Grundlagen der Datenbanken

Sommersemester 1995/96

Christoph Kreitz

FG Intellektik, TH Darmstadt, Telephon (06151) 16-2863

kreitz@intellektik.informatik.th-darmstadt.de

1. Einführung:
   - Datenbanksysteme: Verwendungszweck und historische Entwicklung
   - Grundprinzipen von Datenbanksystemen
   - Funktionen und Komponenten von Datenbankmanagementsystemen
2. Lehrziele und geplanter Aufbau der Vorlesung

---

3. Organisatorisches

Wozu Datenbanksysteme?

Daten sind ein wichtiger Aktivposten jedes Unternehmens

- **Verhindere Datenredundanz**
  - jedes Anwendungsprogramm verwaltet seine eigenen Daten
  - Datenmengen verschiedener Anwendungen überlappen sich

  $\Rightarrow$ Speicherplatzverschwendung, Inkonsistenzen

- **und Datenabhängigkeit** von Hardware & Betriebssystem

- **durch Datenintegration**
  - zentrale, einheitliche Darstellung des Datenbestandes (Datenbank)
  - Verwaltung durch **D**aten**b**ank-**M**anagement-**S**ystem
    (Einfügen, Lesen, Ändern, Löschen)
  - Benutzergerechte Anfragesprachen, unabhängig von konkreter Datendarstellung
  - Schnittstellen zu Anwendungsprogrammen mit lokaler Sicht

Vorteile von Datenbanksystemen

- **Effiziente Verwaltung großer Datenmengen**
  - Redundanzfreiheit durch integrierten Datenbestand
  - interne Optimierung der Datenhaltung
  - effiziente Such- und Änderungsmechanismen
- **Korrektheit der Daten**
  - sichergestellt durch Einhaltung von Konsistenzregeln
  - auch im Mehrbenutzerbetrieb (Concurrency Control/Transaktionskonzept) keine unerwünschte Nebeneffekte bei gleichzeitigem Datenzugriff
- **Datenunabhängigkeit**
  - Anwendungsicht entkoppelt von interner Datendarstellung
  - physikalisch: Unabhängigkeit von Speicherstruktur, -medien und Zugriffspfaden
  - logisch: Unabhängigkeit von logischer Beschreibung der Daten
- **Zugriffskontrolle für Anwender (lokale Sichten)**

  $\Rightarrow$ Datenschutz (kein unbefugter Zugriff)

  $\Rightarrow$ Datensicherheit (kein ungewollter Datenverlust)

Einsatzgebiete

- **Verwaltung großer Datenmengen**
  - Bibliotheken, Kontoführung, Telephonvermittlung und -abrechnung
  - Reservierungen, Buchhaltung, Auftragserfassung, Aktienhandel, ...
- **Viele Objekte**
  - 100000 Bücher, 2000 Benutzer, 5000 Ausleihvorgänge/Woche
  - 10000 Konten, 4000 Kunden, 100000 Buchungen/Woche
  - 40 Millionen Anschlüsse, 38 Millionen Kunden, 200 Millionen Gespräche/Tag
- **Wenige Objekttypen**
  - Buch (Autor, Titel, ...) Benutzer (Name, Adresse, ...), Ausleihvorgang, ...
  - Konto, Kunde, Buchung
  - Anschluß, Kunde, Gespräch, Zeitschablone, ...
- **Viele Anwender gleichzeitig**
- **Wenige Transaktionsarten, hohe Wiederholrate**
- **Kurze Antwortzeiten erforderlich**
  - Platzreservierung interaktiv, Antwortzeit unter 1 Sekunde

Nachteile von Datenbanksystemen

- **Allgemeinsoftware weniger effizient als Spezialsoftware**
  - Optimierung schwer bei konkurrierenden Anforderungen
- **Zusätzliche Kosten für DBMS**
- **Zusätzliche Kosten für Hardware**
  - Optimales Medium: Magnetplatten (schnell und preiswert)
- **Spezialisiertes Personal erforderlich**
  - Datenbankadministrator
- **Verwundbarkeit durch Zentralisierung der Daten**

  $\mapsto$ verteilte Datenbanken

Historische Entwicklung I

- **Urzeit: (vor 1960)**
  - Berechnungsvorgänge wichtiger als Verarbeitung von Datenmengen
  - Datenhaltung auf Lochkarten oder Magnetbändern
  - Batchverarbeitung
- **Steinzeit: (1960–1965)** **direkter Datenzugriff**
  - Datenorganisation und Zugriff in Anwendung integriert
  - Medienabhängigkeit (Plattenspeicher, Magnetbänder für große Datenmengen)
  - Struktur: Datensätze fester oder variabler Länge
  - Zugriff: sequentiell, direkt mit Schlüssel, indexsequentiell mit Schlüssel
- **Dunkles Mittelalter: (1965–1970)** **Geräteunabhängigkeit**
  - Datenverwaltung wichtiger als Verarbeitung und Berechnung
  - Datenzugriff und Organisation durch Dateiverwaltungssysteme (separat von Anwendu
  - Zugriffe auch über mehrere Schlüssel
  - Logische Datenstruktur und Synchronisation durch Anwenderprogramm

  $\Rightarrow$ Redundanzen, Inkonsistenzen, keine logische Datenunabhängigkeit

  $\Rightarrow$ Datenschutz, Datensicherheit nur durch das Anwenderprogramm

Zugriff auf Daten ohne spezielle Verwaltung

Heuer Saake, Abb 1.2 einkleben

Dateiverwaltungssysteme

Heuer Saake, Abb 1.3 einkleben

Datenbankmanagementsysteme

Heuer Saake, Abb 1.4 einkleben

Historische Entwicklung II

- **Neuzeit: (1970 – . . .)** **Standard DBMS**
  - Zentrale Verwaltung der Daten (separat von Anwendung)
  - Logische Struktur, Konsistenzkontrolle, Synchronisation, etc durch DBMS
  - Kontrolle von Inhalt, Datensicherheit, Datenschutz durch Datenbankadministrator
  - erweiterte Datenstrukturen möglich (Felder, Records, Tupel, Relationen)
  - erweiterte Zugriffsmöglichkeiten (Indizes, inhalts- oder mengenorientiert, interaktiv. .

  ⇒ Datenunabhängigkeit, wenig Redundanz, syntaktische Integrität

  ⇒ Datenschutz, Datensicherheit

  ⇒ minimale semantische Integrität

- **Aktuell: (1985 – . . .)** **Objektorientierung**
  - Anwendungsbezogene Objektklassen ersetzen Datenstruktur + Operationen
  - Redundanz- und Konsistenzkontrolle durch Typenhierarchie und Vererbung
  - Datenunabhängigkeit durch Datenkapselung und Schnittstellen

  ⇒ Ausrichtung auf 'Nichtstandard Anwendungen' (CAD, CIM, multimediale IS)

  ⇒ erhöhte Datensicherheit und Datenschutz

  ⇒ semantische Integrität z.T. kontrollierbar

Aktuelle Forschung: Verteilte DBS

Geringere Verwundbarkeit durch Dezentralisierung

- **Client-Server Architektur**
  - Datenbankrechner (Server) + vernetzte Arbeitsplatzrechner (ohne Daten)
  - volle DBMS Funktionalität am Arbeitsplatz

  $\Rightarrow$ transparente Kommunikation

  $\Rightarrow$ i.w. wie zentrales DBS

- **Verteilte homogene DBS**
  - Daten verteilt über mehrere Knoten (z.T. redundant)
  - verteiltes DBMS mit einheitlichem Protokoll wickelt alle Transaktionen ab

  $\Rightarrow$ volle DBS Funktionalität in jedem Knoten, starke Koppelung

  $\Rightarrow$ transparente Kommunikation, Datenverteilung für Benutzer unsichtbar

  $\Rightarrow$ Ausfall eines Knotens kann Konsistenz stören ($\mapsto$ System blockiert)

- **Heterogene verteilte DBS**
  - unabhängige Datenverteilung
  - lokale DBMS Software, schwache Kopplung, verschiedene Autonomiegrade

  $\Rightarrow$ lokale Transaktionen unabhängig vom globalen System

  $\Rightarrow$ Konsistenz und globale Korrektheit schwer zu garantieren

Grundkonzeption von Datenbanksystemen

DBS = DB + DBMS

- **DB: Datenbank**
  - einheitlich beschriebene Darstellung diskreter Daten
  - Repräsentation auf externen und persistenten Speichermedien
- **DBMS: Datenbankmanagementsystem**
  - System zur zentralen Manipulation von Daten
  - ermöglicht Definition von internen Strukturen und externen Sichten
  - stellt effiziente Zugriffsoperationen, Schnittstellen, und deskriptive Anfragesprachen bereit
  - realisieren ein logisches Datenmodell
- **Datenmodell**
  - Definition von Datenstruktur, Operationen und Konsistenzregeln

Grundprinzipien moderner Datenbanksysteme

- **Trennung zwischen Schema und Instanz**
  - Das Schema beschreibt die Struktur der Daten
  - Konkrete Daten(-inhalte) sind Instanz eines Schemas
- **Trennung der Schemata in 3 Ebenen**
  - externe Schemata $\hat{=}$ einzelne Benutzersichten
  - konzeptuelles Schema $\hat{=}$ globale Sicht in standardisierter Form
  - internes Schema $\hat{=}$ konkrete physikalische Repräsentation der Daten

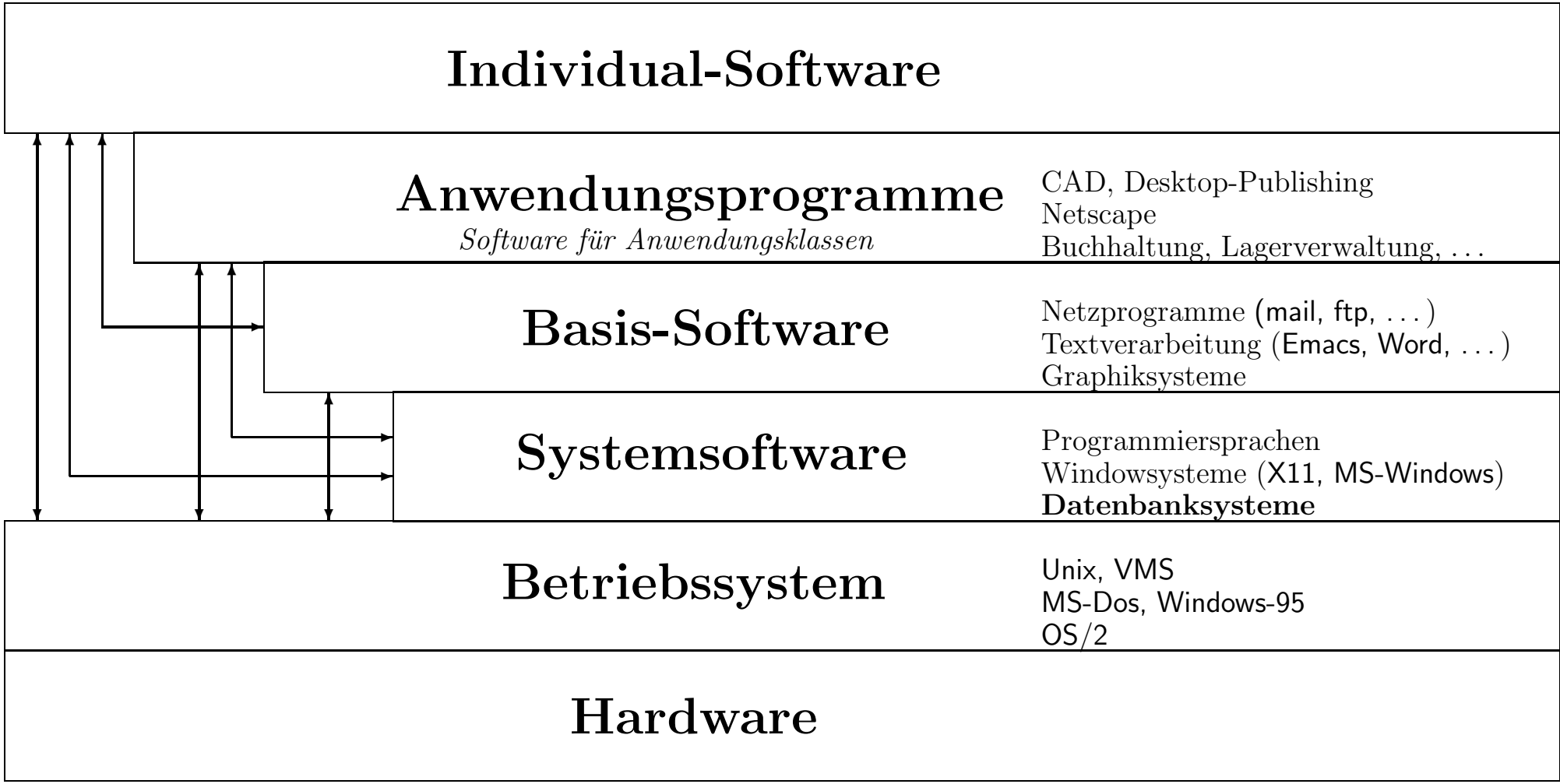
  $\Rightarrow$ **physikalische und logische Datenunabhängigkeit**
- **Deskriptive Anfragesprache (Data Manipulation Language**
  - oft in Wirts-Sprache (COBOL, PL/I, PASCAL, C) eingebettet
    durch Erweiterung der DML oder Precompiler in Wirtssprache
  - deskriptiv: Beschreibung der gesuchten Instanzen durch Prädikate
    (navigierend: positioniere Suche und verfolge Zeiger)

Entwicklungslinien von Datenmodellen

- **Hierarchisches bzw. Netzwerkmodell**
  - Zeigerstrukturen zwischen Daten
  - Schwache Trennung zwischen interner und logischer Datendarstellung
  - Navigierende DML
- **Relationale Datenbanken**
  - Daten in Tabellenstrukturen
  - Trennung von interner, logisch-konzeptioneller und externer Ebene
  - Deklarative DML
- **Deduktive Datenbanken**
  - wenige Objekte, viele Objektarten, komplizierte logische Operationen
  - Daten in Tabellenstrukturen, stark deklarative DML
  - Integration der DML in Programmiersprache
  - Anwendung: Expertensysteme
- **Objektorientierte Datenbanken**
  - viele Objekte, viele Objektarten, stark strukturierte Objekte
  - Daten in komplexen Objektstrukturen, deklarative und navigierende DML
  - Integration der DML in Programmiersprache, keine deutliche Trennung der Ebene
  - Anwendung: CAD, technische Anwendungen (zusammengesetzte Objekte)

Einordnung in der Software-Hierarchie

Systemprogramm ohne eigene Anwendung



Zentrale Funktionen eines DBMS

1. **Datenintegration**
   – einheitliche Beschreibung der Daten in einem Datenmodell
2. **Bereitstellung von Operationen und Sprachen**
   – für Zugriff und Änderung
3. **Katalog (Data Dictionary)** für Zugriff auf Datenbeschreibungen
4. **Bereitstellung von Benutzersichten**
   – Auswahl relevanter Daten in angepaßter Strukturierung
5. **Konsistenzüberwachung / Integritätskontrolle**
   – Änderungen dürfen Konsistenz der Daten nicht verletzen
6. **Datenschutz**
   – Verhinderung unauthorisierter Zugriffe (Datenschutzrecht / Werksspionage)
7. **Transaktionen**
   – (intern optimierte) Zusammenfassung von DB-Änderungen zu einer Funktionseinhei
   – atomar: Effekte unvollständiger Transaktionen unsichtbar
   – permanent: Effekte vollständiger, korrekter Transaktionen sind dauerhaft
8. **Concurrency Control**
   – Synchronisation konkurrierender Transaktionen – unsichtbar für Anwender
9. **Datensicherung** und Wiederherstellung nach Systemfehlern

Wichtige Komponenten eines DBMS

Heuer Saake 8

Relationenmodell – Datendefinition

Heuer Saake 9

Relationenmodell – Integritätsbedingungen

Heuer Saake 10

Relationenmodell – Anfrageoperationen

Heuer Saake 11 / 12a

Relationenmodell – Sichtdefinition

Heuer Saake 12b

Relationenmodell – Anfrageoptimierer

Heuer Saake 13

Relationenmodell – Interne Strukturen

Heuer Saake 14

Relationenmodell – Zugriffe intern

Heuer Saake 15

Aufgaben beim Einsatz eines DBMS

- **Datendefinition durch Datenbankadministrator**
  - Logische Strukturierung der Daten
  - Zuordnung externer Sichten zu internen Daten
  - Hilfsmittel: Data Definition Language (**DDL**)
- **Dateiorganisation durch Systemadministrator**
  - Zuordnung logischer Datenstrukturen zu interner Datenverwaltung
  - Hilfsmittel: Storage Structure Language (**SSL**)
- **Sichtdefinition durch Anwendungsadministrator**
  - Festlegung externer Sichten passend zur Anwendung
  - Hilfsmittel: Subscheme Data Definition Language (**SDDL**) (auch View Definition Language (**VDL**) genannt)
- **Interaktive Anfragen und Manipulationen durch Anwende**
  - Hilfsmittel: Data Manipulation Language (**DML**)
  - alternativ auch Menüs und Masken für ungeübte Benutzer
- **Programmierte Anwendungen**
  - Integriere Datenbankkonzepte in Anwendungsprogramme
  - Hilfsmittel: Data Base Programming Language (**DBPL**)

Sprachen und verantwortliche Personen oft überlappend

Ziel dieser Vorlesung

Datenbanksysteme effektiv nutzen

- **Datenbankadministration**
  - Verständnis von Datenmodellen
  - Kenntnis der wichtigsten Sprachen und ihrer Möglichkeiten
  - Bewertung individueller Vor- und Nachteile
  - Entwurfsprinzipien
- **Anwendungsadministration und -programmierung**
  - Ermittlung von Benutzeranforderung
  - Konfiguration und Verwendunge geeigneter Sichten
- **Direkte Anfragen an Datenbanken**
  - Anfragesprachen kennenlernen (soweit keine Menüs bereitgestellt)

$\Rightarrow$ Interne Ebene und Programmierung von DBMS weniger wichtig

- Thema von Vertiefungsvorlesungen

Geplanter Aufbau der Vorlesung

1. Grundkonzepte von Datenbanken (3.4. – 17.4)
- Einführung: Grundbegriffe
- Architektur von Datenbanksystemen
- Informations- und Datenmodelle

2. Das relationale Datenmodell (18.4. – 15.5.)
- Grundlagen, Algebra, Kalkül
- Anfragesprachen (SQL, Quel, QBE, ...)
- Entwurfstheorie

3. Alternative Datenmodelle (22.5. – 5.6.)
- Netzwerkmodell – CODASYL
- Objektorientierung

4. Transaktionen und Administration (6.6. – 20.6.)
- Transaktionskonzept, Recovery, Concurrency
- Datenbankadministration

5. Aktuelle Entwicklungen (26.6. – 4.7.)
- Aktive Datenbanken, Echtzeitdatenbanken, Heterogene Datenbanken
- Anwendungen in anderen Gebieten (Deduktive Datenbanken, ...)

Grundlagen der Datenbanken

Lektion 2

Architektur von Datenbanksystemen

1. Anforderungen an ein Datenbanksystem
2. Schema-Architektur – Strukturierung der Daten
3. System-Architektur – Strukturierung des Systems
   - ANSI/SPARC Architektur (3 Ebenen)
   - Fünf-Schichten Architektur (Schnittstellen)
   - Konkrete Architekturen
4. Anwendungsarchitekturen

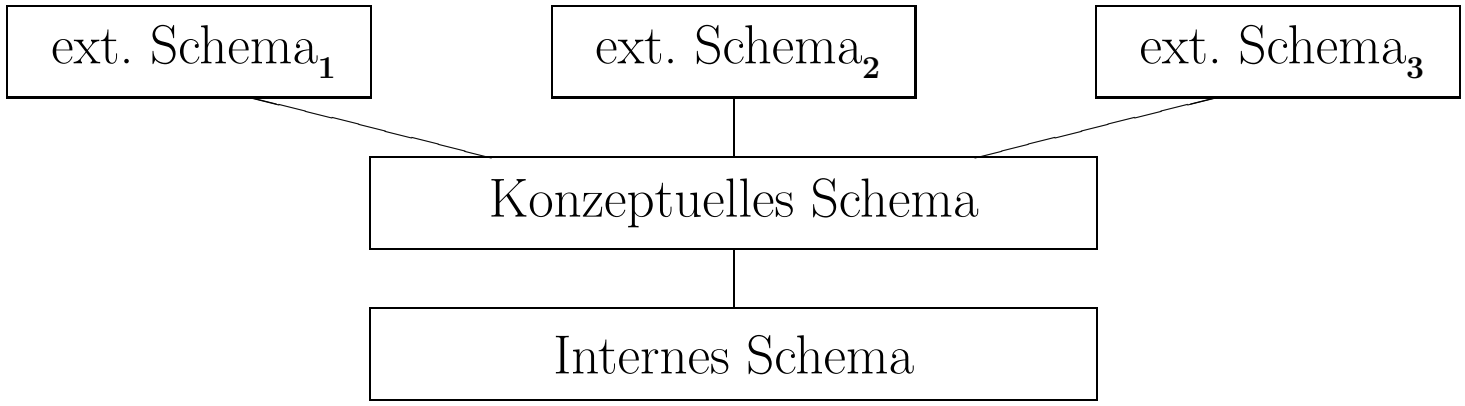
Anforderungen an ein Datenbanksystem

- **Kontrolle über die operationalen Daten**
  - Alle Daten gemeinsam benutzbar
  - Elimination von Redundanz
  - Durchsetzung von Standards
- **Kontrolle der Datenintegrität**
  - Zugriffskontrollen (Datenschutz)
  - logische ("Richtigkeit") und physische ("Sicherheit") Integrität
  - Synchronisation von Mehrbenutzerbetrieb
- **Leichte Handhabung der Daten**
  - Einfache Datenmodelle und Sprachen
  - Logische Sicht der Anwendung
  - Erweiterbarkeit der Benutzerklassen
- **Hoher Grad an Datenunabhängigkeit**
  - Geräte, Seitenzuordnungsstruktur, Speicherungsstruktur
  - Zugriffspfade, Datenstruktur
- **Effizienz**
  - Wirksamkeit des Zugriffs, losgelöst vom Anwendungsprogramm
  - globale Optimierung von Anfragen

Schema-Architektur: Strukturierung der Datenbank

Datenunabhängigkeit erfordert drei Abstraktionsebenen

- **Internes Schema $\hat{=}$ physikalische Repräsentation (SSL)**
  - logische Datensätze, Zugriffspfade
  - Abbildung logischer Records auf Speicherstrukturen
- **konzeptuelles Schema $\hat{=}$ globale Sicht (DDL)**
  - logische Sicht des gesamten Datenbestandes, Integritätsbedingungen
  - keine Details über Datenstrukturen und Zugriffspfade
- **externe Schemata $\hat{=}$ einzelne Benutzersichten (SDDL)**
  - Gefilterte Ausschnitte des konzeptuellen Schemas, Autorisierungen
  - Anwendungsspezifische Neustrukturierung der Daten



Beispieldatenbank: konzeptionelles Schema

Heuer Saake Abb 2.2

Beispieldatenbank: externe Schemata

unstrukturierte Relation / hierarchische Relation

Heuer Saake Abb 2.3 + 2.4

Beispieldatenbank: internes Schema

Baumzugriffsstruktur und Hash-Tabelle

Heuer Saake Abb 2.5

System-Architektur: Strukturierung des DBMS

Heuer Saake Abb 2.7 Grobklassifizierung

Unabhängig von konkreter Datenbank

ANSI/SPARC Architektur

Detailliertere Version des Drei-Ebenen Modells

Fünf-Schichten Architektur der Datentransformation

Datenunabhängigkeit erfordert Trennung von Abstraktionsebenen

- **Jede Ebene beschreibt eine abstrakte Maschine**
- **Schichten werden standardisiert**
  - begrenzte Anzahl
  - optimale Bedinung der darüberliegenden Schicht
  - allgemeine, implementierungsunabhängige Funktionsbeschreibung
  - Fixierung von Schnittstellen, aber nicht der Komponenten
- **Höhere Systemebene ist Oberfläche für nächsttiefere Eben**
  - wird nur mit Mitteln der darunterliegenden Schicht realisiert
- **Strenge Trennung erleichtert Visualisierung**
  - tatsächliche strenge 5-Schichten Architektur hätte Performanzprobleme

