems The Complete Book" Pearson Prentice Hall

[20] Jim Gray und Andreas Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann (1992)

[21] Terry Halpin und Tony Morgan "Information Modeling and Relational Databases", Morgan Kaufmann 2008

[22] Andreas Heuer und Gunter Saake, "Datenbanken: Konzepte und Sprachen" MITP 2000

[23] Uwe Hohenstein und Volkmar Pleßer, "Oracle8 Effiziente Anwendungsentwicklung mit objektrelationalen Konzepten", dpunkt.verlag 1998

[24] John G. Hughes, "Objektorientierte Datenbanken", Hanser und Prentice-Hall International 1992

[25] Jennifer Little in "High Performance Web-Databases", ed. Sanjiv Purba, Auerbach 2001

[26] Alfons Kemper und André Eickler, "Datenbanksysteme", R.Oldenbourg 1999

[27] W. Kim, "On Optinizing an SQL-like Nested Query", ACM Transactions on Database Systems 7, 443 (1982)

[28] Michael Kofler, "MySQL", Addison-Wesley 2001

[29] Jochen Ludewig, "Einführung in die Informatik", Verlag der Fachvereine Zürich 1989

[30] D. Maier, "The Theory of Relational Databases", Computer Science Press 1983

[31] Jim Melton und Alan R. Simon, "Understanding The New SQL: A Complete Guide", Morgan Kaufmann 1993

[32] Greg Riccardi, "Principles of Database Systems with Internet and Java Applications", Addison-Wesley 2001

[33] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy und William Lorensen, "Object-Oriented Modeling and Design", Prentice Hall (Deutsch bei Hanser)

[34] Gunter Saake und Andreas Heuer, "Datenbanken: Implementierungstechniken" MITP 2005

[35] Gunter Saake und Kai-Uwe Sattler, "Datenbanken & Java JDBC, SQLJ und ODMG" dpunkt 2000

[36] Pramod J. Sadalage und Martin Fowler, "NoSQL Distilled" Addison-Wesley 2013

[37] Sanjiv Purba in "High Performance Web-Databases", ed. Sanjiv Purba, Auerbach 2001

[38] "Database Language SQL", ISO/IEC 9075:1992 oder ANSI X3.135-1992

[39] Michael Stonebraker und Paul Brown, "Object-Relational DBMSs", Morgan Kaufmann 1999

[40] Christof Strauch, "NoSQL Databases"
`http://home.aubg.bg/students/ENL100/Cloud%20Computing/Research%20Paper/nosqldbs.pdf`
(24. Januar 2014)

[41] Niklaus Wirth, "Algorithmen und Datenstrukturen" Teubner 1986