Sicht auf logische Datenstruktur

Als Beispiel nur: Härder 2–9, 200%

Sicht auf Zugriffspfade

Härder 2–10, 130%

Sicht auf Speicherungsstruktur

Härder 2–11, 120%

Sicht auf Speicherzuordnungsstruktur

Härder 2–12, 120%

Schnittstellen zwischen verschiedenen Abstraktionsebenen

- **Mengenorientierte Schnittstelle (MOS)**
  - Relationen, Sichten, Tupel
  - deklarativer Zugriff
- **Satzorientierte Schnittstelle (SOS)**
  - logische Sätze und Zugriffspfade
  - navigierender Zugriff
- **Interne Satzschnittstelle (ISS)**
  - Sätze, Zugriffspfade, Bäume, Hashtabellen
  - Manipulation von Satzgruppen und Zugriffspfaden
- **Systempufferschnittstelle (SPS)**
  - Seiten, Segmente
  - freigeben, bereitstellen
- **Dateischnittstelle (DS)**
  - Blöcke, Dateien
  - holen, schreiben
- **Geräteschnittstelle (GS)**
  - Spuren, Zylinder
  - Bewegung von Geräteteilen

Fünf-Schichten Architektur: Funktionen

Heuer Saake 24 – 130% vergrößert

Fünf-Schichten Architektur: Operationen/Objekte

Heuer Saake 25 – 130% vergrößert

Komponenten eines DBMS

Vossen, S 30 90% mit Text aus Buchmann 2:16, Vossen §3

Konkrete Architekturen: IMS (hierarchisch)

Heuer Saake 28

Konkrete Architekturen: IMS – Struktur

Heuer Saake 29 – 170% vergrößert

Konkrete Architekturen: UDS (Netzwerk)

Heuer Saake 30 – 170% vergrößert

Konkrete Architekturen: Relationale Systeme

Heuer Saake 31 – 130% vergrößert

Architektur eines DBS aus Sicht der Anwender

- **Welche Benutzerkomponenten stellt ein DBMS bereit?**
- **Welche Schnittstellen bietet ein DBMS?**
- **Wie wird ein Anwendungsprogramm verarbeitet?**
- **Welche Arbeitsschritte sind erforderlich bei Erstellung und Ausführung eines Anwendungsprogramms?**

Benutzerkomponenten eines relationalen DBMS

Heuer Saake Abb 2.11

Benutzerkomponenten von DB2

Heuer Saake 33

Verarbeitung eines Anwendungsprogramms

Heuer Saake 34 – 150% vergrößert

Grundlagen der Datenbanken

Lektion 3

Informations und Datenmodelle I: Das Entity–Relationship Modell

1. Die Rolle von Datenmodellen beim Entwurf
2. Abstraktionskonzepte für DB-Schemata
3. Das Entity–Relationship Modell
   - Grundkonzepte und ihre Semantik
   - Kardinalität von Beziehungen
   - Spezielle Aspekte
   - Leitbeispiel: Universitätsdatenbank

Datenmodelle

Modellhafte Abbildung eines anwendungsspezifischen Ausschnitts der realen Welt

- **Statische Eigenschaften**
  - Objekte der zu modellierenden Welt (Entities: Daten+Ereignisse)
  - Beziehungen (Relationen) zwischen Objekten
  - Datentypen (Struktur) zur Beschreibung von Objekten und Beziehungen
- **Dynamische Eigenschaften**
  - Operationen (z.B. Zugriff, Speicherung, Änderung)
  - Beziehungen zwischen Operationen (z.B. Reihenfolge)
- **Integritätsbedingungen**
  - an Objekte und Operationen
  - sichern syntaktische und semantische Korektheit

Grundlegend für Entwicklung von Software

Programmiersprachen: Typsysteme, Klassenstrukturen
Expertensysteme: semantisch Netze, Formeln
Graphiksysteme: Repräsentationsmodelle
Datenbanken: abstrakte und konkrete Datenbankmodelle

Entwicklung von Datenmodellen

- **Beschreibung einer Miniwelt (Wirklichkeitsausschnitt)**
  - Gegenstände, Informationen, Zusammenhänge, Sachverhalte
  - Personen, Tatsache
  - Vorgänge und Veränderungen
- **Systemanalyse liefert Informationsmodell**
  - Diskrete Darstellung in "formaler" Sprache mit festen Regeln
  - Objekte, Attribute (Eigenschaften), Beziehungen
  - nur relevante, unterscheidbare und selektiv beschreibbare Informationen
- **Realisierungsentwurf liefert konkretes Datenmodell**
  - Typ- oder Klassenstruktur von Programmiersprachen
  - relationales, hierarchisches, Netzwerk-, objektorientiertes DB-Konzept
- **Implementierung mit einem konkreten DBMS**
  - Programmiersprache, Datenbanksprache, ...

Datenbankmodelle

- **Abstraktionskonzepte zur Beschreibung von DBS**
  - Syntax und Semantik von Datenbankschemata
- **Klassische Datenbankmodelle besonders geeignet für**
  - große Informationsmengen mit starrer Struktur
  - Darstellung statischer Eigenschaften und Integritätsbedingungen
- **Abstrakte Datenbankmodelle für Entwurf**
  - Entity-Relationship Modell (ER) und Erweiterungen (EER)
  - Semantische Datenmodelle (Sem DM)
  - Objektorientierte Datenbankmodelle (OODM,OMT)
- **Konkrete Datenbankmodelle für Realisierung**
  - Hierachisches Modell (HM), Netzwerkmodell (NWM)
  - Relationenmodell (RM), geschachtelte Relationen (NF^2: Non-First-Normal-Form)
  - Objektorientierte Programmiersprachen und -Datenmodelle (OODM)

Datenbankmodelle: Historische Einordnung und Bezüge

Heuer/Saake Folie 45

Grundlegende Abstraktionsschemata

- **Datentyp (Objekttyp)** $\mathsf{ID}(\mathsf{A}_1{:}\mathsf{D}_1,\ldots,\mathsf{A}_n{:}\mathsf{D}_n)$
  ID: Identifikator des Typs, $\mathsf{A}_i{:}\mathsf{D}_i$: Name und Wertebereich des i-ten Attributs
  - Beschreibung der Struktur von Objekten
  - aufgebaut durch elementare Wertebereiche (Domains) wie `int, string, ...`
  - oder zusammengesetzt durch Typkonstruktoren

- **Attribut A**
  - benannte 'Eigenschaft' eines Objekts
  - semantisch: Abbildung von Datentypen in Wertebereiche
  - Anwendung auf konkretes Objekt liefert Attributwert

- **Schlüssel**
  - Menge von Attributen, deren Werte ein Objekt eindeutig identifizieren

- **Objektklasse**
  extensional: Menge der Objekte des Objekttyps (Semantik des Typs)
  intensional: Struktur- und Verhaltensbeschreibung von 'gleichartigen' Objekten

- **Objektmenge**
  - extensional: Teilmenge einer Objektklasse
  - intensional charakterisiert durch Prädikat auf Attributen

Abgeleitete Typen und Attribute

- **Attribute sind Funktionen auf Objekten**
  - Üblicherweise Auswertung durch Zugriff auf gespeicherten Wert

  Abgeleitetes Attribut:
  - Attributwert wird aus gespeicherten Werten berechnet
  - z.B. Alter = Datum - Geburtsdatum
- **Typen sind abstrakte Strukturbeschreibungen**
  - Üblicherweise Beschreibung durch Komponenten

  Abgeleiteter Typ (Sichtdefinition)
  - Spezialisierung eines Typs durch Prädikat auf Attributen
  - z.B. TEENAGER ist PERSON mit Alter $<$ 20 AND Alter $>$ 12

Typkonstruktion

- **Klassifikation**
  - elementarer Konstruktor: Zuordnung eines Typs X zum Wert x
  - x Instanz von X (Ausprägung)
- **Aggregation (Tupelbildung)**
  - Bildung eines neuen Objekttyps aus Komponenten
  - auch als Beziehung PART-OF zwischen Objekten verwendbar
  - Erweiterung: Listen- und Multimengen
- **Assoziation (Mengenaggregation)**
  - Aufbau eines Objekttyps, dessen Werte endliche Mengen von Objekten eines anderen Typs sind
  - auch als Beziehung ELEMENT-OF zwischen Objekten verwendbar
  - Erweiterung: Mengenassoziation (Vereinigung, SUBSET-OF Beziehung)

Mögliche Beziehungen zwischen Typen

- **Vererbung:** Subklasse erbt Attribute einer Oberklasse
  - Zusätzliche Attribute und Operationen erlaubt
  - Wertebereich von Attributen kann eingeschränkt werden
  - Implementierung geerbter Attributen/Operationen darf sich ändern
  - Integritätsbedingungen müssen eingehalten werden
  - Konflikte bei Mehrfachvererbung möglich ($\mapsto$ Umbenennung)

- **Generalisierung:** $\mathbf{G \subseteq \bigcup S_i}$
  - Definition der allgemeineren Klasse G
  - beschreibe Gemeinsamkeiten von Subklassen, unterdrücke Unterschiede
  - Instanzen der Subklassen sind Instanzen der neuen Klasse
  - Instanzen der neuen Klasse können Instanzen mehrerer Subklassen sein

- **Spezialisierung:** $\mathbf{\bigcup S_i \subseteq G}$
  - Invers zur Generalisierung – unterstützt Top-Down Entwurf
  - Definition einer spezielleren Klasse S_i (IS_A Beziehung)
  - vollständig, falls $G=\bigcup S_i$ (sonst partiell)
  - disjunkt, falls $S_i \cap S_j = \emptyset$ für $i \neq j$ (sonst überlappend)

- **Partitionierung:** disjunkte Spezialisierung $\mathbf{\dot{\bigcup} S_i = G}$

Semantikfestlegung für Datenbankmodelle (Prinzipien)

- **μ(D): Trägermenge für mögliche Werte des Domains D**
  - elementare Wertebereiche: $\mu(\text{int})=\mathbb{Z}$, $\mu(\text{string})=\{\text{a,..,z,A,..Z,0,..9,..}\}^*$, ...
  - Typkonstruktoren: $\mu(\mathsf{prod}(\text{D}_1,..,\text{D}_n)) = \mu(\text{D}_1)\times..\mu(\text{D}_n)$, $\mu(\mathsf{set}(\text{D})) = 2^{\mu(\text{D})}$, ...
- **Datenbankzustand (state) σ**
  - Beschreibung der DB-Einträge durch Werte aus $\mu(\mathtt{typ}(DB))$
  - $\mathtt{typ}(DB)$: fiktiver Gesamttyp aller Datenbankvariablen (mengenwertig)
  - Angabe relativ zu einem Zeitpunkt t $\in \mathcal{T}$
- **Semantik der Datenbank**
  - Menge möglicher Datenbankzustände als Funktion $\sigma(DB)$: $\mathcal{T} \rightarrow \mu(\mathtt{typ}(DB))$

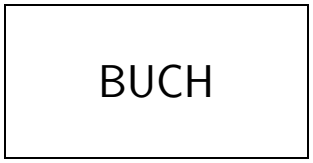
z.B. $\sigma(\mathsf{BÜCHER})(42) = \{\mathtt{(Heuer, OODB, 1\text{-}453\text{-}, 1992), (SAAKE, OOSIS, 1\text{-}321\text{-}, 1993)}\}$

Zur Vereinfachung wird die Zeit im folgenden ignoriert

Entity-Relationship Modell (Chen, 1976)

Graphisches Modell zur Darstellung eines Weltausschnitts

- **Entity:** Objekt/Konzept der modellierten Wirklichkeit
  - z.B. VORLESUNG, BUCH, PROFESSOR, auch PRÜFUNG
  - dargestellt als Rechteck



- **Relationship:** Beziehung zwischen Entities
  - z.B. Professor LIEST Vorlesung
  - dargestellt als Raute

- **Attribut:** Eigenschaft von Entities oder Beziehungen
  - z.B. ISBN eines Buchs, Semester der gelesenen Vorlesung
  - dargestellt als Oval/abgerundetes Rechteck

- **Wertebereich:** zulässige Werte für Attribute
  - z.B. string für Namen
  - dargestellt im Attribut Name:string (oder gar nicht)
- **Schlüssel (Key):** Attribute, deren Wert ein Entity identifiziert
  - z.B. ISBN eines Buchs, Name, Fach für Professoren
  - dargestellt durch Unterstreichung im Attribut

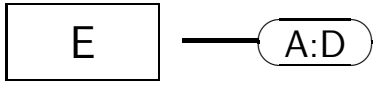
ER-Modell für Vorlesungsdatenbank

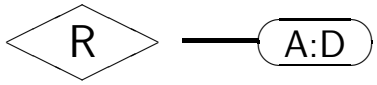
HeuerSaake Folie 51 (einfärben)

ER-Konzepte: Entity-, Relationship- und Wertemengen

- **Wertemengen $\mathbf{D_k}$**
  - primitive Datentypen int, string, ... (Standard ERM ohne Typkonstruktoren)
  - Semantik: $\mu(\mathrm{D})$ – Menge aller möglichen Werte
- **Entity-Typen $\mathbf{E_i}$**
  - Einteilung der zu repräsentierende Informationseinheiten des DBS
  - Schema enthält endlich viele Entity-Mengen (nicht notwendig disjunkt)
  - Semantik: $\mu(\mathrm{E})$: unendliche Menge möglicher Werte (festgelegt durch Attribute)
    $\sigma(\mathrm{E})$: endliche Menge aktueller Entities vom Typ E
- **Beziehungstypen $\mathbf{R(E_1,..,E_n)}$**
  - Typ gleichartiger Beziehungen zwischen gleichen Entity-Mengen
  - R verbindet n$\geq$2 Entity-Typen $\mathrm{E}_1,..,\mathrm{E}_n$ (R hat Grad n)
    binäre Beziehung sind häufigster Fall, n$\geq$4 sehr selten
  - Rollennamen nötig, falls $\mathrm{E}_i=\mathrm{E}_j$: verheiratet(Frau:PERSON, Mann:PERSON)
  - Semantik: $\mu(\mathrm{R}) = \mu(\mathrm{E}_1)\times\ldots\times\mu(\mathrm{E}_n)$
    $\sigma(\mathrm{R}) \subseteq \sigma(\mathrm{E}_1)\times\ldots\times\sigma(\mathrm{E}_n)$
    (aktuelle Beziehungen nur zwischen aktuellen Entities!)

ER-Konzepte: Attribute und Schlüssel

- **Attributdeklaration** 

und 

  - Deklaration von Eigenschaften eines Entity-Typs E
  - Semantik: $\sigma(\mathrm{A})$: $\sigma(\mathrm{E}) \rightarrow \mu(\mathrm{D})$, (D beschränkt auf Wertemengen)
    $\sigma(\mathrm{A})$: $\sigma(\mathrm{R}) \rightarrow \mu(\mathrm{D})$ bei Beziehungsattributen
  - Notation: $\mathrm{E}(\mathrm{A}_1{:}\mathrm{D}_1, \ldots, \mathrm{A}_m{:}\mathrm{D}_m)$ bzw. $\mathrm{R}(\mathrm{E}_1, .., \mathrm{E}_n;\ \mathrm{A}_1{:}\mathrm{D}_1, \ldots, \mathrm{A}_m{:}\mathrm{D}_m)$

- **Schlüsselattribute $\mathrm{S}_1, .., \mathrm{S}_k$ für Entity-Typ E**
  - Menge von Attributen, deren Wert ein Entity eindeutig identifiziert
  - Semantik: $\forall \mathrm{e}_1, \mathrm{e}_2 : \sigma(\mathrm{E}).\ \sigma(\mathrm{S}_1)(\mathrm{e}_1) = \sigma(\mathrm{S}_1)(\mathrm{e}_2) \wedge \ldots \wedge \sigma(\mathrm{S}_k)(\mathrm{e}_1) = \sigma(\mathrm{S}_k)(\mathrm{e}_2) \Rightarrow \mathrm{e}_1 = \mathrm{e}_2$
  - $\mathrm{S}_1, .., \mathrm{S}_k$ muß minimal sein (jede echte Teilmenge ist kein Schlüsselkandidat)
  - bei mehreren Schlüsselkandidaten wähle Primärschlüssel
  - Notation: $\mathrm{E}(\ldots, \underline{\mathrm{S}_1}, \ldots, \underline{\mathrm{S}_k}, \ldots)$

Zweistellige vs. mehrstellige Beziehungen

HeuerSaake Folie 57 (einfärben)

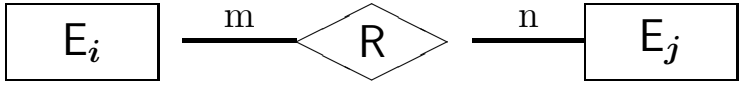
Ternäre Beziehung nicht automatisch umwandelbar

HeuerSaake Folie 58 (einfärben)

Strukturelle Integritätsbedingungen

Wieviele Instanzen nehmen an einer Beziehung teil?

- **Kardinalität**



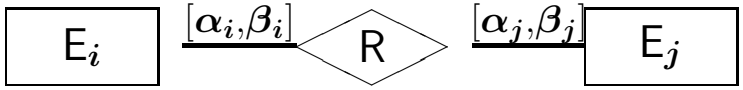
1:1 – für jedes Entity vom Typ E_i gibt es höchstens eines vom Typ E_j

1:n – für jedes Entity vom Typ E_i gibt es (evtl.) mehrere vom Typ E_j

m:n – für jedes Entity vom Typ E_i gibt es mehrere vom Typ E_j und umgekehrt

i.a. zu ungenau

- **Komplexität** $R(E_1,..,E_i[\alpha_i,\beta_i],..,E_n)$

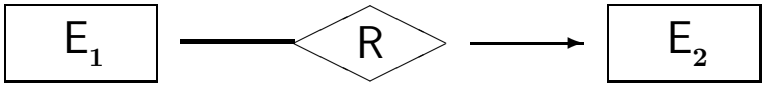


  - Ein Entity aus E_i kommt in mindestens α_i und höchstens β_i Beziehungen vor
  - Semantik: $\forall i \leq n.\, \forall e_i : \sigma(E_i).\ \alpha_i \leq |\{\, r \in R \mid r.E_i = e_i \,\}| \leq \beta_i$
  - [0,*] (beliebig viele Teilnahmen) ist Standardannahme
  - $R(E_1,..,E_i[0,1],..,E_j[0,*],..,E_n)$ entspricht Kardinalität n:1

Statt Komplexität sagt man zuweilen ebenfalls Kardinalität

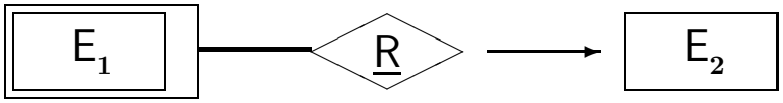
Spezielle Aspekte

- **Funktionale Beziehung**



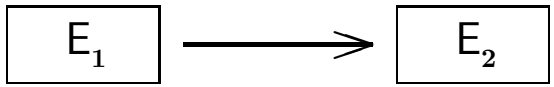
  - Komplexität R(E_1[0,1],E_2): partielle Funktion $\sigma(R):\sigma(E_1) \not\rightarrow \sigma(E_2)$
  - Komplexität R(E_1[1,1],E_2): totale Funktion $\sigma(R):\sigma(E_1) \rightarrow \sigma(E_2)$
  - besonders geeignet zum Navigieren in einer Datenbank

- **Abhängige Entity-Typen**



  - Funktionale Relation R ist ein Schlüssel'attribut' für Entity-Typ E_1
  - Entities aus E_1 bestimmbar durch Werte aus R (und weitere Schlüsselattribute)
  - E_1 ist **schwacher Entity-Typ**

- **IS_A-Beziehung**

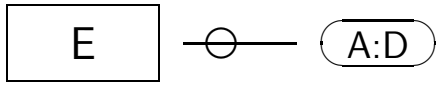


  - Funktionale (injektive) Relation der Komplexität IS_A(E_1[1,1],E_2[0,1])
  - σ(IS_A) festgelegt als *identische* Abbildung

  $\Rightarrow$ E_1 spezieller abhängiger Typ: IS_A ist eindeutiger Schlüssel

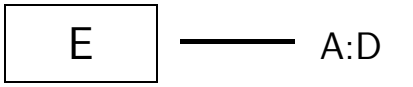
  $\Rightarrow$ Spezialisierung $\sigma(E_1) \subseteq \sigma(E_2)$: E_1 erbt Attribute und Instanzen von E_2

- **Optionale Attribute**



  - Attribut muß nicht für jedes Entity einen Wert annehmen

- **Abgeleitete Attributwerte**



  - Attributwert wird berechnet und nicht direkt gespeichert

Universitätsdatenbank – ER-Modellierung

HeuerSaake A1, Folie 175

Universitätsdatenbank – Attribute der Entity-Typen

HeuerSaake A1, Folie 176

Universitätsdatenbank – Spezifikation des Typs Person

HeuerSaake Folie 177 A1

Grundlagen der Datenbanken

Lektion 4

Informations- und Datenmodelle II: Modellierungsalternativen

1. Erweiterungen des Standard-ERM
   - Nicht-Standard Datentypen
   - Modifiziertes Schlüsselkonzept
   - Beziehungen zwischen Entity-Typen
2. Das OMT Objektmodell
   - Klassen, Objekte und Methoden
   - Beziehungen zwischen Objektklassen
   - Diagrammtechniken

Erweiterungen des Standard Entity-Relationship Modells

- **Unterstützung strukturierter Attributwerte**
  - Konstruktoren für mengen- und tupelwertige Attribute
- **Komplexe Entity-Typen**
  - Aggregation: Entity zusammengesetzt aus Instanzen anderer Typen
  - Assoziation/Sammlung: Entity als Menge von Instanzen eines Typs
- **Erweitertes Schlüsselkonzept**
  - veränderte Notation
  - erlaubt Verzicht auf abhängige Entity-Typen
- **Vererbungsbeziehungen zwischen Entity-Typen**
  - Generalisierung: allgemeinerer Kontext für Entities
  - Spezialisierung: Ersatz für die IS-A-Beziehung
  - Partitionierung: Zerlegung eines Entity-Typs in diskjunkte Typen
  - modelliert durch allgemeinen Typkonstruktor
- **Beziehungen höheren Typs** ($\mapsto$ hierarchisches ERM)
  - Generalisierung und Spezialisierung für Relationentypen
  - Beziehungen zwischen Instanzen von Relationen

Anpassung an Konzepte des objektorientierten Programmierens

EER-Modell: Nicht-Standard Datentypen

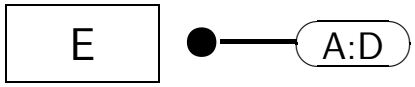
- **Komplexe Wertemengen D_k**
  - primitive Datentypen int, string, ...
  - \+ Konstruktoren prod, list, set, bag mit fester Semantik
  - Attributdeklaration Adresse: prod(string,int,string)
- **Benutzerdefinierte Datentypen und Operationen**
  - Deklaration point = prod(real,real)
  - Semantik: $\mu(\text{point}) = \mu(\text{real}) \times \mu(\text{real}) = \mathbb{R} \times \mathbb{R}$
  - Spezifikation von Operationen durch Gleichungen
    dist((x,y),(x',y')) = $(x-x')^2+(y-y')^2$
- **Objektwertige Attribute möglich**
  - Deklaration E(...,A_i:E_i,...): Attribut A_i hat Werte vom Entity-Typ E_i
  - Semantik: $\sigma(A_i):\sigma(E) \rightarrow \sigma(E_i)$ — funktionale Beziehung zwischen E und E_i
  - Darstellung von Aggregation und Assoziation durch Konstruktoren
    z.B. durch Deklaration Autoren: list(PERSON)

Aggregation und Assoziation durch objektwertige Attribute

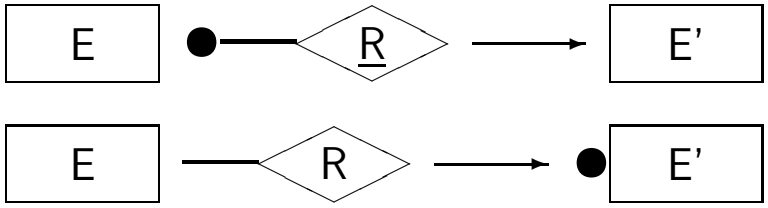
HeuerSaake Folie 86

Erweitertes Schlüsselkonzept

- **Schlüssel definiert über Attribute**



  oder funktionale Beziehungen



  - nur Primärschlüssel werden gekennzeichnet
  - veränderte Notation (• am Entity-Typ) erforderlich

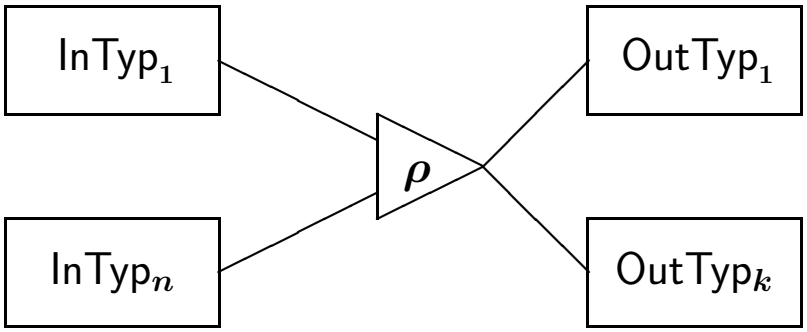
- **Objektwertige Attribute als Schlüssel möglich**
  - in E(. . . ,A_i:op(..E_i..),. . .) kann A_i Schlüssel für E und für E_i sein

⇒ Simulation abhängiger Entity-Typen möglich

HeuerSaake Folie 87b

Allgemeiner Typkonstruktor

Ein Konzept für Generalisierung, Spezialisierung, Partitionierung



$\boldsymbol{\rho}$ ist '=' oder $\supseteq$

- **Semantik**
  - $\bigcup\sigma(\mathsf{InTyp}_i)\ \rho\ \bigcup\sigma(\mathsf{OutTyp}_j)$: Ausgabetypen Spezialisierung der Eingabetypen
  - $i \neq j \Rightarrow \mathsf{OutTyp}_i \cap \mathsf{OutTyp}_j = \emptyset$: Ausgabetypen sind disjunkt
  - Schlüssel nur für Eingabetypen erlaubt
  - **Keine zyklischen Graphen von Typkonstruktoren**
- **Spezialfälle**
  - **Spezialisierung**: n=k=1: $\sigma(\mathsf{OutTyp}) \subseteq \sigma(\mathsf{InTyp})$
  - **Generalisierung**: n>k=1: $\sigma(\mathsf{OutTyp}) \subseteq \bigcup\sigma(\mathsf{InTyp}_i)$
  - **Partitionierung**: 1=n<k: $\bigcup\sigma(\mathsf{OutTyp}_i) \subseteq \sigma(\mathsf{InTyp})$
- **EER unterstützt nur die Spezialfälle**
  - Allgemeiner Typkonstrukter zu komplex für Modellierungen

Mehrfachspezialisierung

HeuerSaake Folie 85

Eingabetypen müssen (indirekt) aus gleicher Ausgangsklasse stammen

Generalisierung / Spezialisierung

HeuerSaake Folie 79

HeuerSaake Folie 80b

Partitionierung $\neq$ mehrfache Spezialisierung

HeuerSaake Folie 82

Disjunktheitsbedingung nur bei Partitionierung

Partitionierung vs. Generalisierung

HeuerSaake Folie 83

Partitionierung: manche Dokumente sind weder Bücher noch Zeitschriften
alle Bücher sind Dokumente
Generalisierung: alle Dokumente sind Bücher oder Zeitschriften
manche Bücher sind keine Dokumente

Universitätsdatenbank – EER-Modellierung

HeuerSaake A1, Folie 178

Begriffe des ER-Modells und des EER-Modells

HeuerSaake buch Abb. 3.2, 115%

OMT: Objekt Modelling Technique (Rumbaugh, 1991)

Diagrammtechnik zur Beschreibung von Softwareentwürfen
System wird in 3 Stufen modelliert

- **Objektmodell**
  - statische Strukturen und Daten
  - Objektklassen, Attribute, Beziehungen, Operationen und Methoden

  Darstellung graphisch durch Objektdiagramme (ähnlich zum ERM)
- **Dynamisches Modell**
  - zeitliche, Verhaltens- und Kontrollaspekte
  - Sequenz der Operationen (Events) und Kontext für Events (Zustände)

  Darstellung graphisch durch Zustandsdiagramme
- **Funktionales Modell**
  - Zustandsveränderungen, Wertveränderungen und Abbildungen

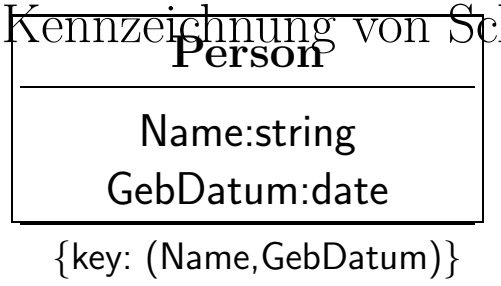
  Darstellung graphisch durch Flußdiagramme

Objektmodell entspricht abstraktem Datenbankmodell

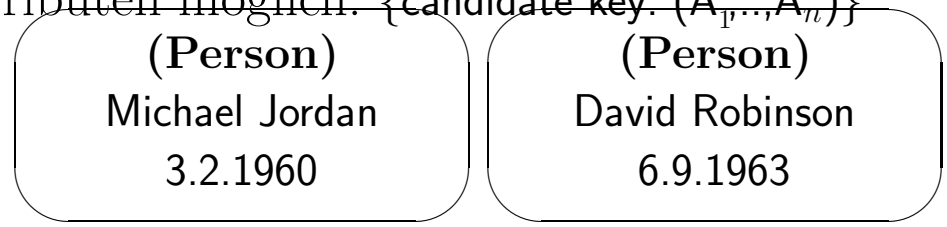
Objektdiagramme: Klassen und Objekte

Graphische Notation für Objekte, Klassen und Beziehungen

- **Klassendiagramm**
  - Beschreibung der Beziehungen zwischen Objektklassen
  - korrespondiert zu unendlicher Menge von Instanzendiagrammen
- **Instanzendiagramm**
  - Beschreibung der Beziehungen zwischen konkreten Objekten
  - Darstellung von Testfällen, Szenarien, Beispielen
- **Attribute dargestellt als Teil einer Objektklasse**
  - korrespondieren zu konkreten Werten in einer Instanz
  - optionale Angabe von Typ und Defaultwert
  - sollen Attributen der realen Welt entsprechen (keine Navigationsdaten)
  - Kennzeichnung von Schlüsselattributen möglich: {candidate key: (A_1,..,A_n)}



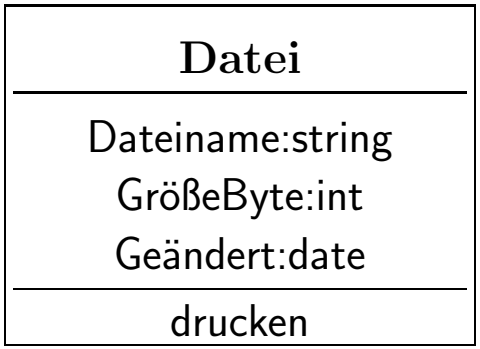
Klasse mit Attributen

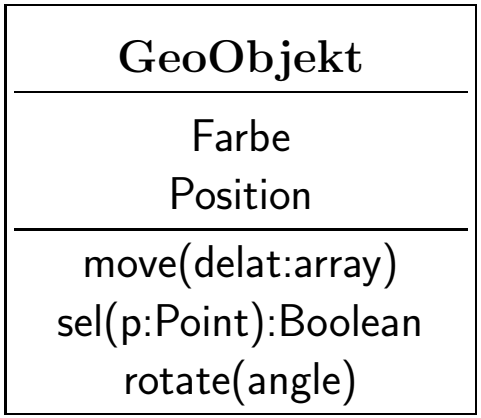


Instanzen mit Werten

Objektdiagramme: Operationen und Methoden

- **Funktion/Transformation auf Objekten einer Klasse**
  - Aktuelles Objekt der Klasse als implizites Argument (Zielobjekt)
  - zusätzliche Argumente möglich
  - dargestellt als dritter Teil einer Objektklasse



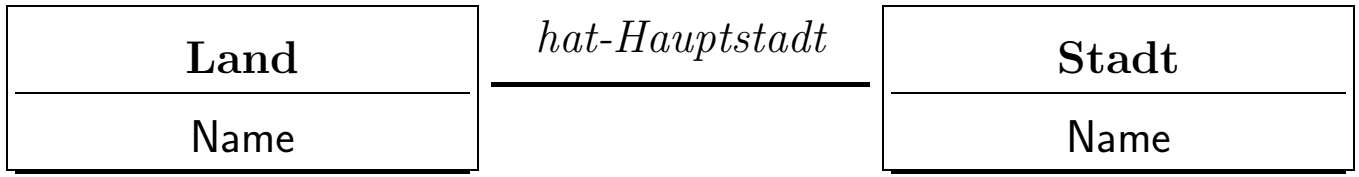


- **Polymorphismus**
  - Operationen durch Vererbung auf mehrere Klassen anwendbar
  - verschiedene Implementierungen in Subklassen (Methoden) möglich
    dieselbe Operation ist auf verschiedene Weisen ausführbar
  - Signatur und Integritätsbedingungen müssen erhalten bleiben
  - dynamisches Binden: Klasse des Zielobjekts bestimmt angewandte Methode

- **Operationen mit und ohne Nebeneffekte**
  - Funktionen/Queries = Operationen ohne Nebeneffekte
  - Abgeleitete Attribute = Queries ohne Parameter

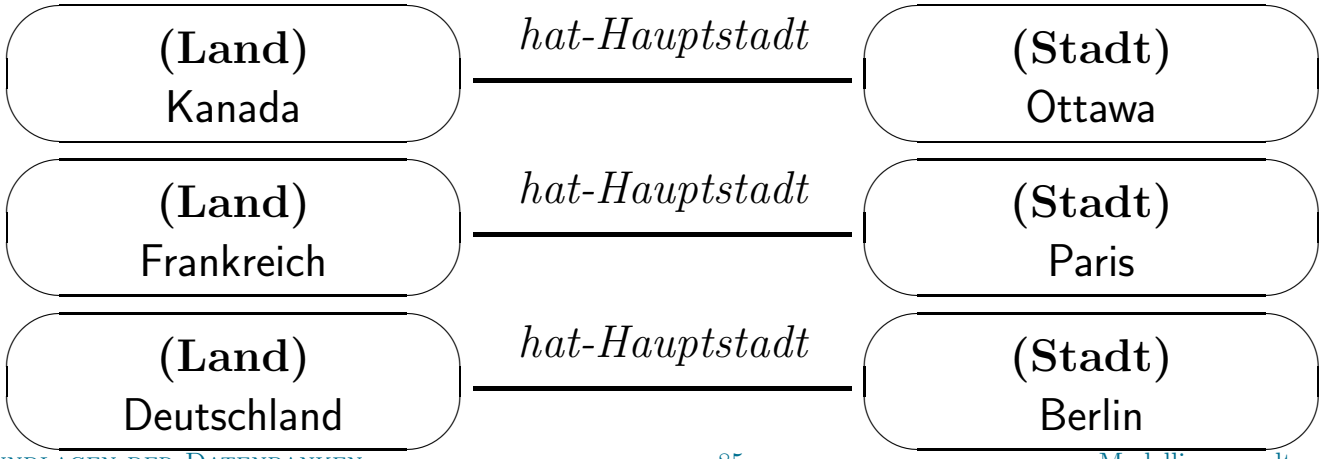
Objektdiagramme: Relationen

- **Beziehung: Relation zwischen Objektklasse**
  - entspricht dem Begriff 'Relationship set'
  - orig. 'Association' (nicht verwechseln mit Assoziation = Mengenaggregation)
  - meist bidirektional – Name gibt Semantik der Vorwärts-Richtung
  - Graphische Repräsentation als Linie mit Namen



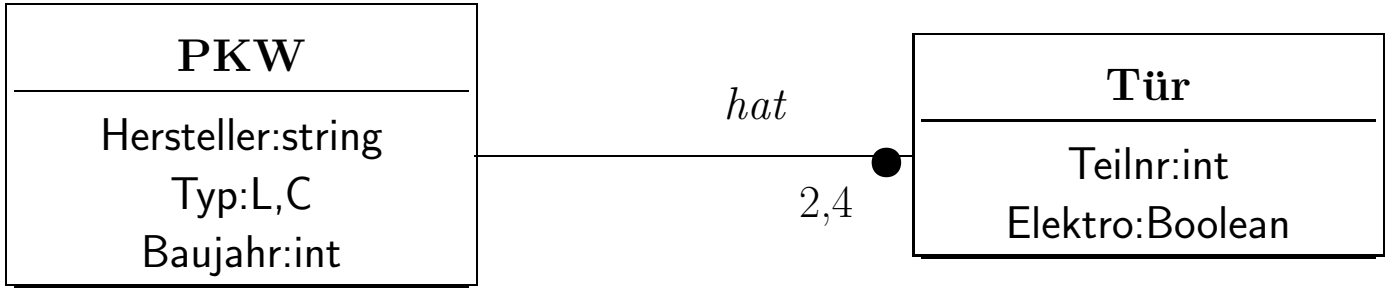
  - Bei 3- und mehrstelligen Beziehungen Raute wie im ERM

- **Link: Relation zwischen Objektinstanzen**
  - entspricht dem Begriff 'Relationship'



Anzahl von Objekten einer Klasse, die mit demselben Objekt in Beziehung stehen

- **Deklaration als Zahl, Intervall oder Aufzählung**
- **Graphische Darstellung zusätzlich mit Punkten**
  - schwarzer Punkt ●: 'mehrere' – 0 oder mehr Teilnehmer
  - weißer Punkt ○: 'optional' – 0 oder 1 Teilnehmer
  - kein Punkt: 'eins' – 1:1 Beziehung

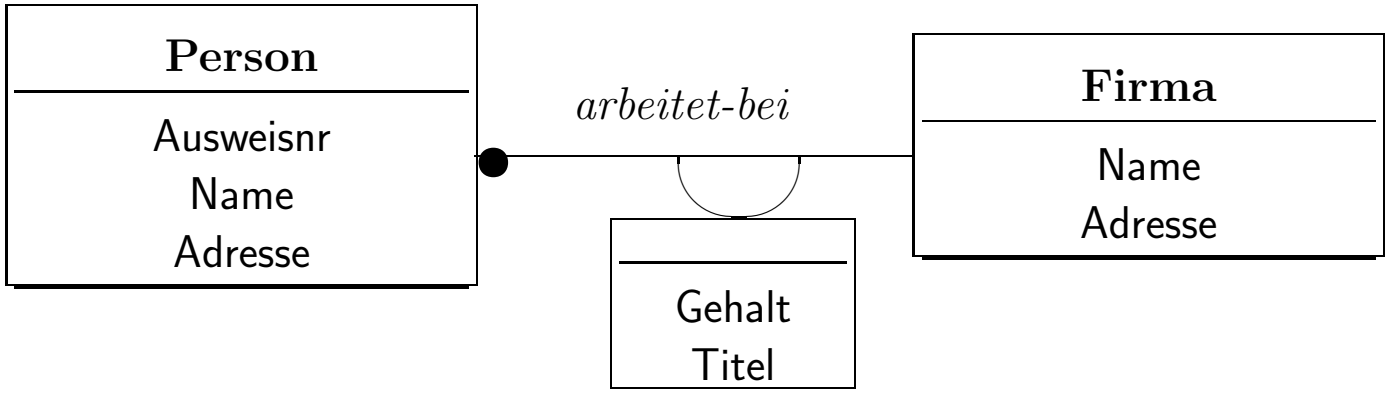


ein normaler PKW hat 2 oder 4 Türen

Achtung: andere Semantik als Komplexitäten im ERM

Attribute von Beziehungen

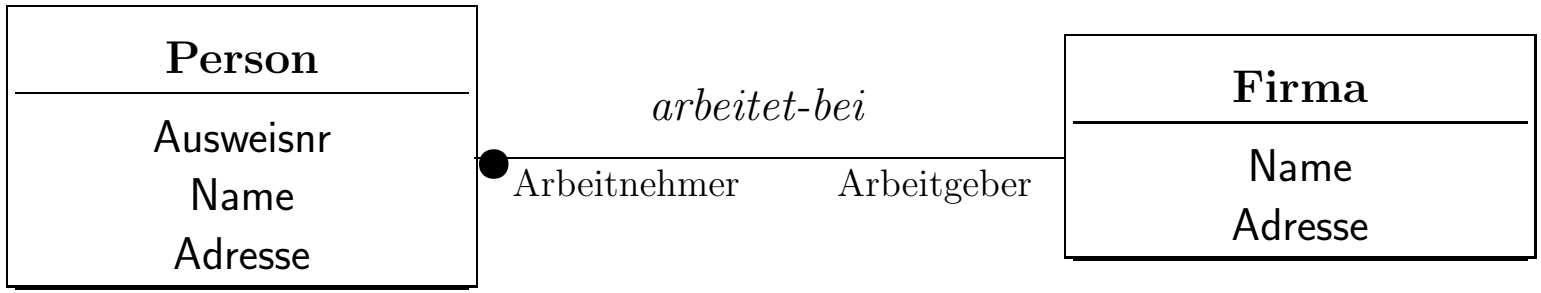
- **Beziehungen können eigene Attribute haben**
  - dargestellt als Box, die durch einen Halbkreis verbunden ist



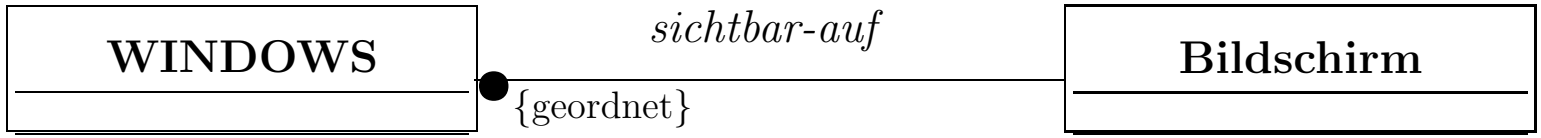
- Attribute, die von Beziehungen zwischen 2 Klassen abhängen, sollen als Beziehungsattribute, nicht als Klassenattribute modelliert werden
  - Gehalt, Titel gehört zum Arbeitsverhältnis, nicht zur Person
  - wichtig bei m:n Beziehungen, sonst integrierbar in Klasse

Rollen und Ordnung in Beziehungen

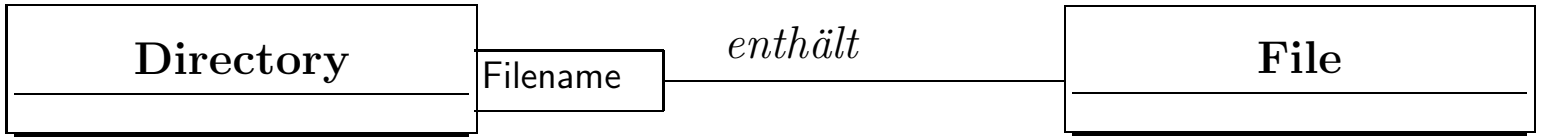
- **Rollenname:** Bezeichnung für eine Komponente einer Beziehung
  - eindeutige Klassifizierung beteiligter Objekte
  - graphisch am jeweiligen Ende notiert
  - besonders wichtig bei Beziehungen zwischen Objekten derselben Klasse



- **Ordnung**
  - Kennzeichnung, daß die Instanzen in der Beziehungen geordnet sind
  - z.B. Reihenfolge der Fenster auf einem Bildschirm im Window-Manager



Reduziere effektive Multiplizität einer Beziehung



Attribut Filename **qualifiziert Beziehung** *enthält*

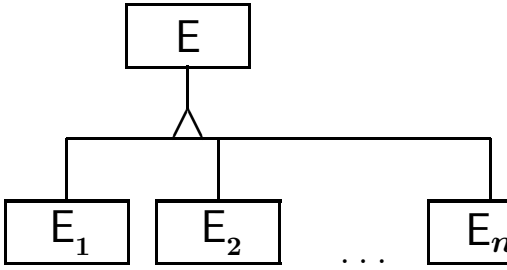
- Filename ist **Qualifikator** für **Directory**
- Klasse **Directory** wird implizit um Qualifikator-Attribut erweitert
- Anzahl der **File**-Objekte, die mit einem Objekt der 'erweiterten' Klasse in Beziehung stehen, sinkt (manchmal sogar auf 1)
- Informationsgehalt der Beziehung *enthält* steigt (1:1–Beziehung zwischen **Directory**+Filename und **File**)

Nur möglich bei 1:n und m:n-Beziehungen

Generalisierung

Abstraktionsmechanismus zur Beschreibung von Ähnlichkeiten

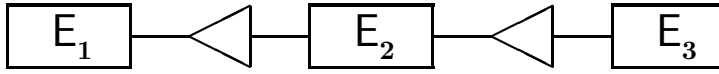
- **Beziehung mit IS_A Semantik**
  - Graphisch dargestellt durch Dreieck in Beziehungslinie
  - E generalisiert E_i (E_i spezialisiert E)
  - Elemente von E_i sind Elemente von E
  - Unterklasse E_i erbt Attribute und Operationen von Oberklasse E
  - Unterklassen dürfen Attribute und Operationen einschränken (Restriktion)
  - Unterklassen dürfen neue Attribute einführen (Erweiterung)



- **Transitive Beziehung**
  - Kurze Generalisierungshierarchien empfehlenswert

  $\Rightarrow$ Entwurf wird übersichtlicher

  $\Rightarrow$ Performanz der Implementierung (Vererbung) besser

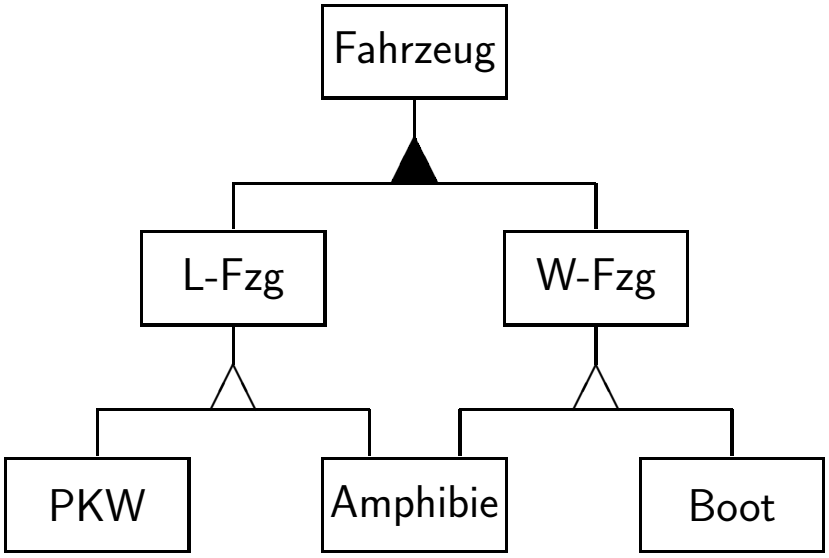


Reduziert Anzahl unabhängiger Konzepte
Erhöht Wiederverwendung von Programmcode

Generalisierungshierarchie in OMT

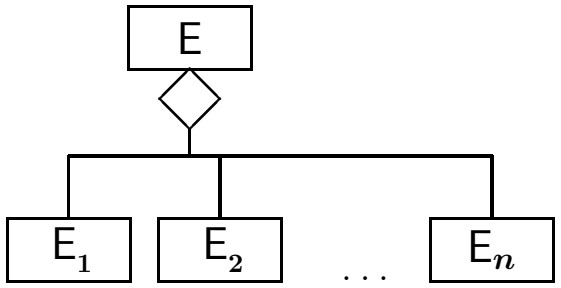
OMT Bild 3.24 Seite 41 150%

Mehrfach-Generalisierung

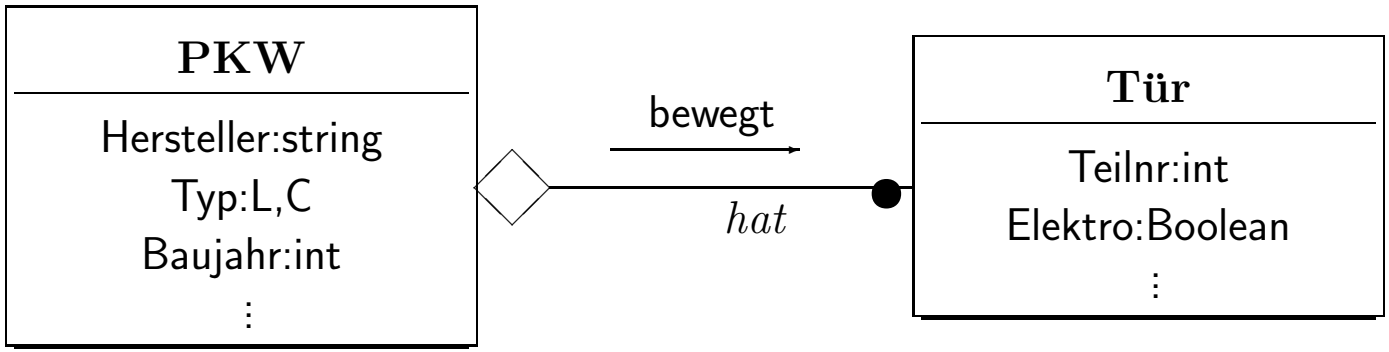


- **Klasse erbt von mehreren Oberklassen**
  - Subklasse ist Join-Klasse (Summe aller Features)
  - Identische Features eines Ahnen werden nur einmal geerbt
  - Mehrdeutigkeiten und Konflikte explizit auflösen (!)
- **Überlappende Join-Klassen** (gemeinsame Objekte)
  - Dargestellt durch schwarzes Dreieick ▲
- **Disjunkte Join-Klassen** (keine gemeinsamen Objekte)
  - Dargestellt durch leeres Dreieick △

Aggregation

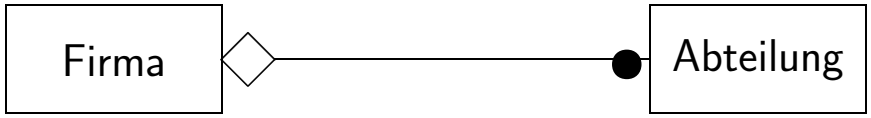


- **Beziehung mit PART-OF Semantik**
  - E_i sind Komponenten der Aggregatsklasse E
  - Graphisch dargestellt durch Raute am Aggregatsende der Beziehung
- **Transitive und antisymmetrische Beziehung**
  - Formal: Beziehung zwischen einem einzelnen E_i und E
- **Propagation von Features auf Komponenten möglich**
  - Autoteile bewegen sich, wenn Auto sich bewegt

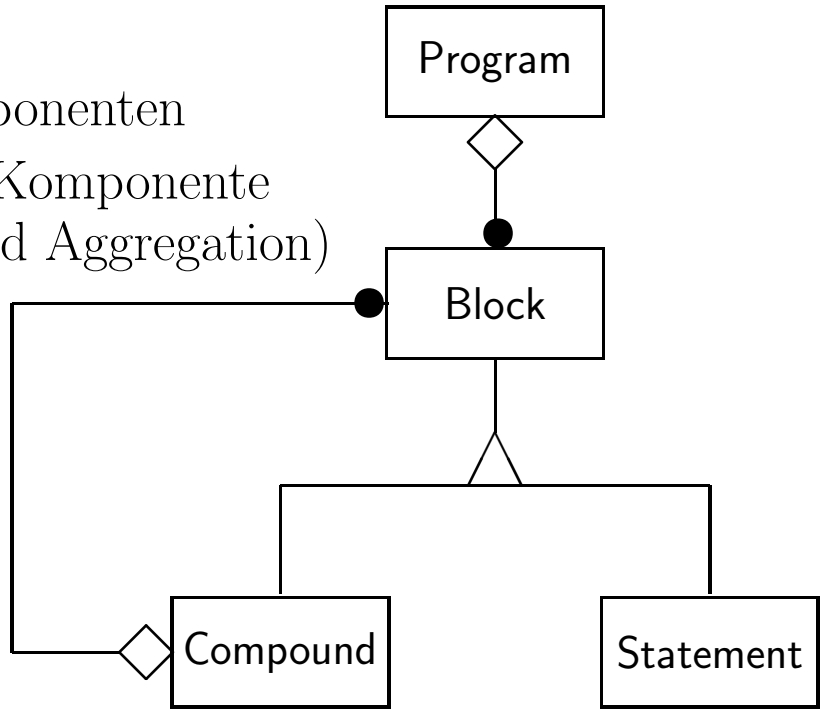


Klassifizierung von Aggregatsklassen

- **Feste Aggregatsklasse**
  - Anzahl und Typ der Komponenten eines Objektes fixiert
    (eine Lampe besteht aus einem Fuß, einem Schirm, einem Schalter, ...)
- **Variable Aggregatsklasse**
  - Objekte haben verschiedene Anzahlen von Komponenten



- **Rekursive Aggregatsklasse**
  - Objekte der gleichen Klasse sind Komponenten
  - Terminierung durch Unterklasse einer Komponente
    (Kombination von Generalisierung und Aggregation)



Aggregation vs. Generalisierung

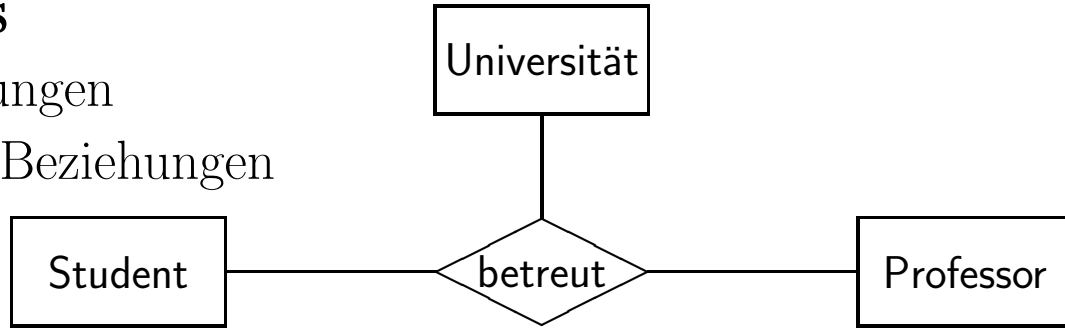
OMT Bild 4.2 Seite 59 200%

Aggregation ist UND-Beziehung

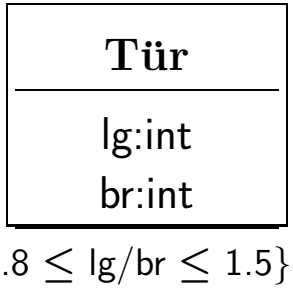
Generalisierung ist ODER-Beziehung

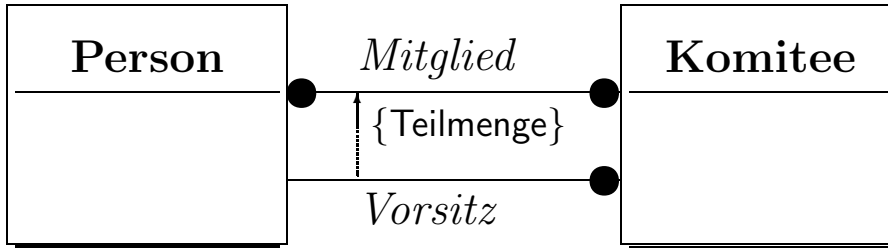
Einschränkungen an Objekte und Beziehungen

- **Strukturelle Constraints**
  - Schlüssel für Klassen und Beziehungen
  - Multiplizitäten und Ordnung für Beziehungen



- **Logische Constraints**
  - Einschränkung der Werte von Attributen
  - auch zwischen zwei Objekten
  - auch zwischen zwei Beziehungen
  - Gleiche Notation {constraint}, formuliert in Formeln oder natürlicher Sprache





Entwurfshilfen für komplexe Diagramme

- **Module**
  - Logische Strukturierung eines Datenmodells in überschaubare Teile
  - Gruppierung von Klassen und Beziehungen zu größerer Einheit
  - Bezeichner innerhalb eines Moduls müssen verschieden sein
  - Kennzeichnung nur durch Modulnamen
  - Empfehlung: Anzahl der Beziehungen zwischen Modulen klein halten
- **Sheets**
  - Aufteilung von Modulen auf druckbare Seiten
  - Kennzeichnung mit Namen
  - nur Notationsvereinfachung – kein logisches Konstrukt

Beispiel: Modellierung eines Window-Systems

OMT-Buch Abb. 3.25, Seite 44 100%

+ Nachträgliche Verbesserung des Entwurfs $\Rightarrow$ *klaerere Strunktur*

1. Beschreibe Linien und Ellipsen mit Punkten statt Koordinaten

2. verbinde Linie und Punkt durch Aggregation defined-by

3. verbinde Ellipse und Punkt mit has-center

OMT Objektmodell vs. Entity-Relationship Modell

OMT-Buch Abb.12.1 / 12.2, Seite 272 110% (mit schneiden)

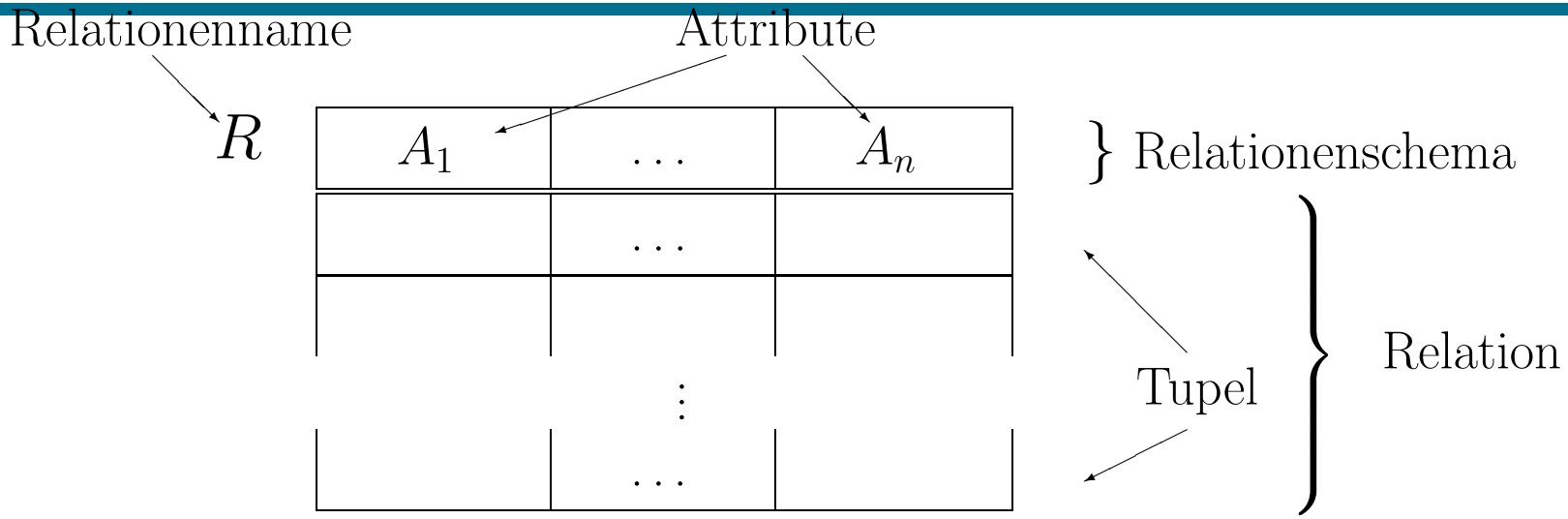
Grundlagen der Datenbanken

Lektion 5

Grundlagen des Relationalen Datenmodells

1. Grundlegende Konzepte
2. Darstellung von ER-Modellen
3. Relationenkalküle – Relationenalgebra

Das Relationenmodell (Codd, 1970)



- **Datenbanken bestehen auschließlich aus Relationen**
  - Datenbankschemata sind Menge von Relationenschemata
  - Relationenschemata sind Mengen von Attributen
  - Attribute gehören zu Wertebereichen (Standard-Datentypen bei Normalform)
  - Relation ist Teilmenge des Produkts der Wertebereiche
  - Relationen bestehen aus Tupeln
  - Datenbankzustand beschreibbar durch Menge der aktuellen Relationen
- **Grundregeln**
  - Reihenfolge der Zeilen und Spalten ohne Bedeutung
  - Jedes Tupel ist eindeutig (keine Wiederholungen)
  - Es existieren Primärschlüssel

Relationen zur Darstellung von Personen

Heuer/Saake Folie 94 110%

Grundlegende Konzepte

Gegeben sei ein Universum $\boldsymbol{\mathcal{U}}$ von Namen, eine Menge $\boldsymbol{\mathcal{D}} = \{D_1, ..D_m\}$ von endlichen, nichtleerer Mengen und eine Domänenfunktion $\mathbf{dom}: \mathcal{U} \rightarrow \mathcal{D}$

- **Attribut**: Element A von $\mathcal{U}$
- **Wertebereich**: Element D von $\mathcal{D}$
- **Attributwert für A**: Element w von $\mathrm{dom}(A)$ (Wertebereich von A)
- **Relationenschemata**: Teilmenge $R = \{A_1, ..A_n\}$ von $\mathcal{U}$
- **Tupel über R**: Abbildung $t: R \rightarrow \bigcup_{i=1..m} D_i$ mit $\forall A \in R.t(A) \in \mathrm{dom}(A)$
  homomorph fortgesetzt auf Teilmengen von R
- **Relation über R**: endliche Menge r von Tupeln über R – $\boldsymbol{r \in \mathbf{REL}(R)}$
- **Datenbankschema**: Menge $S = \{R_1, .., R_j\}$ von Relationenschemata
- **Datenbank über S**: Menge $d = \{r_1, .., r_j\}$ mit $r_i \in \mathrm{REL}(R_i)$ – $\boldsymbol{d \in \mathbf{DB}(S)}$
- **Basisrelation**: Element r von $d \in \mathrm{DB}(S)$

Achtung: Tupel in ungeordneter Mengendarstellung

Schlüssel und lokale Integritätsbedingungen

- **Identifizierende Attributmenge für R**: Menge $K = \{B_1, ..B_k\} \subseteq R$ mit
  $\forall t_1, t_2 \in r.(\forall B \in K.t_1(B) = t_2(B)) \Rightarrow t_1 = t_2$ für alle $r \in \mathrm{REL}(R)$
- **Schlüssel**: minimale identifizierende Attributmenge

  Primärschlüssel: ausgezeichneter Schlüssel
- **Lokale Integritätsbedingung für R**:
  - Menge $\mathcal{B}$ von Abbildungen b:REL$(R) \rightarrow$`bool`
  - Schlüssel sind lokale Integritätsbedingungen
- **Erweitertes Relationenschema**: $\mathcal{R} = (R, \mathcal{B})$
  - R Relationenschemata, $\mathcal{B}$ lokale Integritätsbedingung für R
- **Relation über $\mathcal{R} = (R, \mathcal{B})$**:
  - Relation $r \in \mathbf{REL}(R)$ mit $b(r)$=`true` für alle $b \in \mathcal{B}$
  - $r \in \mathbf{SAT}_R(\mathcal{B})$
- **Lokal erweitertes Datenbankschema**:
  - Menge $\mathcal{S} = \{\mathcal{R}_1, .., \mathcal{R}_j\}$ von erweiterten Relationenschemata
- **Datenbank über $\mathcal{S} = \{\mathcal{R}_1, .., \mathcal{R}_j\}$**:
  - Menge $d = \{r_1, .., r_j\}$ mit $r_i \in \mathrm{SAT}_{R_i}(\mathcal{B}_\rangle))$

Fremdschlüssel und globale Integritätsbedingungen

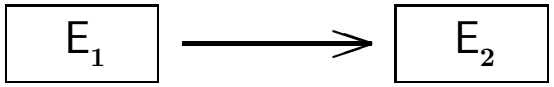
Bedingungen an das Zusammenspiel der Relationen

- **Identifizierende Attributmenge für** R: Menge $K = \{B_1, ..B_k\} \subseteq R$ mit
  $\forall t_1, t_2 \in r.(\forall B \in K.t_1(B) = t_2(B)) \Rightarrow t_1 = t_2$ für alle $r \in \mathrm{REL}(R)$
- **Fremdschlüssel für** R_i: Attributmenge $K \subseteq R_i$, zu der es in einem Relationenschema R_k
  einen kompatiblen Primärschlüssel $K_k \subseteq R_k$ gibt
  – kompatibel: in der aktuellen Datenbank $d = \{r_1, .., r_j\}$ gilt
  $\{t(K_i) | t \in r_i\} \subseteq \{t(K_k) | t \in r_k\}$
- **Globale Integritätsbedingung für** S:
  – Menge Γ von Abbildungen γ:$\mathbf{DB}(S) \rightarrow$ `bool`
  – Fremdschlüssel sind globale Integritätsbedingungen
- **Global erweitertes Datenbankschema**: $\mathcal{S} = (S, \Gamma)$
  – S Datenbankschema, Γ globale Integritätsbedingung für S
- **Datenbank über** $\mathcal{S} = (S, \Gamma)$:
  – Datenbank d mit $\gamma(d) =$ `true` für alle $\gamma \in \Gamma$ – $d \in \mathbf{DAT}_R(\mathcal{B})$

Zusammenfassung der wichtigsten Begriffe

Heuer/Saake Folie 98 120%

Darstellung von Entity-Relationship Modellen

- **Entity-Typen $E(A_1{:}D_1,\ldots,A_m{:}D_m)$**
  - einfaches Relationenschema $\{A_1,\ldots,A_m\}$ mit Namen E
- **Entity-Typen E mit strukturiertem Attribut A:list(D)**
  - Relationenschema für E enthält A nicht
  - zusätzliches Relationenschema $\{key_E,A\}$ mit Schlüsselattributen key_E von E
- **Beziehungstypen $R(E_1,..,E_n;\ A_1{:}D_1,\ldots,A_m{:}D_m)$**
  - Relationenschema $\{key_{E_1},..,key_{E_n},A_1,\ldots,A_m\}$ mit Namen R
- **Funktionale (1:n) Beziehungen $R(E_1,E_2)$** ohne eigene Attribute
  - Ergänze Schema für E_1 um Schlüsselattribute von E_2 (Fremdschlüssel)
  - kein separates Relationenschema für R
- **Rekursive 1:1 Beziehungen R(E,E)** ohne eigene Attribute
  - Ergänze Schema für E um neues Attribut, das key_E entspricht
  - z.B. verheiratet(Frau:PERSON, Mann:PERSON) $\mapsto$ PERSON(...,Gatte)
- **IS_A-Beziehung** 

  - Schema für E_1 enthält nur Schlüsselattribute von E_2 und neue Attribute
  - speichereffizient, aber aufwendige Suche und Aktualisierung ($\mapsto$ Alternativen?)

Universitätsdatenbank – Repräsentation von Objekten

- Entity-Typen
  - Person: {PANr,Vorname,Nachname,PLZ,Ort,Straße,HNr,Geb.datum}
  - Mitarbeiter: {PANr,AngNr,Fachbereich,Gehalt,Raum,Einstellung} *spezialisiert* Person
  - Professoren: {PANr,Lehrstuhlbezeichnung,Stufe} *spezialisiert* Mitarbeiter
  - Studenten: {PANr,Matrikelnummer,Studienfach,Immatrikulationsdatum} *spezialisiert* Person
  - Lehrstühle: {Lehrstuhlbezeichnung,Anzahl_Planstellen}
  - Vorlesungen: {V_Bezeichnung,SWS,Semester,Studiengang}
  - Bücher: {ISBN,Titel,Typ,Verlagsname}
  - Buch_Versionen: {ISBN,Auflage,Jahr,Seiten,Preis} *spezialisiert* Bücher
  - Buch_Exemplare: {Inventarnr,ISBN,Auflage} *zusätzlich zum ER-Modell*
  - Verlage: {Verlagsname,Verlagsort}
  - Lehrbuch,Tagungsband: *codiert im Atrribut* Typ *von* Bücher
- Strukturierte Attribute
  - Pers_Telefon: {PANr,Telefon} *für* Person
  - Buch_Autor: {ISBN,Autor} *für* Bücher
  - Buch_Stichwort: {ISBN,Stichwort} *für* Bücher

Universitätsdatenbank
Repräsentation von Beziehungen

- Ausleihe: {PANr,Inventarnummer}
- Prüft: {PANr,Matrikelnummer,V_Bezeichnung,Note}
- Empfiehlt: {PANr,ISBNV_Bezeichnung}
- Vorl_Voraus: {V_Bezeichnung,Voraussetzung}
- Liest: {PANr,V_Bezeichnung,Semester}
- Hört: {Matrikelnummer,V_Bezeichnung,Semester}
- hat(Professor,Lehrstuhl): *funktional codiert in der Relation* Professoren
- von(BuchExemplar,Buch): *funktional codiert in der Relation* Buch_Exemplare
- in(Buch,Verlag): *funktional codiert in der Relation* Bücher

Beschreibung von Teilansichten einer Datenbank

- **Relationenalgebra (RA)**
  - Anwendung algebraischer Operationen auf Relationen der Datenbank
  - Ausdrücke zusammengesetzt aus Grund- und abgeleiteten Operatoren

    Grundoperatoren: Vereinigung $\cup$, Differenz -, Produkt $\times$, Projektion π, Selektion σ, Verbund $\bowtie$
- **Relationentupelkalkül (RTK)**
  - Deskriptive Beschreibung durch Ausdrücke der Form $\{t \mid \Psi(t)\}$
  - Ψ prädikatenlogische Formel bestehend aus
    - atomaren Formeln: $t \in r,\ t(A)\,\rho\,t'(B),\ t(A)\,\rho\,a,\ a\,\rho\,t(A)$

      t, t' Tupelvariablen, r Relation, A, B Attribute, a Konstante, $\rho \in \{=,\neq,\leq,<,\geq,>\}$
    - logischen Symbolen $\neg,\ \wedge,\ \vee,\ \Rightarrow,\ \forall,\ \exists$
  - Semantische Beschränkung auf endliche Relationen
- **Relationenwertebereichskalkül (RWK)**
  - wie RTK, mit Ausdrücken $\{x_1 ... x_k \mid \Psi(x_1 ... x_k)\}$, ($x_i$ Wertebereichsvariable)
- **Varianten:** geordnete / ungeordnete Tupel (Indizes/Attribute)
- **RA, RTK und RWK sind äquivalent**
  - Beschreibung der Semantik von RA-Operatoren oft im RTK

Relationenalgebra: Selektion, Projektion

- Selektion: $\boldsymbol{\sigma_P(r) := \{t \in r \mid \text{`}P(t)'\}}$
  - (logische) Auswahl von Tupeln einer Relation
  - P aussagenlogische Formel bestehend aus
    - Operanden: Konstante oder Attribute
    - (zulässige) Vergleichsoperatoren $\rho \in \{=, \neq, \leq, <, \geq, >\}$ auf Attributwerten
    - Logische Symbole $\neg$, $\wedge$, $\vee$

  Konstanten-Selektion: $\boldsymbol{\sigma_{X \rho x}(r) := \{t \in r \mid t(X) \rho x\}}$ $(X \subseteq R)$
  - Vergleiche X-Komponenten der Tupel aus r mit dem X-Wert x

  Attribut-Selektion: $\boldsymbol{\sigma_{X \rho Y}(r) := \{t \in r \mid t(X) \rho t(Y)\}}$ $(X, Y \subseteq R)$
  - Vergleiche X- und Y-Komponenten der Tupel aus r

- Projektion: $\boldsymbol{\pi_X(r) := \{t(X) \mid t \in r\}}$ $(X \subseteq R)$
  - Auswahl von Spalten einer Relation
  - Doppelte Tupel werden entfernt (Relationen sind Mengen)

Relationenalgebra: Produkt und Quotient

- Produkt: $\underline{\boldsymbol{r_1 \times r_2}} := \{\boldsymbol{t_1 \otimes t_2} \mid \boldsymbol{t_1 \in r_1} \wedge \boldsymbol{t_2 \in r_2}\}$
  - Menge aller Tupel, die durch Kombination von r_1- und r_2-Tupeln entstehen
  - $\underline{\boldsymbol{t_1 \otimes t_2}}$: Abbildung $t{:}R_1 \dot\cup R_2 \rightarrow \bigcup_{\text{i=1..m}} D_i$ mit $t(R_1){=}t_1$ und $t(R_2){=}t_2$
  - $\underline{\boldsymbol{R_1 \dot\cup R_2}}$: disjunkte Vereinigung der Attribute von R_1 und R_2
  - evtl. Umbenennung der Atrribute erforderlich

  Nicht identisch mit konventionellem kartesischen Produkt!

- Quotient: $\underline{\boldsymbol{r_1 \div r_2}} := \{\boldsymbol{t \in \mathbf{REL}(R_1 - R_2)} \mid \forall \boldsymbol{t_2 \in r_2}.\ \boldsymbol{t_1 \otimes t_2 \in r_1}\}$
  - Menge aller Tupel, deren Kombination mit allen r_2-Tupeln zu R_1 gehört
  - Inverse Operation zum Produkt: $(r_1 \times r_2) \div r_2 = r_1$

Relationenalgebra: Verbund (Join)

- **Θ**-Join: $\boldsymbol{r_1 \underset{A\Theta B}{\rhd\lhd} r_2 := \{t_1 \otimes t_2 \mid t_1 \in r_1 \wedge t_2 \in r_2 \wedge t(A)\Theta t(B)\}}$

  ($A \in R_1$, $B \in R_2$, $\Theta \in \{=, \neq, \leq, <, \geq, >\}$ zulässiger Vergleichsoperator auf dom(A) und dom(B))

  – Produkt von R_1 und r_2, eingeschränkt durch Θ-Bedingung zwischen A und B

- Verbund: $\boldsymbol{\underline{r_1 \bowtie r_2} := \{t \in \mathbf{REL}(R_1 \cup R_2) \mid t(R_1) \in r_1 \wedge t(R_2) \in r_2\}}$

  – Natürlicher Verbund: Verknüpfung von Tupeln mit gemeinsamen Attributwerten

  – 'Gleichverbund' (Θ ist '=') über alle gleichen Attribute mit anschließender Projektion der verschiedenen Attribute (Elimination der doppelten)

  – $r_1 \bowtie r_2 = r_1 \cap r_2$, wenn $R_1 = R_2$ $\qquad$ $r_1 \bowtie r_2 = r_1 \times r_2$, wenn $R_1 \cap R_2 = \emptyset$

  – Für Kombination von Relationen, die aus Entwurfsgründen zerlegt wurden

- Semi-Verbund: $\boldsymbol{\underline{r_1 \ltimes r_2} := \{t_1 \in r_1 \mid \exists t_2 \in r_2.\ t_1(R_1 \cap R_2) = t_2(R_1 \cap R_2)\}}$

  – Menge der r_1 Tupel, die mit einem r_2-Tupel gemeinsame Attributwerte haben

  – Wichtig für Optimierung verteilter Datenbanksysteme

  – Simulierbar durch $r_1 \ltimes r_2 = r_1 \bowtie \pi_{R_1 \cap R_2}(r_2)$

Relationenalgebra: Standard-Mengenoperationen

- Vereinigung: $\underline{r_1 \cup r_2} := \{t \mid t \in r_1 \vee t \in r_2\}$
  - Menge aller Tupel, die aus mindestens einer von zwei Relationen stammen
- Differenz: $\underline{r_1 - r_2} := \{t \mid t \in r_1 \wedge t \notin r_2\}$
  - Menge aller Tupel, die in r_1 aber nicht in r_2 enthalten sind
- Durchschnitt: $\underline{r_1 \cap r_2} := \{t \mid t \in r_1 \wedge t \in r_2\}$
  - Menge aller Tupel, die in r_1 und in r_2 enthalten sind
  - simulierbar durch $r_1 \cap r_2 = r_1 - (r_2 - r_1)$
- Symmetrische Differenz: $\underline{r_1 \triangle r_2} := \{t \mid t \in r_1 \dot{\vee} t \in r_2\}$
  - Menge aller Tupel, die aus genau einer von zwei Relationen stammen
  - simulierbar durch $r_1 \triangle r_2 = r_1 \cup r_2 - r_1 \cap r_2$

Achtung: gleichartige Relationen können verschiedene Attribute haben

- Umbenennung $\underline{\beta_{B \leftarrow A}(r)} := \{t' \mid \exists t \in r.\ t'(B) = t(A) \wedge t'(R \backslash A) = t(R \backslash A)\}$ $(A \in R, B \notin R \backslash A)$
  - nötig, um Mengenoperationen kompatibel zu machen
  - Benennt Attribut A in B um, wenn $\mathrm{dom}(A) = \mathrm{dom}(B)$
  - Erzeugt modifiziertes Relationenschema $R' = (R \backslash A) \cup \{B\}$

Formulierung von Anfragen in der Relationenalgebra

Relationenschema PKW mit Attributen Marke, Modell, Werk

Relationenschema ORT mit Attributen Werk, Land

Marke	Modell	Werk
VW	Käfer	Puebla
VW	Golf	Puebla
VW	Golf	Brasilia
VW	Golf	Sidney
VW	Jetta	Sidney
VW	Golf	Wolfsburg
VW	Brasilia	Brasilia
VW	Brasilia	Puebla
VW	Polo	Wolfsburg
Ford	Fiesta	Köln
Ford	Fiesta	Detroit
Ford	Taurus	Detroit
Ford	Taurus	Toronto
Ford	Escort	St. Louis
Ford	Escort	Los Angeles

Werk	Land
St. Louis	USA
Toronto	CAN
Brasilia	MEX
Detroit	USA
Sidney	AUS
Wolfsburg	BRD
Los Angeles	USA
Köln	BRD
Puebla	MEX

- *'Finde alle Pkw, deren Modell* Käfer *oder* Golf ist*'*

$$\boldsymbol{\sigma}_{\text{Modell='Käfer'} \vee \text{Modell='Golf'}}(\text{PKW})$$

Marke	Modell	Werk
VW	Käfer	Puebla
VW	Golf	Puebla
VW	Golf	Brasilia
VW	Golf	Sidney
VW	Golf	Wolfsburg

- *'Finde alle Werke, in denen ein* Käfer, Golf, Fiesta *oder* Taurus *hergestellt wird'*

$$\boldsymbol{\Pi}_{\text{Werk}}(\boldsymbol{\sigma}_{\text{Modell='Käfer'} \vee \text{Modell='Golf'} \vee \text{Modell='Fiesta'} \vee \text{Modell='Taurus'}}(\text{PKW}))$$

Marke	Modell	Werk
VW	Käfer	Puebla
VW	Golf	Puebla
VW	Golf	Brasilia
VW	Golf	Sidney
VW	Golf	Wolfsburg
Ford	Fiesta	Köln
Ford	Fiesta	Detroit
Ford	Taurus	Detroit
Ford	Taurus	Toronto

Werk
Puebla
Brasilia
Sidney
Wolfsburg
Köln
Detroit
Toronto

- *'Finde alle Marken und Modelle aus Deutschland oder Mexiko'*

$$\Pi_{\text{Marke,Modell}}\left(\sigma_{\text{Land='BRD'} \vee \text{Land='MEX'}}(\text{PKW} \bowtie \text{ORT})\right)$$

Marke	Modell	Werk	Land
VW	Käfer	Puebla	MEX
VW	Golf	Puebla	MEX
VW	Golf	Brasilia	MEX
VW	Golf	Sidney	AUS
VW	Jetta	Sidney	AUS
VW	Golf	Wolfsburg	BRD
VW	Brasilia	Brasilia	MEX
VW	Brasilia	Puebla	MEX
VW	Polo	Wolfsburg	BRD
Ford	Fiesta	Köln	BRD
Ford	Fiesta	Detroit	USA
Ford	Taurus	Detroit	USA
Ford	Taurus	Toronto	CAN
Ford	Escort	St. Louis	USA
Ford	Escort	Los Angeles	USA

...

Marke	Modell
VW	Käfer
VW	Golf
VW	Brasilia
VW	Polo
Ford	Fiesta

Besser: $\Pi_{\text{Marke,Modell}}\left(\text{PKW} \bowtie \sigma_{\text{Land='BRD'} \vee \text{Land='MEX'}}(\text{ORT})\right)$

Werk	Land
Brasilia	MEX
Wolfsburg	BRD
Köln	BRD
Puebla	MEX

Marke	Modell	Werk	Land
VW	Käfer	Puebla	MEX
VW	Golf	Puebla	MEX
VW	Golf	Brasilia	MEX
VW	Golf	Wolfsburg	BRD
VW	Brasilia	Brasilia	MEX
VW	Brasilia	Puebla	MEX
VW	Polo	Wolfsburg	BRD
Ford	Fiesta	Köln	BRD

Marke	Modell
VW	Käfer
VW	Golf
VW	Brasilia
VW	Polo
Ford	Fiesta

Optimierung von Anfragen

- Gegeben: ein Ausdruck der Relationenalgebra
- Gesucht: äquvalenter, möglichst effizient auszuführender RA-Ausdruck
- Methode: Heuristische Auswahl von Äquvalenzumformungen

• Umformungsregeln für RA-Ausdrücke

- **Kommutativität**: $r_1 \circ r_2 \equiv r_2 \circ r_1$ für $\circ \in \{\bowtie, \times, \cup, \cap\}$
- **Assoziativität**: $(r_1 \circ r_2) \circ r_3 \equiv r_1 \circ (r_2 \circ r_3)$ für $\circ \in \{\bowtie, \times, \cup, \cap\}$
- **Projektionsfolgen**: $\pi_{A_1,..,A_k}(\pi_{A_1,..,A_k,B_1,..B_m}(r)) = \pi_{A_1,..,A_k}(r)$
- **Selektionsfolgen**: $\sigma_P(\sigma_Q(r)) = \sigma_{P \wedge Q}(r) \ldots = \sigma_Q(\sigma_P(r))$
- σ-π **Vertauschung**: $\sigma_P(\pi_{A_1,..,A_k}(r)) = \pi_{A_1,..,A_k}(\sigma_P(r))$ (P enthält nur $A_1, .., A_k$)
- σ-$\times$ **Vertauschung**: $\sigma_P(r_1 \times r_2) = \sigma_P(r_1) \times r_2$ (P enthält nur r_1-Attribute)

• Heuristiken zur Effizienzsteigerung

- Selektion so früh wie möglich anwenden
- Einfache Selektionen zusammenfassen ($\mapsto$ keine Zwischenergebnisse)
- Projektion ohne Duplikatenelimination (teuer!!) möglichst früh
- Gemeinsame Zwischenergebnisse nur einmal berechnen (Speicheraufwand?)
- Minimiere Größe der Zwischenergebnisse durch Anpassung der Verbundreihenfolge
- Verknüpfe zuerst die kleinsten Relationen

Zusammenfassung

- **Datenstruktur Tabelle (Relation)**
  - einzige Datenstruktur neben atomaren Werten
  - alle Informationen ausschließlich durch Werte dargestellt

  $\Rightarrow$ Integritätsbedingungen zwischen Relationen erforderlich
- **Abbildung von Beziehungen**
  - Simulation durch 1:n Beziehungen zwischen Relationen
  - Hinzunahme neuer Relationen bei komplexeren Beziehungen
  - Kardinalität/Komplexitätsbeschränkungen nur eingeschränkt darstellbar
- **Abbildung der Abstraktionskonzepte**
  - Generalisierung, Aggregation nicht direkt darstellbar
  - IS_A-Beziehung nur beschränkt simulierbar
- **Anfrage- und Manipulationssprachen**
  - Navigierend auf Basis der Relationenalgbra
  - Deskriptiv auf Basis von Relationentupelkalkül, Relationenwertebereichskalkül

Grundlagen der Datenbanken

Lektion 6

Relationale Datenbanksprachen I: SQL

1. Relationale Datenbanksprachen
   – Anforderungen und Übersicht
2. Anfragen in SQL
   – Kernbestandteile von SQL-89
   – Ereiterungen in SQL-92
3. Änderungsoperationen in SQL

Grundaufgaben von Datenbanksprachen

Ableitung virtueller Relationen

- Anfrage $\mapsto$ DML
  - Folge von Operationen, die aus Basisrelationen neue Relatione bestimmt
  - Ergebnis interaktiv angezeigt oder weiterverarbeitet durch Programme
- Sicht $\mapsto$ DDL
  - Folge von Anfrageoperationen, die unter festem Namen abgelegt werden
  - Sichtrelation wird bei jedem Aufruf neu berechnet
- Schnappschuß
  - Unter Namen abgespeicherte Ergebnisrelation einer Anfrage (konstant!)

Modifikation von Basisrelationen

- Update $\mapsto$ DML
  - Erzeugen, Löschen oder Ändern von Tupelmengen (Teilrelationen)
  - Konsistenzprüfung erforderlich

Datenbeschreibung

- Relationenschemata Erzeugen, Löschen, Ändern $\mapsto$ DDL
- Indexstrukturen Erzeugen, Löschen $\mapsto$ SSL

$\vdots$

Anforderungen an Datenbanksprachen

- **Vollständigkeit:**
  - Anfragesprache umfaßt Ausdruckskraft der Relationenalgebra (bzw. RTK)
- **Zusatzfunktionen:**
  - Update-Kommandos und Zuweisung berechneter Werte an Relationen
  - Aggregationsfunktionen: Summe, Minimum, Maximum, Mittelwert, ...
  - Berechnung der transitiven Hülle (bei binären 'reflexiven' Relationen)
- **Abgeschlossenheit:**
  - Anfragen liefern Relationen, die weiterverarbeitet werden können
- **Ad-hoc Formulierbarkeit und Orthogonalität:**
  - Anfragen losgelöst von Programmen leicht formulierbar
  - Sprachkonstrukte in ähnlichen Situationen ähnlich anwendbar
- **Deskriptiv und mengenorientiert:**
  - Operationen sagen, was man haben will
  - Operationen auf ganzen Relationen (nicht navigierend auf Tupeln)
- **Effizienz und Optimierbarkeit:**
  - Spezielle Algorithmen für Grundoperationen, Reformulierung von Anfragen
- **Sicherheit:**
  - keine Endlosschleifen bei korrekten Anfragen ($\mapsto$ keine volle Programmiersprache)

Übersicht über Relationale Datenbanksprachen

- **SQL: Structured English QUEry Language**
  - ANSI-ISO-Norm-Datenbanksprache für Relationale Datenbanksysteme
  - Mischung von Relationenalgebra und Relationentupelkalkül
  - Standards unter ständiger Erweiterung
  - Verschiedene Versionen und Levels im praktischen Einsatz
- **QUEL: Query-Sprache von INGRES**
  - Rein deskriptiv (Relationentupelkalkül)
  - Großer Einfluß auf Forschungsarbeiten
  - Nur mäßige praktische Verbreitung
- **QBE: Query-by-Example**
  - Formorientierte Sprache für naive Benutzer (Relationenwertebereichskalkül)
  - Graphische unterstützte Schnittstelle
  - Wachsende Verbreitung
- **Universalrelationen-Anfragesprachen**
  - Attribute einer virtuellen 'Universalrelation' ersetzen einzelne Basisrelationen
- **DATALOG: Regelbasierte Anfragesprache**
  - Mengenbasierte Datenbanksprache auf PROLOG-Basis

SQL: Structured English QUEry Language

Normsprache für Relationale Datenbanksysteme

- **Strukturierte Sprache mit englischen Schlüsselwörtern**
  - Selbsterklärende Schlüsselworte ersetzen RA-Operationen und RTK-Formeln
  - Vermeidung komplizierter mathematischer Konstrukte (Quantoren)
- **Genormte kommerzielle Form einer Forschungsprache**
  - Untermenge von SEQUEL2 (Nachfolger von SEQUEL – für System R, 1976)
  - SQL-86: erste Normierung durch ANSI
  - SQL-89: ANSI-ISO Norm mit Integritätssicherung IEF, 3 Ebenen

    Aktueller Stand bei vielen kommerziellen Systemen
  - SQL-92 (SQL2): aktuell gültige revidierte Norm, 3 Ebenen
  - SQL3 Projekt: objektorientierte und andere aktuelle Erweiterungen (1996)

  Normbeschreibung extrem lang $\Rightarrow$ Notationsvarianten in Kurzpräsentationen
- **Enthält mehrere Teilsprachen**
  - Datenmanipulation (DML): Anfrage und Updates,
  - Datenbeschreibung (DDL): Relationenschemata und Sichten
  - Datenkontrolle: Zugriffsrechte, Integritätskontrolle
  - Speicherstrukturen (SSL): Indexstrukturen
  - Koppelung mit Wirtssprache

SQL-Anfragen: Basisstruktur

select $A_1, .., A_n$ **from** $r_1, .., r_m$ [**where** P] $\hat{=}$ $`\pi'_{A_1,..,A_n}(\sigma_P(r_1 \times .. \times r_m))$

- **select**
  - Multimengen-Projektion mit Duplikaten
  - Projektionsliste zählt gewünschte Attribute der Zielrelation auf
  - Arithmetische Operationen und Aggregatfunktionen
- **from**
  - Zu verwendende Relationen
  - Mehrere Relationen als Produkt/Verbund kombinierbar (ggf. Umbenennung)
- **where**
  - Selektions- und Verbundbedingungen für Relationen der **from**-Klausel
  - Geschachtelte Anfragen (als **select-from-where** (SFW) Block)

- **group by** – Virtuelle Gruppierung von Tupeln für 'lokale' Aggregatfunktionen
- **having** – Selektionsbedingungen für Auswahl von Gruppen
- **order by** – Sortierung der Ergebnisrelation
  - Grundlage: Ordnung auf Wertebereichen selektierter Attribute

Die **select**-Klausel

select [**distinct**] { Attribut | Index | arith. Ausdruck | Aggregatfunktion | * } ...

- **Festlegung von Attributen der Ergebnisrelation**
- **Auswahlkriterien**
  - Attribute der mit **from** ausgewählten Relationen
  - Spaltenindex einer Relation
  - Arithmetische Ausdrücke über Attributen dieser Relationen
  - Aggregatfunktionen über Attributen dieser Relationen
  - *: Auswahl aller Attribute
- **Zielrelation normalerweise Multimenge**
  - Elimination von Duplikaten teuer und oft unnötig
  - Echte Projektion (Ergebnismenge) durch Schlüsselwort **distinct**
- **Mehrdeutigkeiten auflösbar durch Relationenpräfix**
  - z.B. **select** `Bücher.ISBN,Titel,Stichwort` **from** `Bücher,BuchStichwort` falls `ISBN` Attribut von `Bücher` und `BuchStichwort`

Berechnungen während einer Selektion/Projektion

- **Arithmetische Ausdrücke**
  - Operieren auf einzelnen Tupeln einer Relation
  - Bestandteile: skalare Operationen, verfügbare Attribute, Konstanten
    Operationen z.B. `+,-,*,/` (Zahlen); `length, substring, ||` (Strings); ...

Anwendung in select-Klausel:

select `ISBN, Preis/1.51` **from** `BuchVersion`

- Erzeugt Relation mit abgeleitetem Attribut (in SQL-89 ohne Namen)

$\Rightarrow$ Zugriff mit Index erforderlich: `select 2 ...`)

- **Aggregatfunktionen**
  - Operieren auf allen selektierten Tupeln einer Relation – Ergebnis skalar
  - Vordefiniert: sum, avg, max, min, count
  - Argumente: verfügbare Attribute, arithmetische Ausdrücke, * (nur für count)
  - Parameter: distinct (vorherige Elimination doppelter Elemente)
    all (Operation auf Multimenge / Default)

Beispielanfragen in SQL

- *Finde alle Noten der Relation* `Prüft`

  select `Note` **from** `Prüft`

  Duplikate werden nicht eliminiert (sinnvoll für Statistiken)

- *Finde alle Kunden(namen) der Relation* `Konto`

  select distinct `K_Name` **from** `Konto`

  Duplikate werden eliminiert

- *Bestimme die Durchschnittsnote aller Prüfungen*

  select avg `(`**all** `Note)` **from** `Prüft`

- *Bestimme die Anzahl der (verschiedenen) Prüfer aller Prüfungen*

  select count `(`**distinct** `PANr)` **from** `Prüft`

- *Bestimme die Anzahl aller Prüfungen*

  select count`(*)` **from** `Prüft`

- *Finde alle Kunden, die ein Konto in der Innenstadtfiliale haben*

  select `K_Name` **from** `Konto` **where** `Filiale = 'Innenstadt'`

Die **from**-Klausel

$$\ldots \textbf{from}\ r_1\ [\text{var}_1]\ [,\ldots,r_n\ [\text{var}_n]]$$

- **Auflistung auszuwählender Basisrelationen**
  - Bei mehr als einer Relation wird das Produkt gebildet
- **Tupelvariablen**
  - lokale Benennung von Tupeln einer Relation
  - ermöglichen mehrfachen Zugriff auf dieselbe Relation
    z.B. `...from Bücher b1, Bücher b2`
  - Attribute unter Tupelnamen zugreifbar (z.B. `b1.InvNr, b2.Titel`)

SQL-92: zusätzlich Verbundbildung und lokale Namen für Zwischenrelationen

Einfache **where**-Klauseln

. . . **where** {Bedingung}: **Selektion von Tupeln der Ergebnisrelation**

Bedingung zusammengesetzt aus

- Konstantenselektion: X ρ x Attributselektion: X ρ Y
- Verbundbedingung: r_1.X Θ r_2.X
- Bereichsselektion: X **between** x **and** y
  - Abkürzung für X≥x **and** X≤y
- Ungewißheitsselektion: X **like** pattern
  - Vergleich von pattern mit 'ähnlichen' Strings (wie in Unix)
  - Wildcards: _ (ein beliebiges Zeichen) % (beliebig viele Zeichen)
  - z.B. `H_lle` paßt zu `Halle, Hülle, Hölle, Hxlle, ...`
- Nullselektion: X **is null**
  - Auswahl von Tupel, die 'Nullmarken' (z.B. undefinierte Werte) enthalten
- Logische Konnektoren: **and**, **or**, **not**
  - z.T verschiebbar in Ausdrücke (z.B. X **not between** x **and** y, X **is not null**)

$\rho,\Theta \in \{=,\neq,\leq,<,\geq,>\}$, X,Y Attribute; x,y Konstante – auch Ergebnis arithmetischer Ausdrücke
Wertebereiche müssen kompatibel sein (gleich, Strings oder numerisch)

Beispielanfragen in SQL (II)

- *Matrikelnummer aller Studenten, die eine Prüfung mit gut bestanden haben*

```
select distinct Matrikelnummer from Prüft
    where Note between 1.7 and 2.3
```

- *Namen aller Studenten, deren Matrikelnummer mit 38 beginnt*

```
select distinct Vorname, Nachname from Student
    where Matrikelnummer like '38%'
```

- *Namen und Adresse aller Kunden, die einen Kredit haben*

```
select Kunde.K_Name, Kunde.Adresse from Kredit, Kunde
    where Kredit.K_Name = Kunde.K_Name
```

  Verbundoperation (über `K_Name`) mit Projektion

- *Namen und Adresse aller Kunden, die bei der gleichen Filiale wie Herr Schmidt ein Konto haben*

```
select Kunde.K_Name, Adresse from Kunde, Konto K1, Konto K2
    where K1.K_Name = 'Schmidt' and K1.Filiale = K2.Filiale
```

- *ISBN-Nummern aller Bücher von Heuer und Saake* (Selbstverbund !)

```
select B1.ISBN from Buch B1, Buch B2
    where B1.ISBN = B2.ISBN and B1.Autor = 'Heuer' and B2.Autor = 'Saake'
```

Geschachtelte **where**-Klauseln

Selektionsbedingung erlaubt Vergleiche mit Tupelmengen

- **Elementbeziehung: X in (select .. from . . .)**
  - Test, ob Attributwert in Zielrelation des SWF-Blocks vorkommt
- **Blockkonzept für verschachtelte SWF-Blöcke**
  - ein Name innerhalb eines Blocks referenziert auf die letzte 'Deklaration'
  - z.B. `...from Person where PANr in (select PANr from Prüft)`
- **Verzahnt geschachtelte Anfragen**
  - innerer SWF-Block benutzt Relationen/Tupelvariablen des äußeren Blocks
  - z.B.
    ```
    ...from Person where 1.0 in
                (select Note from Prüft where PANr = Person.PANr)
    ```
  - Abarbeitung: 1. wähle Tupel der äußeren Anfrage
    2. werte innere Anfrage mit konkretem äußeren Tupelwert aus
    3. überprüfe **in**-Prädikat mit konkretem Wert
- **Existenztest: exists (select * from .. where P)**
  - Test, ob die Zielrelation der inneren Anfrage nicht leer ist ('es gibt Elemente')
  - Allquantor simulierbar durch **not exists** (**select** * **from** .. **where not** P)

where-Klauseln mit quantifizierten Bedingungen

. . . **where** X ρ {**all** | **any** | **some** } (**select** .. **from** . . .)

Vergleich mit allen Einträgen einer Relation

- **Allquantor: X ρ all (select .. from . . .)**
  - X steht mit allen selektierten Tupeln in Relation ρ
  - z.B. `Note` $\leq$ `all (select Note from Prüft where Fach = 'Informatik')`

    jemand, der mindestens so gut war wie der beste Informatiker

- **Existenzquantor: X ρ any (select .. from . . .)**
  - X steht mit einem selektierten Tupel in Relation ρ
  - z.B. `Note < any (select Note from Prüft where Fach = 'Informatik')`

    jemand, der besser war als der schlechteste Informatiker
  - **some** ist identisch mit **any**, zuweilen aber sprachlich angemessener

- **Aggregatfunktionen: X ρ (select F(..) from . . .)**
  - **sum**, **avg**, **max**, **min**, **count**

Beispielanfragen in SQL (III)

- *Finde alle Studenten, die mindestens eine Prüfung besser als der Durchschnitt abgelegt haben*

```
select distinct Matrikelnummer from Prüft
    where Note < (select avg (all Note) from Prüft)
```

- *Finde alle Studenten, die in der Datenbanken Prüfung die Bestnote hatten*

```
select distinct Matrikelnummer from Prüft
    where V_Bezeichnung = 'Datenbanken'
      and Note ≤ all (select Note from Prüft
                      where V_Bezeichnung = 'Datenbanken')
```

```
select distinct Matrikelnummer from Prüft
    where V_Bezeichnung = 'Datenbanken'
      and Note ≤ (select min(Note) from Prüft
                  where V_Bezeichnung = 'Datenbanken')
```

- *Finde alle Filialen, die mehr als 500 Kontoinhaber haben*

```
select filiale from Konto Fil
    where 500 < (select count(K_Name)from Konto
                 where Fil.Filiale = Filiale)
```

Mengenoperationen

- **Vereinigung:**

  (**select** $A_1,..,A_n$ **from** $r_1 \ldots$) **union** (**select** $B_1,..,B_n$ **from** $r_2 \ldots$)

  – Attributnamen spielen keine Rolle

  – Attribute bzw. Spalten von r_1 und r_2 müssen kompatibel sein

  – Duplikate werden entfernt (außer bei ... **union all** ...)

r_1	A	B	C
	1	2	3
	2	3	4
	3	4	5

r_2	A	E	F
	1	5	3
	2	3	4
	5	6	7

union	A	B	C
	1	2	3
	2	3	4
	3	4	5
	1	5	3
	5	6	7

  – In SQL-89 nur als äußerste Operation (**union** nicht innerhalb einer Anfrage)

  $\Rightarrow$ *SQL-89 ist nicht abgeschlossen*

- **Differenz und Durchschnitt in SQL-89 nicht explizit**

  – Simulierbar durch **where** und **in**

  $r_1 \cap r_2$: **select distinct** $r_1.A_1,..,r_1.A_n$ **from** r_1,r_2 **where** $r_1.A_1=r_2.B_1$ **and** ... $r_1.A_n=r_2.B_n$

  $r_1 - r_2$: **select distinct** $r_1.A_1$ **from** r_1,r_2 **where** $r_1.A_1 \neq r_2.B_1$ (schwierig bei mehreren Attributen)

  – Ineffizient, Formulierung optimierter Versionen für Spezialfälle fehleranfällig

Beispielanfragen in SQL (IV)

- *Finde alle Kunden, die Kredit oder Konto bei der Innenstadtfiliale haben*

```
(select K_Name  from Konto where Filiale = 'Innenstadt')
   union
(select K_Name  from Kredit where Filiale = 'Innenstadt')
```

- *Finde alle Kunden, die Kredit und Konto bei der Innenstadtfiliale haben*

```
select K_Name  from Konto
     where Filiale = 'Innenstadt'
       and K_Name in (select K_Name  from Kredit
                                  where Filiale = 'Innenstadt'
                      )
```

- *Finde alle Kunden, die einen Kredit, aber kein Konto bei der Innenstadtfiliale haben*

```
select K_Name  from Konto
     where Filiale = 'Innenstadt'
       and K_Name not in (select K_Name  from Kredit
                                  where Filiale = 'Innenstadt'
                          )
```

SQL ist (fast) relational vollständig

Alle Grundoperationen der Relationenalgebra sind darstellbar

- Projektion $\boldsymbol{\pi_{A_1,..A_n}}(\mathrm{r})$: select distinct $\mathrm{A}_1,..\mathrm{A}_n$ from r
- Selektion $\boldsymbol{\sigma_P}(\mathrm{r})$: select * from r where P
- Produkt $\mathrm{r}_1 \times \mathrm{r}_2$: select * from $\mathrm{r}_1,\mathrm{r}_2$
- Verbund $\mathrm{r}_1 \bowtie \mathrm{r}_2$: select distinct $\mathrm{r}_1.\mathrm{A}_1,..,\mathrm{r}_1.\mathrm{A}_n,\mathrm{r}_2.\mathrm{B}_{k+1},\mathrm{r}_2.\mathrm{B}_m$
  from $\mathrm{r}_1,\mathrm{r}_2$ where $\mathrm{r}_1.\mathrm{A}_1=\mathrm{r}_2.\mathrm{B}_1$ and $\ldots$ $\mathrm{r}_1.\mathrm{A}_k=\mathrm{r}_2.\mathrm{B}_k$

  (o.B.d.A. Übereinstimmung der ersten k Attribute)
- Umbenennung $\boldsymbol{\beta_{B \leftarrow A}}(\mathrm{r})$:
  – nur lokal durch Einsatz von Tupelvariablen und Präfixnotation
- Vereinigung $\mathrm{r}_1 \cup \mathrm{r}_2$: (select * from r_1) union (select * from r_2)
  – nur als äußerste Operation (nicht orthogonal!)
- Differenz $\mathrm{r}_1 - \mathrm{r}_2$: ????
  – nur im Spezialfall lösbar

Formulierung im Spezialfall oft fehleranfällig

Strukturierungsklauseln

- **group by $A_1,\ldots,A_n$**
  - Virtuelle Gruppierung von Tupeln nach gleichen Werten in den $A_1,\ldots,A_n$
  - Nur mit Aggregatfunktion, die auf alle Tupel einer Gruppe angewandt wird
  - z.B. `select Note, count(*) from Prüft group by Note` *(Notenstatistik)*
- **group by $A_1,\ldots,A_n$ having P**
  - Zusätzliche Einschränkung der ausgewählten Gruppen
  - z.B. `select Note, count * from Prüft group by Note having Note>4.0`
  - z.B.
    ```
    select Matrikelnummer from Prüft group by Matrikelnummer
      having max(Note) < (select avg (all Note) from Prüft)
    ```
    alle Studenten, die in allen Prüfungen besser als der Durchschnitt waren

  Abarbeitung:
  1. Auswahl von Tupeln mit **where**
  2. Gruppierung der Tupel mit **group by**
  3. Auswahl von Gruppen mit **having**
- **order by A_1 [asc|desc],$\ldots$,A_n [asc|desc]**
  - Benutzerdefiniere Reihenfolge der Ausgabe
  - Sortierung in der Reihenfolge der angegebenen Attribute (auf- oder absteigend)
  - Attribute müssen in **select**-Klausel vorkommen
  - z.B.
    ```
    select Matrikelnummer, Note from Prüft
      where V_Bezeichnung='Datenbanken' order by Matrikelnummer asc
    ```
    Notenliste der Datenbanken-Prüfungen, sortiert nach Matrikelnummern

Grundlagen der Datenbanken

Lektion 7

Relationale Datenbanksprachen II

1. Datenmanipulation in SQL
   - Erweiterungen der Anfragesprache in SQL-92
   - Änderungsoperationen
2. QUEL
3. QBE

Erweiterungen in SQL-92 (I)

- **Tupelbildung in der where-Klausel**
  - Tupel $(e_1,\ldots,e_n)$, wobei jedes Element e_i Konstante oder Attribut
  - $\Rightarrow$ Erweiterte Selektion: $(X_1,\ldots,X_n)\ \rho\ (e_1,\ldots,e_n)$ (lexikographische Ordnung)
  - $\Rightarrow$ Elementbeziehung mit Tupeln: X in $(x_1,\ldots,x_n)$ statt $X{=}x_1$ or $\ldots X{=}x_n$
  - Anfragen in Klammern (innerer SWF-Block), die ein Tupel liefern
  - z.B. `('Informatik',1.0) =`
    `(select Fach, Note from Prüft where Matrikenummer= 23456)`

    ... der Student mit Matrikelnummer 123456 hatte in der Informatik-Prüfung eine 1.0

- **Tupelvariablen für abgeleitete Attribute**

  z.B. `select ISBN, Preis/1.51 as DollarPreis from BuchVersion`

  $\Rightarrow$ übersichtlichere Zugriffe auf Attribute

- **Tupelvariablen für Zwischenrelationen**
  - ...from r_1 natural join r_2 as $neu{-}r$

  $\Rightarrow$ übersichtlichere Zugriffe auf Relationen

- **Abgeleitete Relationen in der from-Klausel**
  - ...from (select $e_1,\ldots,e_n$ from ...) as $neu{-}r(A_1,\ldots,A_n)$

Erweiterungen in SQL-92 (II)

Verbund von Relationen in der **from**-Klausel

- **Produkt:** ...**from** r_1, r_2 **oder** ...**from** r_1 cross join r_2
- **Verbund:** ...**from** r_1 join r_2 on *Bedingung*
- **Gleichverbund:** ...**from** r_1 join r_2 using (*Attribute*)
- **Natürlicher Verbund:** ...**from** r_1 natural join r_2
- **Vereinigungsverbund** ...**from** r_1 union join r_2
- **Äußerer Verbund:** ...**from** r_1 outer join r_2
  ...**from** r_1 left outer join r_2
  ...**from** r_1 right outer join r_2

r_1

A	B
1	2
2	3

r_2

B	C
3	4
4	5

natural

A	B	C
2	3	4

outer

A	B	C
1	2	$\perp$
2	3	4
$\perp$	4	5

left

A	B	C
1	2	$\perp$
2	3	4

right

A	B	C
2	3	4
$\perp$	4	5

union

A	B	C
1	2	$\perp$
2	3	$\perp$
$\perp$	3	4
$\perp$	4	5

$\Rightarrow$ geringere Fehleranfälligkeit bei Formulierung komplexer Anfragen

Behandlung von Nullmarken

- **Können verschiedene Bedeutungen haben**
  - Wert existiert nicht (Bankverbindung) oder ist undefiniert (Maximum von $\emptyset$)
  - Wert existiert, ist aber unbekannt (Geheime oder verweigerte Daten)
  - Attribut trifft bei diesem Tupel nicht zu (Geburt bei männlichen Patienten)
  - Wert ungültig (Alter eines Rentners ist 2 Jahre)
  - Eingefügter Wert bei outer join oder union join
- **SQL unterscheidet Bedeutungen nicht**
  - Alle Ausdrücke und Vergleiche mit Nullmarken ergeben null (außer is null)
  - null = null liefert null!!
  - Aggregatsfunktionen (außer count) ignorieren Nullmarken
  - Aggregatsfunktionen (außer count) leifern Nullmarken bei leeren Mengen
  - Boolesche Operationen basieren auf einer dreiwertigen Logik

not	
true	false
null	null
false	true

and	true	null	false
true	true	null	false
null	null	null	false
false	false	false	false

or	true	null	false
true	true	true	true
null	true	null	null
false	true	null	false

- **Alternative: Defaultwerte definieren ($\mapsto$ DDL)**

Erweiterungen in SQL-92 (III)

Mengenoperationen positionsweise oder Attributbezogen

- **Vereinigung:** (**select**. . . **from**. . .) **union** (**select**. . . **from**. . .)
- **Durchschnitt:** (**select**. . . **from**. . .) intersect (**select**. . . **from**. . .)
- **Differenz:** (**select**. . . **from**. . .) except (**select**. . . **from**. . .)
- **corresponding-Klausel:**
  - Mengenoperation nur über gemeinsame Attribute der Relationen
  - corresponding by: . . . nur über explizit genannte gemeinsame Attribute
  - z.B. `...from Professoren union corresponding Studenten`
    Menge der Personalausweisnummern von Professoren und Studenten
- **Obermengenprädikat: contains**

```
select K_Name from Konto K1  where
    (select Filiale from Konto K2 where K1.K_Name = K2.K_Name)
       contains
    (select Filiale from Bank where B.Stadt = 'Darmstadt')
```

Alle Kunden, die bei jeder Darmstädter Bank ein Konto haben

- **Mengenoperationen auch innerhalb von Anfragen**

⇒ Vollständig und (fast) orthogonal

Spezielle Konstrukte

- **unique (select... from...):**
  - Test auf Eindeutigkeit einer Relation
- **X match [unique] (select... from...):**
  - Test, ob X (genau einmal) in der Relation vorkommt
- **Fallunterscheidung**
  - **case when** P_1 **then** e_1 ... **when** P_n then e_n else e end
  - liefert einen von mehreren möglichen (konkreten) Werten
  - z.B. innerhalb einer **select**-Klausel

Die **update** Anweisung

update r **set** $A_1 = e_1, .., A_n = e_n$ [**where** P]

- **Änderung von Tupeln einer Basisrelation oder Sicht**
  - In allen Tupeln von r, die P erfüllen, werden die Attributwerte ersetzt
  - Fehlt die **where**-Klausel, so werden alle Tupel verändert
  - (Kompatible) Ausdrücke e_i dürfen Attribut A_i enthalten
  - Auch als Eintupeloperation verwendbar (P fixiert Wert der Schlüsselattribute)
  - **Achtung:** Änderungen können Integritätsbedingungen ($\mapsto$ DDL) verletzen
    (SQL weist lokale Verletzungen zurück, globale Integritätsbedingungen nicht prüfbar!)
  - Reihenfolge der Änderungen ist wichtig
  - **where**-Klausel darf nicht auf r verweisen (sonst $\rightarrow$Anomalien möglich)

Beispiele für Änderungen

1. **update** `Angestellte` **set** `Gehalt=Gehalt*1.05`
2. **update** `Angestellte` **set** `Gehalt=Gehalt*1.05` **where** `Gehalt<5000`
3. **update** `Angestellte` **set** `Gehalt = 9000` **where** `Name = 'Bond'`

0.	Name	Gehalt
	Meyer	3000
	Schulz	3500
	Bond	7200
	Schmidt	4400

1.	Name	Gehalt
	Meyer	3150
	Schulz	3675
	Bond	7560
	Schmidt	4620

2.	Name	Gehalt
	Meyer	3150
	Schulz	3675
	Bond	7200
	Schmidt	4620

3.	Name	Gehalt
	Meyer	3000
	Schulz	3500
	Bond	9000
	Schmidt	4400

Die **delete** Anweisung

delete from r [where P]

- **Löschen von Tupeln aus <u>einer</u> Basisrelation oder Sicht**
  - Alle Tupel von r, die P erfüllen, werden aus r gelöscht
  - Fehlt die where-Klause, so wird die gesamte Relation gelöscht (!!)
  - Auch als Eintupeloperation verwendbar
  - Löschungen können Fremdschlüsselbedingungen ($\mapsto$ DDL) verletzen !!

    Relation mit Fremdschlüssel verweist ins Leere

- **Vermeidung von Anomalien**
  - z.B. `delete from Konto where Saldo < (select avg (Saldo) from Konto)`
    - Vorzeitiges Entfernen würde Durchschnittswert ändern
    - Ergebnis würde von Reihenfolge der Tupel abhängen

  Hier wäre eine statische Optimierung möglich, was normalerweise nicht automatisch geht

  $\Rightarrow$ where-Klausel darf nicht auf r verweisen

  $\Rightarrow$ Alternativ: Tupel als gelöscht eintragen aber erst später entfernen

Die **insert** Anweisung

insert into r $[(A_1, .., A_n)]$ **values** $(k_1, .., k_n)$

insert into r $[(A_1, .., A_n)]$ $(SQL\text{-}Anfrage)$

- **Einfügen von Tupeln in eine Basisrelation oder Sicht**
  - Integritätsbedingungen müssen eingehalten werden
- **Einfügen konstanter Tupel ohne Attributliste**
  - Für alle Attribute müssen konstante Werte angegeben werden
  - Reihenfolge muß Deklaration in DDL entsprechen
  - z.B. `insert into Buch values (4687,'Wissenbanken','3-876','Bibel')`
- **Einfügen konstanter Tupel mit Attributliste**
  - Für genannte Attribute müssen konstante Werte angegeben werden
  - nicht aufgeführte Attribute werden mit Nullwerten (oder Defaults) belegt
  - z.B. `insert into Buch(Invnr, Titel) values (4687,'Wissenbanken')`
- **Einfügen berechneter Tupelmengen**
  - Mit oder ohne Attributliste (gleiche Regelungen!)
  - Berechnete Werte müssen kompatibel sein
  - z.B.
```
insert into Buch(Titel,Autor,Jahr)
       (select Titel,Autor,1996 from Verlage where Jahr>1995)
```
Ergänzen aller Neuerscheinungen der Verlage

QUEL – Query Language

- **DML/DDL auf Basis des Relationentupelkalküls**
  - Aufbau analog zur SQL aber mit deskriptiver Grundlage
  - Orthogonaler Sprachentwurf

- **Muster von QUEL-Anfragen**

  range of t_1 **is** r_1 ... **range of** t_k **is** r_k

  retrieve [**into** s] [**unique**] ([B_1=] e_1, ... [B_n=] e_n) **where** P

  - Deklaration von Tupelvariablen t_i mit Bereichsbegrenzung $t_i \in r_i$
  - Auswahl von Komponenten für die Zielrelation: Datenterme der Form $t_x.A_y$
    - optionale Angabe von Name s und Attributen B_j (für weitere Verwendung)
    - optionale Entfernung von Duplikaten
  - Einschränkende Selektionsformel mit freien Variablen r_i (ohne Quantoren)

- **Keine verschachtelten Queries**
  - Verwende Zwischenrelationen, die durch **into** erzeugt werden

- **Keine Mengenoperationen der Relationenalgebra**
  - Simulation von $\cup$, $\cap$, – durch Zwischenrelationen und Änderungsoperationen

- **Anfragen nicht streng relational vollständig**
  - Vollständig nur mit Zwischenrelationen und Änderungsoperationen

QUEL – weitere Bestandteile

- **Aggregatfunktionen**
  - Notation: . . . F(e **where** P)
  - Als Gruppierung: . . . F(e **by** t.A)
  - Vordefiniert: **sum**, **avg**, **max**, **min**, **count**, **any**
- **Änderungsoperationen**
  - Einfügen von Tupeln in eine Relation s

    range of t_1 **is** r_1 . . . **range of** t_k **is** r_k
    append to s ([B_1=] e_1, . . . [B_n=] e_n) **where** P
  - Ändern von Tupeln

    range of t_1 **is** r_1 . . . **range of** t_k **is** r_k
    replace t([B_1=] e_1, . . . [B_n=] e_n) **where** P
  - Löschen von Tupeln

    range of t_1 **is** r_1 . . . **range of** t_k **is** r_k
    delete t **where** P
- **Statische Einbettung in C möglich**
  - Quel Statements mit '##' kennzeichnen ($\mapsto$ Preprocessing)

QUEL – Beispielanfragen

- *Finde alle Kunden, die ein Konto in der Innenstadtfiliale haben*

  range of t **is** Konto **retrieve** (t.K_Name) **where** t.Filiale = 'Innenstadt'

  SQL: select K_Name from Konto where Filiale='Innenstadt'

- *Finde Namen und Adresse aller Kunden, die einen Kredit haben*

  range of s **is** Kunde **range of** t **is** Kredit

  retrieve (s.K_Name,s.Adresse) **where** s.K_Name = t.K_Name

  SQL: select Kunde.K_Name, Kunde.Adresse from Kredit, Kunde
  where Kredit.K_Name=Kunde.K_Name

- *Bestimme das durschnittliche Saldo aller Innenstadtkonten*

  range of t **is** Konto **retrieve avg** (saldo **where** Filiale = 'Innenstadt')

  SQL: select avg(saldo) from Konto where Filiale='Innenstadt'

- *Finde alle Kunden, die Kredit oder Konto bei der Innenstadtfiliale haben*

  range of x **is** Konto
  retrieve into TMP(Name=x.K_Name) **where** x.Filiale = 'Innenstadt')
  range of s **is** Kredit **append to** TMP(s.K_Name) **where** Filiale = 'Innenstadt'
  range of t **is** TMP **retrieve unique** (t.Name)

  SQL: ...union...

QBE – Query by Example

- **Formorientierte (zweidimensionale) Sprache**
  - Benutzer beschreibt Wünsche durch Beispieleinträge in Tabellen
  - Deskriptiv: Verzicht auf Operatoren und Prozeduren (analog zu PROLOG)
  - Beispielelemente ($\hat{=}$ Bereichsvariablen) gekennzeichnet mit '_' am Anfang
  - Kontrollworte für Anfrage, Einfügen, Löschen, Ändern etc. (Punkt am Ende)
  - Sonstige Einträge sind Konstante
  - Semantik abgestützt auf Relationenwertebereichskalkül
- **Initialisierung durch Aufruf eines Skeletts**

Vorl	V_Bezeichnung	SWS	Semester	Studiengang

  - Benutzer muß nur Relationennamen, nicht aber Attritbutnamen kennen
  - Benutzer trägt beispielhafte Instanzen und Bedingungen in das Skelett ein

Anfragen in QBE (I)

- **Einfache Selektion und Projektion**

  Alle Informatikvorlesungen ab dem siebten Semester

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	P.	P.	>7	Informatik

```
select V_Bezeichnung,SWS from Vorl where Semester>7 and Studiengang=Informatik
```

  – Kontrollwort '**P.**' markiert Ausgabespalte für Ergebnisrelation (Projektion)
    · 'P.' in Relationenspalte entspricht Auswahl aller Attribute (select *)
    · Duplikate werden eliminiert (sonst Spalte mit 'ALL.' kennzeichnen)
  – Bedingungen in Spalten schränken ausgewählte Tupel ein (Selektion)

- **Einfacher Verbund von zwei Relationen**

  Vorlesungen mit mehr als 2SWS, für die 'Datenbanken I' Voraussetzung ist

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	P. _DatenbankenII	P. >2		

Vorl_Voraus	V_Bezeichnung	Voraussetzung
	_DatenbankenII	Datenbanken I

```
select V.V_Bezeichnung,SWS from Vorl V,Vorl_Voraus VV
   where V.V_Bezeichnung=VV.V_Bezeichnung and SWS>2
```

  – Gleiche Beispielelemente verbinden Attribute aus mehreren Relationen
  – Beispielelemente dürfen innerhalb von Ausdrücken erscheinen
  – Beispielelemente müssen in einer Zeile einer Relation gebunden werden.

Anfragen in QBE (II): Komplexe Bedingungen

Alle Vorlesungen für Informatiker und Mathematiker

– **Condition Box**: Explizite Angabe einer Bedingung

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	P.	P.		_St

CONDITIONS
St = Informatik or St = Mathematik

– **Selbstverbund:**

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	P. _VL1 P. _VL2	P. _SWS1 P. _SWS2		Informatik Mathematik

- Kennzeichnung von Bedingungen in mehreren Zeilen derselben Spalte
- Verschiedene Beispielelemente entsprechen Alternativen
- Gleiche Beispielelemente verbinden Attribute aus derselben Relation

Welche Vorlesungen werden in einem Fach in demselben Semester gehört?

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	P. P.		_Dasselbe _Dasselbe	_Informatik P. _Informatik

```
select V1.V_Bezeichnung,V2.V_Bezeichnung,V2.Studiengang from Vorl V1, Vorl V2
   where V1.SWS=V2.SWS and V1.Studiengang=V2.Studiengang
```

QBE Anfragen (III)

- **Definition temporärer Ausgabetabellen**

  Alle Informatikvorlesungen mit ihren Voraussetzungen

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	_DB	_sws	_ab_eins	Informatik

Vorl_Voraus	V_Bezeichnung	Voraussetzung
	_DB	_DBVoraus

Inf_VL	Name	Voraussetzung	SWS	Semester
P.	_DB	P. >2 _DBVoraus	_sws	_ab_eins

  – Temporäre Ausgangstabelle `Inf_VL` übernimmt alle Daten
  – Attributnamen und -reihenfolge neu definiert (SQL-92: abgeleitete Relation)
  – Alle Tupel von `Inf_VL` werden vollständig gedruckt (sonst keine)

- **Negierte Zeilen**

  Vorlesungen mit maximaler Anzahl von Semesterwochenstunden

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
P. ¬		_viele > _viele		

```
select * from Vorl where not exists (select * from Vorl V2 where Vorl.SWS<V2.SWS)
```

  – Negierte Zeilen drücken eine ‘es gibt nicht’ Beziehung aus
  – Kein Ausdruckbefehl in negierten Zeilen erlaubt

QBE Anfragen (IV)

- **Sortierung von Ausgaben**

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
P.			AO(2).	AO(1).

  – Sortierung von Attributwerten (AO=aufsteigend, DO=absteigend)

  – Angabe der Prioritäten in der Sortierreihenfolge in Klammern.

- **Aggragatfunktionen**

  Gesamtzahl aller Semesterwochenstunden von Informatikvorlesungen

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
		P.SUM.ALL. _sem		Informatik

```
select sum(SWS) from Vorl where Studiengang='Informatik'
```

  – Vordefiniert: SUM., AVG., MAX., MIN., CNT.

  – Aggregat über Multimenge (ALL.),

  – Elimination doppelter Elemente auf Wunsch (UN.ALL.)

QBE ist relational vollständig

- **Projektion** $\hat{=}$ Markierung von Spalten mit `P.`
- **Selektion** $\hat{=}$ Vergleich von Spalteneinträgen oder Condition Box
- **Umbenennung** $\hat{=}$ temporäre Ausgabetabelle
- **Produkt** $\hat{=}$ Übernahme aller Spalten in temporäre Ausgabetabelle
- **Verbund**
  $\hat{=}$ Verbindung durch gleiche Beispielelemente in beiden Relationen
- **Vereinigung**
  $\hat{=}$ Übernahme aller Tupel in zwei Zeilen einer temporären Ausgabetabelle
- **Differenz**
  $\hat{=}$ Projektion aus temporärer Produkttabelle mit negierten Zeilen

Ausdrucksweise z.T. komplex, da keine Quantoren (Schachtelung)

Änderungen in QBE

- **Einfügen**

  Ergänze neue Veranstaltungen – konkret und aus Katalogen

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
I.	Datenbanken I	4	5	Wirtschaftsinformatik
I.	_Seminare	2	_fünf	Informatik

Inf_Sem	V_Bezeichnung	Typ	Semester	Dozent
	_Seminar e	Seminar	_fünf	

  – Kontrollwort '**I.**' markiert Neueinfügen

  – Explizite Angabe aller Komponenten oder Berechnung aus anderen Relationen

- **Löschen**

  Entferne alle Informatik-Vorlesungen des Grundstudiums

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
D.			<5	Informatik

  – Kontrollwort '**D.**' markiert Löschung

Änderungen in QBE (II)

- **Ändern von Attributwerten**

  Datenbankenvorlesung um 2 SWS verlängern

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	Datenbanken	_Alt	_fünf	_Informatik
U.	Datenbanken	_Neu	_fünf	_Informatik

CONDITIONS
_Neu = _Alt +2

  – Kontrollwort '**U.**' markiert Änderungen

  – Kurzform ohne Condition Box möglich

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	Datenbanken	_Alt	_fünf	_Informatik
U.	Datenbanken	_Alt+2	_fünf	_Informatik

  – Kurzform mit lokaler Änderung möglich

Vorl	V_Bezeichnung	SWS	Semester	Studiengang
	Datenbanken	U. _Alt+2	_fünf	_Informatik

Relationale Datenbanksprachen im Vergleich

- **SQL**
  - Standardsprache mit relativ hohem Bekanntheitsgrad
  - Operationale Denkweise
  - Kernsprache für computergeübte Benutzer leicht erlernbar
  - Komplexe Anfragen und Datenbeschreibung elegant formulierbar (ab SQL-92)
  - Gut einbettbar in Wirtssprachen
  - Kein Interface für ungeübte Benutzer

  $\Rightarrow$ Unterstützung durch 4GL Systeme (Masken, Schemata, Sprachanweisungen,...
  + Standard-Anwendungsprogramme (Formulargenerator etc)

- **QUEL**
  - Ähnlich zu SQL, aber geringer Verbreitungsgrad
  - Deskriptive Denkweise – für naive Benutzer leicht erlernbar
  - Komplexe Anfragen mühsam

- **QBE**
  - Anschaulich für ungeübte Benutzer
  - Vorbild für Entwurf von Benutzerschnittstellen
  - Vorteilhaft für naive, einfache Operationen
  - Komplexe Anfragen mühsam, programmierte Anwendung kaum möglich

Grundlagen der Datenbanken

Lektion 8

Relationale Datenbanksprachen III: Weitere SQL-Konzepte

1. Datenbeschreibung
2. Datenkontrolle
3. Einbettung in Wirtssprachen

Datenbeschreibung in SQL

- **Anforderungen an eine Relationale DDL**
  - Definition von Datenbankbeschreibungen
  - Schrittweiser Aufbau sinnvoll in der Reihenfolge
    Attribute → Wertebereiche → Relationenschemata
    → Primärschlüssel → Fremdschlüssel
- **SQL-Wirklichkeit**
  - Attribute und Wertebereiche nur lokal für Relationenschema definierbar
  - Schlüssel in SQL-89 ohne IEF nur simulierbar, Fremdschlüssel unmöglich
  - SQL-92 erlaubt explizite Deklaration von Schlüsseln und Fremdschlüsseln
- **SQL-Konstrukte für alle 3 Ebenen des Entwurfs**
  - externe Ebene: create view, drop view
  - konzeptionelle Ebene: create table, drop table, alter table
    SQL-92 zusätzlich: create domain, drop domain, alter domain
  - interne Ebene: create index, drop index, alter index

Sprachkonstrukte für die konzeptionelle Ebene

- **create table R (A_1 W_1 [not null],..., A_n W_n [not null])**
  - Erzeugt leere Basisrelation mit Namen R, Attributen A_i Wertebereichen W_i
  - Ablage der Deklaration im Data Dictionary
  - Mit **not null** gekennzeichnete Attribute dürfen keine Nullwerte erhalten
  - Primärschlüssel sollten nullwertfrei sein

  Erlaubte Datentypen (SQL-89)
  - numerische Typen: `INT`, `SMALLINIT`, `FLOAT(`p`)`, `DEC(`p,q`)`
  - Zeichenketten: `CHAR(`n`)`, `VARCHAR(`n`)`, `BIT(`n`)`, `VARBIT(`n`)`
  - Spezielle Typen: `DATE`, `TIME`, `TIMESTAMP`

```
create table Bücher (ISBN CHAR(10) not null,
                     Titel VARCHAR(200),
                     Verlagsname VARCHAR(30))
```

- **alter table R add A W** (kein **not null**!)
  - Ergänze neues Attribut A mit Wertebereich W zur Basisrelation R
  - Veränderung der Deklaration im Data Dictionary
  - Erstmalige Anfragen erweitern Tupel um Nullwert (Änderungen wie üblich)
- **drop table R**
  - Löscht Basisrelation R, alle Sichten darauf und alle Indexstrukturen
  - Entfernt Deklaration aus Data Dictionary

Erweiterungen in SQL-92

- **Deklaration von Primär und Fremdschlüsseln**

```
create table Bücher (ISBN CHAR(10),
                     Titel VARCHAR(200),
                     Verlagsname VARCHAR(30),
                     primary key (ISBN),
                     foreign key (Verlagsname)
                         references Verlage (Verlagsname)
                    )
```

  - **primary key**-Deklaration enthält implizites **not null**
  - **references**: Fremdrelation und ihre referenzierten Primärschlüssel

- **Erweitertes Datentypkonzept**
  - Zusätzliche Datentypen
  - Deklaration von Defaultwerten: A W **default** value
  - Deklaration benutzerdefinierter Datentypen durch **create domain**

    z.B. **create domain** `GEBIETE VARCHAR(20)` **default** `'Informatik'`
  - Änderung / Löschen mit **alter domain** und **drop domain**

Erweiterungen in SQL-92 (II)

- **Lokale und globale Integritätsbedingungen**

```
create domain GEBIETE VARCHAR(20) default 'Informatik'
  check (values in ('Informatik', 'Mathematik', 'Jura'))
create table Buch_Versionen (ISBN CHAR(10),
                (ISBN CHAR(10),
                 Auflage SMALLINT check (Auflage > 0),
                 Jahr INT check (Jahr between 1800 and 2020),
                 Seiten INT check (Seiten > 0),
                 Preis DEC(8,2) check (Preis ≤ 250),
                 primary key (ISBN),
                 foreign key (ISBN) references Buch (ISBN),
                 check ( (select sum(Preis) from Buch_Versionen)
                       < (select sum(Budget) from Lehrstühle) ) )
```

- **alter table Kommando flexibler**
  - Ergänzen neuer Attribute mit Defaultwert und Integritätsbedingung
  - Veränderungen (alter) der Defaultwerte bestehender Atrribute
  - Löschen von Attributen (ggf. zugehörige Sichten und Integritätsbedingungen)
- **drop table mit Sicht- und Integritätskontrolle möglich**

Definition externer Sichten

- **create view V $[(A_1,\ldots,A_n)]$ as (select ... from ...)**
  - Definiere virtuelle Relation V durch eine Folge von Anfrageoperationen
  - Anfrage wird erst bei Aufruf durchgeführt

```
create view Kundschaft as
   (select Filiale, K_Namec  from Konto)
     union
   (select Filiale, K_Name  from Kredit)
```

  $\Rightarrow$ Vereinfachung von Anfragen

  \+ Strukturierung der Datenbank spezifisch für Benutzerklassen

  \+ logische Datenunabhängigkeit für Anwendungen

  \+ Zugriffsbeschränkungen (Datenschutz) möglich

- **Abbildungsprozeß auch über mehrere Stufen möglich**
  - aber keine Schachtelung von Aggregatfunktionen und Gruppierungen

- **drop view V**
  - Entferne Sichtdeklaration aus dem Data Dictionary

Auswirkung externer Sichten

- **Bei Anfragen gleichwertig zu Basisrelationen**
- **Änderungsoperationen nicht immer korrekt**
  - `insert into Kundschaft values ('Innenstadt', 'Schulze')`
    - Einfügen in `Konto` oder `Kredit`?
    - Werte für unsichtbare Attribute fehlen (Nullwert/Default?)

  $\Rightarrow$ Defaultbehandlung entwerfen (Datenbankabhängig)

  - Bei Selektionssichten können Tupel bei Änderungen unsichtbar werden

  $\Rightarrow$ Sperren oder explizit zulassen (Datenbankabhängig)

  - Bei berechneten Sichten sind Sichtänderungen nicht sinnvoll umsetzbar
    - Aggregat, Gruppierung, ...

  $\Rightarrow$ Verbot von Änderungen in solchen Fällen

Indexierung: Bezug zur internen Ebene

- **create** [**unique**] **index** i **on** R (A_1 [**asc**|**desc**],. . . ,A_n [**asc**|**desc**])
  - Definiert Zugriffspfad auf Relation R über Werte der Attribute A_i
  - **asc** (Default) / **desc**: Attributwerte im Index auf- bzw. absteigend geordnet
  - Realisierung z.B. durch B-Bäume
  - **unique**: Indexwert darf nicht doppelt vorkommen
    - wird bei Änderungen kontrolliert (ggf. Verweigerung der Operation)

    $\Rightarrow$ Simulation eines Schlüssels $\{A_1,\ldots,A_n\}$

  z.B. `create unique index BuchIndex on Bücher (ISBN asc)`

- **alter index** i **on** R (A_1 [**asc**|**desc**],. . . ,A_n [**asc**|**desc**])
  - Nachträgliche Änderung eines bestehenden Index

- **drop index** i: Löschen des Index i

Indexierung wurde in SQL-92 entfernt

Deklaration von Integritätsbedingungen

- **Lokal definierte Integritätsbedingungen (SQL-92)**
  - Typrestriktion durch Zuordnung von Wertebereichen (ggf. mit Defaultwerten)
  - **not null**: Verbot von Nullwerten
  - **primary key** / **foreign key** . . . **references**: Deklaration von Primär-/Fremdschlüsseln
  - **check**: Attributspezifische aussagenlogische Bedingung

- **create assertion** a **check** (P)
  - Global definierte Integritätsbedingung – P beliebige boolesche Aussage
  - Speicherung der Bedingung mit Namen im Data Dictionary

```
create assertion Preise
      check ((select sum(Preis) from Buch) < 10000)
create assertion Billig
      check (not exists(select * from Buch where Preis >250))
```

- **Überwachungsparameter**
  - **on update | on delete**: Überprüfung nur bei Änderung (bzw. Löschung)
  - **immediate | deferred**: sofortige/verzögerte Kontrolle (Ende der Transaktion)
  - **cascade | set null | set default | no action**: Reaktion auf Verletzung
    - Weiterreichen, Löschen, Defaultwert einsetzen, ohne Reaktion
  - Trigger: automatische Folgeänderungen zur Herstellung von Integrität (SQL3)

+ **Zugriffsrechte, Transaktionskontrolle, Fehlerbehandlung**

Beurteilung von SQL als Datenbeschreibungssprache

- **Vorteile**
  - Gleiche Sprachregelung wie in der Datenmanipulationssprache
  - Administrator kann Vorentwurf erstellen und schrittweise optimieren
  - Schemaänderungen in bestehenden Systemen möglich
  - Vollständige Umsetzung von Schlüssel- und Integritätskonzepten in SQL-92
- **Problematisch**
  - Mangelnde Unterstützung beim Entwurf
  - sehr eingeschränkte Änderungsmöglichkeiten
    (z.B. keine Erweiterung eines Wertebereichs von `VARCHAR(20) auf VARCHAR(30)`)
  - Mißbrauch von Indexstrukturen zur Simulation von Schlüsseln in SQL-89
  - Inkonsequente Umsetzung des Wertebereichskonzepts in SQL-92
    benutzerdefinierte Typen aber keine benutzerdefinierten Operationen darauf)

  $\Rightarrow$ SQL3 Projekt

Programmierte Steuerung von Datenbankoperationen

- **Cursor-Konzept erforderlich**
  - Konventionelle Programmiersprachen unterstützen keine Relationen
  - Ein-/Ausgabe (Programm↔Datenbank) muß tupelweise verarbeitet werden
  - Kommunikation über (bewegliches) Cursor-Element
    - Cursor zeigt auf ein aktuelles Element der zu bearbeitenden Tabelle
- **declare C cursor for (select ... from ...)**
  - Deklariert Cursor-Variable C für die angegebene Zielrelation
  - Optionale Klausel **order by** kennzeichnet Reihenfolge der Abarbeitung
  - Optionale Klausel **for update of** limitiert änderbare Attribute

```
declare AktBuch cursor for
  select ISBN,Titel,Verlagsname from Bücher
     where Verlagsname='Thompson'
  for update of ISBN,Titel
```

- **Zugriff auf Cursor-Wert durch fetch-Anweisung**
  - Liefert aktuelles Tupel in Puffervariablen
  - Setzt Cursor auf nächstes Element der Tabelle
    (SQL-92 erlaubt beliebiges Navigieren der Cursor-Position)

Statische Lösung: Embedded SQL

- **Verwendung eines Precompilers**
  - Kennzeichnung von SQL-Anweisungen durch Schlüsselwort **exec sql**
  - Umwandlung von SQL-Anweisungen in Prozeduraufrufe bei der Compilation

  $\Rightarrow$ statisch: SQL-Anweisungen müssen zur Übersetzungszeit feststehen

- **Abgleich zwischen Datenbank und Programmiersprache**
  - **exec sql connect** r: Kontakt zur Datenbank r öffnen
  - **exec sql disconnect** r: Kontakt zur Datenbank r lösen
  - **exec sql declare** r **table** (...): Deklaration von r an Compiler bekanntgeben
  - **exec sql declare section** v_1 W_1; ... v_n W_n **exec sql end declare section**:
    Deklaration von Programmvariablen, für Verwendung in SQL (Name ':v_i')

Embedded SQL – Datentransfer

- **Datentransfer vom Progamm zur Datenbank**
  - Verwendung von Programmvariablen in SQL-Anweisungen als Konstante

```
exec sql insert into Bücher(Invnr,Titel)
                values (:NeuInvNr,'Wissensbanken')
```

- **Datentransfer von der Datenbank zum Programm**
  - into-Klausel bei Anfragen, die genau ein Tupel liefern
  - z.B.

```
exec sql select Invnr, Titel into :AktINvNr,:Titel
                    from Bücher where ISBN=:SuchISBN
```

  - Cursor und **fetch**-Kommando bei Tupelmengen

```
exec sql open AktBuch;
exec sql fetch AktBuch Into :ISBN,:Titel,:Verlagsname;
exec sql close AktBuch
```

- **Programmierte Änderung und Löschung**
  - Zugriff auf aktuelles Cursor-Element über **current of**
  - z.B. **exec sql delete from** Bücher **where current of** AktBuch

Dynamisches SQL

- **Konstruktion von SQL-Anweisungen zur Laufzeit**
  - Anfrage wird nach Bedarf erzeugt (z.B. Menü)
  - Programm muß Anfragen wie Strings behandeln
  - Strings müssen interpretiert werden
  - Anfrageoptimierung erst zur Programmlaufzeit möglich
- **Zusätzliche SQL-Kommandos**
  - **exec sql declare** v **statement**: Kommandovariable deklarieren
  - **exec sql prepare** v **from** string: Umwandlung String $\mapsto$ SQL-Kommando
  - **exec sql execute** v **using** :v_1...:v_n:
    - Ausführung des SQL-Kommandos mit Programmvariablen als Parametern

```
exec sql declare Anfrage statement
        ⋮
AnfrageString := 'DELETE FROM Buch WHERE ISBN=? AND Titel=?';
exec sql prepare Anfrage from AnfrageString;
exec sql execute Anfrage using :LöschISBN, :LöschTitel;
```

Bisher noch nicht genormt

SQL3 – der nächste Schritt

- **Objektorientierte Erweiterungen**
  - Abstrakte Datentypen (incl. typspezifischer Operationen)
  - Hierarchien von ADT's, Vererbung, Dynamisches Binden
  - Komplexe Datentypen wie Multimenge, Liste, Menge

  keine konsequent objektorientierte Konzeption!
- **Erweiterte Deklaration von Verbunden**
  - Spezifikation über Primär und Fremdschlüssel
- **Begrenzt rekursive Anfragen**
  - Rekursiv definierte Vereinigung von Tabellen (Transitive Hülle, ...)
- **Aktive und temporale Konzepte**
- •
- •
- •

Mehr als 1000 Seiten Beschreibung

SQL Resümee

- **Großes Spektrum an Implementierungen**
  - für PC bis Großrechner
  - Funktionsumfang und Einhaltung der Standards variierend
- **Vorteile**
  - **Einfachheit**: leicht erlernbar, einheitliche Syntax, DML und DDL einheitlich
    Anfragesprache und eingebettete Sprache, Sichtkonzept
    $\Rightarrow$ Für Laien leicht erlernbar
  - **Mächtigkeit**: hohes Auswahlvermögen, Aggregatfunktionen, Sortierung
    Flexibles Autorisierungskonzept, Integrierter Datenbeschreibungskatalog
  - **Datenunabhängigkeit**: physisch (erst SQL-92), interne Leistungsoptimierung
    z.T. logische Unabhängigkeit durch Sichtkonzept
- **Problematisch**
  - Mangelnde Orthogonalität
  - Folgenschwere Fehlermöglichkeiten: Reihenfolgeabhängigkeit,
    irrtümliches Löschen der gesamten Relation,...
  - Keine (einfache) formale Definition
  - Unklare Semantik von Nullwerten
  - Keine globalen Integritätsbedingungen in SQL-89
  - Stark zunehmende Komplexität bei SQL-92/SQL3

Grundlagen der Datenbanken

Lektion 9

Entwurf relationaler Datenbanken

1. Ziele
2. Abbildung von ERM-Modellen in RM-Modelle
3. Entwurfstheorie
   - Funktionale Abhängigkeiten
   - Normalformen
   - Entwurfsverfahren für redundanzarme Datenbanken

Entwurfsziel: Verhindere unerwünschte Nebeneffekte

- **Redundanz**
  - Mehrere Tupel müssen dieselbe (Teil-)Information enthalten

  $\Rightarrow$ Speicherplatzverschwendung + Konsistenzproblemen bei Updates
- **Nullwerte**
  - Relationen enthalten Attribute, zu denen nicht immer ein Wert existiert
  - Nullwerte müssen eingefügt werden

  $\Rightarrow$ undefinierte Situation bei Anfragen möglich
- **Implizite Darstellung von Information**
  - Daten nur als Teilinformation einer Relation erhältlich
    - z.B. Bankfiliale, Stadt, Telephonnumer als Teil der Konto-Relation

  $\Rightarrow$ Löschen eines Tupels kann zu Verlust von Daten (über eine Filiale) führen
- **Ungeschickte Zerlegung in Teilrelationen**
  - Verbund der Teilrelationen erzeugt unerwünschte Tupel ('Fremdtupel')

  z.B. Zerlegung von Kredit(K#,Filiale,K_Name,Betrag)

  in KreditInfo(K#,Filiale,K_Name) und KundenInfo(K_Name,Betrag)

  $\mapsto$ Beim Verbund tauchen viele Kreditbeträge unter einer Kreditnummer auf

Entwurfstheorie

- **Grundlage für Entwurf sicherer DB-Schemata**
  - Übersichtlichkeit und Leichte Handhabbarkeit
  - Vermeidung von Redundanz und potentiellen Inkonsistenzen
    - Änderungs-, Einfüge- und Löschanomalien

  $\Rightarrow$ Theoretische Definition von Normalformen für Relationenschemata

- **Abbildung von ER-Modellen in Relationenschemata**
  - liefert Erstentwurf, Attribute und funktionale Abhängigkeiten

  + Normalisierung von Relationenschemata
  - Verbessere gegebenen Entwurf durch Betrachtung funktionaler Abhängigkeiten
  - Dekomposition von Relationenschemata in kleinere Teile

  $\vee$ Synthese von Relationenschemata
  - Erzeuge Relationenschemata aus Attributen und funktionalen Abhängigkeiten
  - Ziel: theoretisch 'optimales' Gesamt-Schema ('Dritte Normalform')
  - ggf. weitere Dekomposition durch Betrachtung komplexerer Abhängigkeiten

Darstellung von Entity-Relationship Modellen

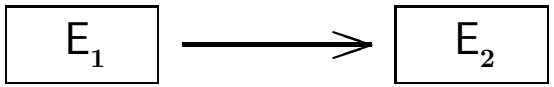
- **Grundprinzipien**
  - Entity-Typen und Beziehungstypen durch Relationenschemata dargestellt
  - Kardinalitäten werden durch Wahl der Schlüssel ausgedrückt
  - Schemata für Entity- und Beziehungstypen werden manchmal verschmolzen
  - Fremdschlüsselbeziehungen regeln Zusammenhänge zwischen Relationen
- **Entity-Typen $\mathbf{E(A_1{:}D_1,\ldots,A_m{:}D_m)}$**
  - einfaches Relationenschema $\{A_1,\ldots,A_m\}$ mit Namen E
  - Schlüssel werden übernommen, Primärschlüssel K_E ausgewählt
- **Entity-Typen E mit strukturiertem Attribut A:list(D)**
  - Relationenschema für E enthält A nicht
  - Zusätzliches Relationenschema $K_E \cup \{A\}$ für Darstellung der Attributliste

ERM-Darstellung: Beziehungen und Schlüssel

- **Beziehungstypen $\mathbf{R(E_1,..,E_n;\ A_1{:}D_1,\ldots,A_m{:}D_m)}$**
  - Neues Relationenschema $K_{E_1} \cup .. \cup K_{E_n} \cup \{A_1,\ldots,A_m\}$ mit Namen R
  - Schlüsselattribute K_{E_i} werden ggf. disjunkt umbenannt
  - (Primär)schlüssel $K_{E_1} \cup .. \cup K_{E_n}$ (Allgemeinfall)
    - bei 0/1:n Beziehung zwischen E_1 und E_2: K_{E_2} wird alleine (Primär)schlüssel
    - bei 0/1:1 Beziehung: Schlüsselmenge $\{K_{E_1}, K_{E_2}\}$, wähle Primärschlüssel
  - Weise die K_{E_i} als Fremdschlüssel für die E_i aus
- **Echte funktionale (1:n) Beziehung $\mathbf{R(E_1,E_2)}$**
  - Integriere E_2 vollständig in das Relationenschema für R
  - Schema für E_2 kann entfallen

  Alternativ: Ergänze E_1 um Schlüsselattribute von E_2 (Fremdschlüssel)
- **Echte 1:1 Beziehung $\mathbf{R(E_1,E_2)}$** ohne Zusatzattribute
  - Integriere E_1 und E_2 vollständig in das Relationenschema für R

ERM Darstellung: Spezialfälle

- **IS_A-Beziehung** 

  - Kein eigenes Relationeschema für die Beziehung
  - Ergänze Schema für E_1 um Primärschlüsselattribute K_{E_2} (ohne Umbenennung)
  - Wähle Primärschlüssel für E_1, weise K_{E_2} als Fremdschlüssel aus

  $\Rightarrow$ speichereffizient, aber aufwendige Suche und Aktualisierung ($\mapsto$ Alternativen?)

- **Rekursive 1:1 Beziehungen R(E,E)**
  - einer der beiden Primärschlüssel muß umbenannt werden (Rollennamen!)
  - z.B. verheiratet(Frau:PERSON, Mann:PERSON) $\mapsto$ PERSON(...,Gatte)

- **Gleichartige 1:n Beziehungen $R(E_1,E_2)$ und $R(E_1,E_3)$**
  - Nicht adäquat modellierbar: 2 Alternativen mit Schwachstellen
    - Separate Modellierung erlaubt globale Inkonsistenzen
    - Beziehung $R(E_1,E)$ zu neuer Generalisierung E von E_2 und E_3:
      Problematisch, wenn bereits Generalisierung E_0 von E_2 und E_3 existiert

- **Generalisierung und Partitionierung**
  - nicht mit Schlüsseln und Fremdschlüsseln alleine modellierbar

Wichtigste Integritätsbedingung zwischen Attributen

- **$X \to Y$ 'Y (direkt) funktional abhängig von X':**
  - Für jeden Wert der Attribute X existiert genau ein Wert der Attribute Y
  - $\forall t_1, t_2 \in r.\ t_1(X) = t_2(X) \Rightarrow t_1(Y) = t_2(Y)$ sinnvoll wenn $X, Y \subseteq R$ (Schema von r)
  - 'Relation r genügt der FD $X \to Y$' (FD $\hat{=}$ Funktionale Abhängigkeit)
  - Es gilt $X \to Y$, falls $Y \subseteq X$ (triviale FD)
  - Kurzschreibweise: $AB \hat{=} \{A,B\}$, $XY \hat{=} X \cup Y$, (A,B Attribute, X,Y Attributmengen)
- **FD-Menge über $\mathcal{U}$**
  - Menge $\mathcal{F} = \{X_1 \to Y_1, \ldots, X_n \to Y_n\}$ mit $\mathbf{ATTR(\mathcal{F})} = \bigcup X_i \cup \bigcup Y_i \subseteq \mathcal{U}$
  - Menge von FD's, die nur aus $\mathcal{U}$-Attributen bestehen
  - Notation $\mathcal{R} = (R, \mathcal{F})$: erweitertes Relationenschema $(R, \mathcal{B}_\mathcal{F})$
    - mit Bedingungen $\mathcal{B}_\mathcal{F} = \{b_f \mid f \in \mathcal{F} \wedge (b_f(r) \Leftrightarrow r \text{ genügt } f)\}$
- **Gute DB-Schemata müssen FD's berücksichtigen**
  - FD's erfassen semantische Interpretation des zu modelliernden Weltausschnitts
  - Nicht modellierte Abhängigkeiten führen zu Anomalien
  - Normalformen garantieren adäquate Repräsentation

Funktionale Abhängigkeiten und Schlüssel

- **Y voll funktional abhängig von X:**
  - Y nur von X, aber nicht von einer echten Teilmenge von X funktional abhängig
  - $X \rightarrow Y \ \wedge \ \forall A \in X.\ \neg(X \backslash A \rightarrow Y)$
  - andernfalls Y partiell funktional abhängig von X
- **Schlüssel K eines Schemas R als Spezialfall**
  - Volle funktionale Abhängigkeit $K \rightarrow R$
  + In einer Relation $r \in R$ taucht jeder K-Wert maximal einmal auf
  - Schlüsseleigenschaften verhältnismäßig leicht zu überwachen
- **Schlüsselabhängigkeiten für R**
  - Menge $\mathcal{F}_{\mathcal{K}} = \{K \rightarrow R \mid K \in \mathcal{K}\}$, wobei $\mathcal{K}$ Menge aller Schlüssel von R

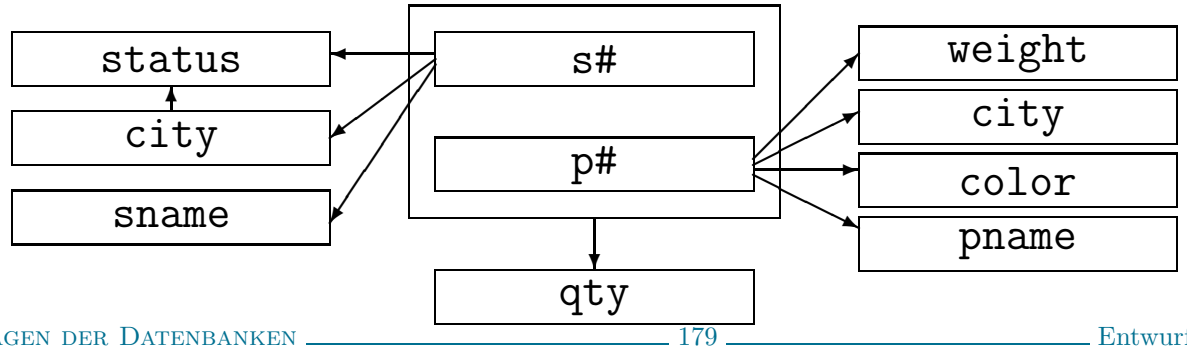
$\Rightarrow$ **Entwurfsaufgabe**

- Gegeben Menge $\mathcal{U}$ on Attributen, FD Menge $\mathcal{F} = \{X_1 \rightarrow Y_1, \ldots, X_n \rightarrow Y_n\}$
- Entwerfe (erweitertes) Datenbankschema $\mathcal{S} = \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ mit $\mathcal{R}_i = (R_i, \mathcal{K}_i)$ das $\mathcal{F}$ durch Schlüsselabhängigkeiten repräsentiert und Anomalien vermeidet

Miniwelt mit Lieferanten, Teilen, und Lieferungen

- Lieferanten (supplier) haben
  - Lieferantennummer (`s#`), -name (`sname`), -status(`status`), -standort (`city`)
  - Der Name ist nicht eindeutig, die Nummer ist eindeutig
- Teile (parts) haben
  - Teilenummer (`p#`), -name (`pname`), -farbe (`color`), -gewicht (`weight`) und -lagerort (`city`)
  - Jedes Teil hat eine feste Farbe und wird nur an einem Ort gelagert
- Lieferungen (supplied parts) haben
  - Lieferantennummer (`s#`), Teilenummer (`p#`), Liefermenge (`qty`)
- Ermittelte funktionale Abhängigkeiten *interpretationsabhängig!*
  - {`s#`→`sname`, `s#`→`status`, `s#`→`city`, `city`→`status`, `p#`→`pname`, `p#`→`color`, `p#`→`weight`, `p#`→`city`, `s#`,`p#`→`qty`}

Graphisch:



Überprüfung Funktionaler Abhängigkeiten

FD's müssen in ihrer Gesamtheit betrachtet werden

- **$\mathcal{F} \models f$: 'f folgt aus der FD-Menge $\mathcal{F}$'**
  - Jede Relation, welche die Abhängigkeiten aus $\mathcal{F}$ erfüllt, erfüllt auch f
  - $\forall r \in \mathrm{SAT}_R(\mathcal{F}).\ r \in \mathrm{SAT}_R(f)$ wobei R=ATTR($\mathcal{F}$) (semantischer Begriff)
- **$\mathcal{F}^+$: 'Hülle von $\mathcal{F}$'**
  - Menge aller implizit und explizit gegebenen funktionalen Abhängigkeiten
  - $\mathcal{F}^+ = \{X \rightarrow Y \mid X,Y \subseteq \mathrm{ATTR}(\mathcal{F}) \wedge \mathcal{F} \models X \rightarrow Y\}$
- **$\mathcal{F} \equiv \mathcal{G}$: '$\mathcal{F}$ überdeckt $\mathcal{G}$'**
  - Gilt gdw. $\mathcal{F}^+ = \mathcal{G}^+$ bzw. $\forall g \in \mathcal{G}.\ \mathcal{F} \models g \wedge \forall f \in \mathcal{F}.\ \mathcal{G} \models f$
  - $\mathcal{F}$ und $\mathcal{G}$ sind semantisch äquivalent ($\mathcal{G}$ überdeckt auch $\mathcal{F}$)

⇒ **Entwurfsziel: Bestimme Schlüsselmenge $\mathcal{K}$ mit $\mathcal{F} \equiv \mathcal{K}$**
- Benötigt Testverfahren für '$\mathcal{F} \models X \rightarrow Y$' (bzw. $\mathcal{K} \models f$) Membership-Problem
- Semantischer Test von '$X \rightarrow Y \in \mathcal{F}^+$' ist i.a. exponentiell ($\mathcal{F}^+$ zu groß)
- Teste stattdessen '$Y \in X_F^+$', wobei $X_F^+ = \{A \mid X \rightarrow A \in \mathcal{F}^+\}$
  - Es ist $X_F^+ = \bigcup_i X_F^i$, wobei $X_F^0 = X$, $X_F^{i+1} = X_F^i \cup \{A \mid \exists Z \subseteq X_F^i.\ Z \rightarrow A \in \mathcal{F}\}$ (Stop bei $X_F^{i+1} = X_F^i$)

Kalküle zur Überprüfung funktionaler Abhängigkeiten

- **Ableitungskalkül**
  - Menge von Ableitungsregeln $\kappa = \{F_1 \vdash f_1, \ldots, F_n \vdash f_n\}$
- **Ableitbarkeit $\mathcal{F} \vdash f$:** (syntaktischer Begriff)
  - f ist aus $\mathcal{F}$ in endlich vielen Schritten mit Regeln aus κ ableitbar
- **Armstrongs Axiome**

A1 : $\emptyset \vdash X \rightarrow Y$, falls $Y \subseteq X$	Reflexivität
A2 : $\{X \rightarrow Y\} \vdash XW \rightarrow YZ$, falls $Z \subseteq W$	Erweiterung
A3 : $\{X \rightarrow Y, Y \rightarrow Z\} \vdash X \rightarrow Z$	Transitivität
R4 : $\{X \rightarrow Y, X \rightarrow Z\} \vdash X \rightarrow YZ$	Additivität
R5 : $\{X \rightarrow Y\} \vdash X \rightarrow Z$, falls $Z \subseteq Y$	Projektivität
R6 : $\{X \rightarrow Y, WY \rightarrow Z\} \vdash XW \rightarrow Z$	Pseudo-Transitivität

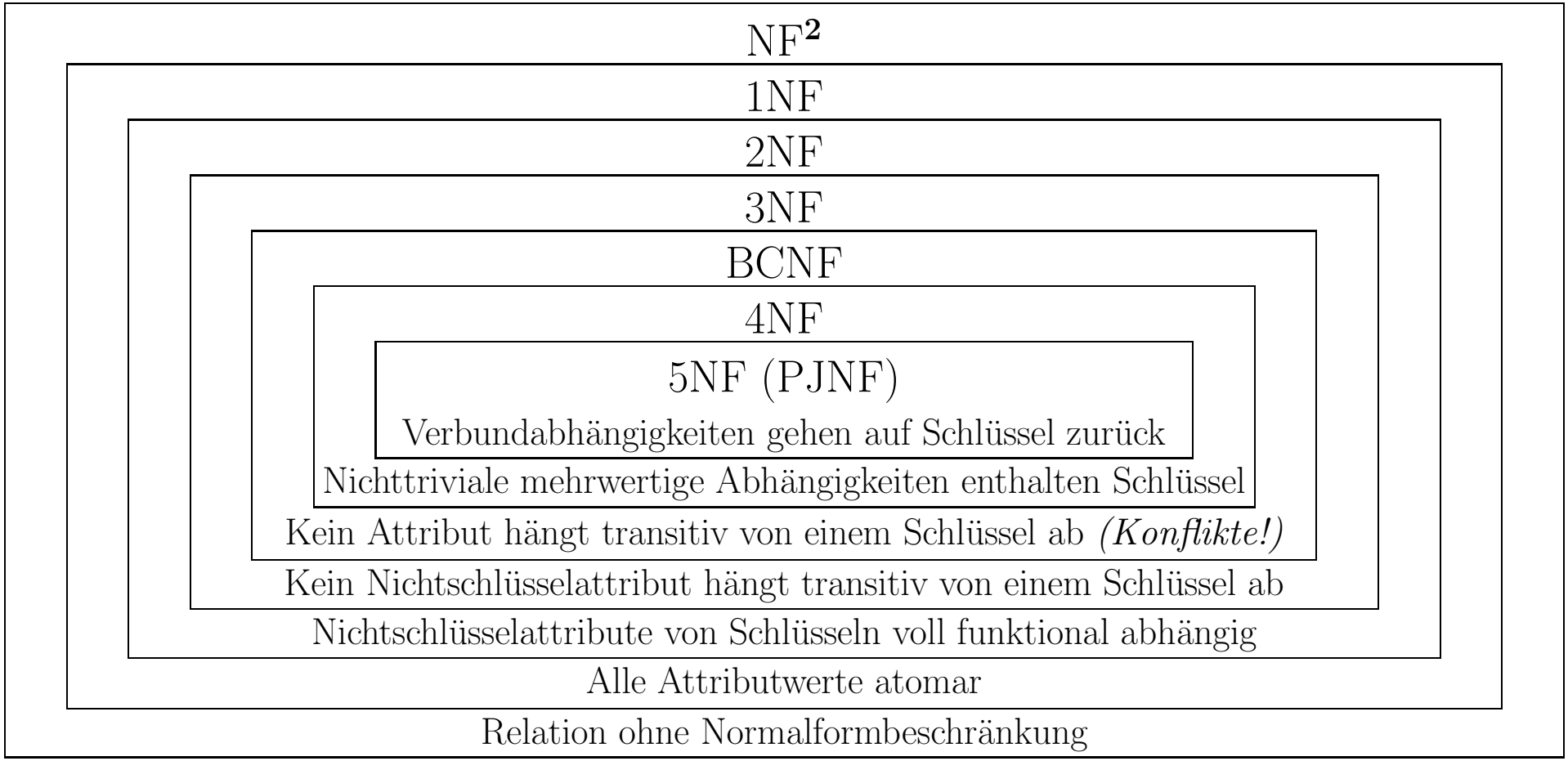
- **Kalkülziel: $\mathcal{F} \vdash f \Leftrightarrow \mathcal{F} \models f$**
  - $\vdash$ ist syntaktische Simulation des semantischen Folgerungsbegriffes
  - κ gültig: $F_i \models f_i$ folgt aus $F_i \vdash f_i$ (was ableitbar ist, folgt semantisch)
  - κ vollständig: aus $\mathcal{F} \models f$ folgt $\mathcal{F} \vdash f$ (was semantisch gilt, kann auch abgeleitet werden)
  - κ unabhängig: Keine echte Teilmenge von κ ist vollständig (keine Regel überflüssig)

Funktionale Abhängigkeiten: Anwendungsbeispiele

- Folgerungen von $\mathcal{F}_2 = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
  - $\mathcal{F}_2 \vdash A \rightarrow H$ (A3 mit $\{A \rightarrow B, B \rightarrow H\}$)
  - $\mathcal{F}_2 \vdash CG \rightarrow HI$ (R4 mit $\{CG \rightarrow H, CG \rightarrow I\}$),
  - $\mathcal{F}_2 \vdash AG \rightarrow I$ (A2: $\{A \rightarrow C\} \vdash AG \rightarrow CG$, A3: $\{AG \rightarrow CG, CG \rightarrow I\} \vdash AG \rightarrow I$)
- Hülle von $\mathcal{F}_1 = \{A \rightarrow B, B \rightarrow C\}$
  - triviale FD's: $\{A \rightarrow A, B \rightarrow B, C \rightarrow C, AB \rightarrow A, AB \rightarrow B, AB \rightarrow AB, AC \rightarrow A, \ldots\}$
  - mit A3: $\{A \rightarrow C, AB \rightarrow C, AC \rightarrow B\}$
  - mit R4: $\{A \rightarrow AB, A \rightarrow AC, A \rightarrow BC, A \rightarrow ABC, B \rightarrow BC, AB \rightarrow AC, AB \rightarrow ABC, AC \rightarrow AB, AC \rightarrow ABC$
- Ermittelung von Schlüsselkandidaten
  - Gegeben R=$\{A,B,C,G,H,I\}$ und $\mathcal{F}_2 = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$
  - Für alle $X \subseteq R$ prüfe $X \rightarrow R$ durch Test $X_F^+ = R$, beachte Minimalität
  - Attribute, die nur rechts stehen (H,I), *können nicht* zu einem Schlüsselkandidaten gehören
    - Einelementige Mengen: A^0=A, A^1=ABC, A^2=ABCH, A^3=ABCH$\neq R$ (alle anderen analog)
    - Zweielementige: AG^0=AG, AG^1=ABCG, AG^2=ABCGHI=R $\mapsto$ AG ist Schlüssel
  - Attribute, die nur links stehen (A,G), *müssen* zu jedem Schlüsselkandidaten gehören

  $\Rightarrow$ Minimalitätsbedingung verbietet Betrachtung größerer Attributmengen

Normalformen und Dekompositionsanforderungen



- Verbundtreue: Alle Originalrelationen durch Verbund wiederherstellbar
- Abhängigkeitstreue: Alle Abhängigkeiten durch Schlüssel repräsentiert
- Minimalität: Keine kleinere Anzahl von Relationenschemata reicht aus

Erste Normalform (1NF)

- **Relationenschema $\mathcal{R}$ ist in erster Normalform**

  = R enthält ausschließlich atomare Attribute $(\mathcal{R}=(\mathrm{R},\mathcal{K}))$

  z.B. `Liefert(s#, sname, status, city, p#, qty)`
  `Teile(p#, pname, color, weight, city)`

- **Grundvoraussetzung für Implementierbarkeit**
  - Strukturierte Attribute nur unsauber modellierbar ($\mapsto$ Spezialsysteme für NF^2)
  - Abhängig von Interpretation der Wertebereiche
    - `Autor = {Heuer,Saake}` wird interpretiert als Menge $\mapsto$ keine 1NF
    - `Autor = Reich-Ranitzki` wird interpretiert als atomarer Wert $\mapsto$ 1NF

- **Transformation in 1NF**
  - Ersetze strukturiertes Attribut `set(A)` durch einfaches Attribut `A`
  - Erzeuge für jede mengenwertige Tupelkomponente eine Menge von Tupeln
  - Probleme: Darstellung der leeren Menge?
    Erzeugte Relation enthält sehr viele Redundanzen

Zweite Normalform (2NF)

- **Problem: partielle funktionale Abhängigkeiten**
  - `Liefert(s#,sname,status,city,p#,qty)` hat Schlüssel {`s#,p#`}
  - Lieferantendaten nur einfügbar, wenn tatsächliche Lieferungen stattfinden
  - Löschen der letzten Lieferung eines Lieferanten entfernt alle seine Daten
  - Konsistente Änderung von Lieferantendaten schwierig (Redundanz)
  - Grund: Attribute `sname,status,city` hängen nur von `s#` ab
- **Relationenschema $\mathcal{R}$ ist in zweiter Normalform**

  $= \mathcal{R}$ ist in erster Normalform

  $+$ Nichtschlüsselattribute sind von allen Schlüsseln voll funktional abhängig

  $\Rightarrow$ keine Abhängigkeiten einzelner Attribute von Teilen eines Schlüssels

  $\Rightarrow$ implizite Verschärfung des Minimalitätsbegriffs
- **Transformation in 2NF**
  - Bilde neue Relation aus Teilschlüssel und davon abhängigen Attributen
  - Entferne abhängige Attribute aus ursprünglicher Relation

  z.B. `Lieferanten(s#,sname,status,city)` + `Lieferungen(s#,p#,qty)`

Dritte Normalform (3NF)

- **Problem: Transitive Abhängigkeit**
  - Attribut `status` in `Lieferanten` funktional abhängig von `city`
  - Attribut `city` ist selbst kein Schlüssel

  $\Rightarrow$ Statuscode einer Stadt nur über Lieferanten erreichbar ($\mapsto$ analoge Anomalien)
- **A transitiv abhängig von X:**
  - A hängt indirekt funktional von X ab

  $\hat{=} \exists Y \subseteq R.\ X \rightarrow Y \wedge Y \rightarrow A \wedge \neg(Y \rightarrow X) \wedge A \notin X \cup Y$
- **Relationenschema $\mathcal{R}$ ist in dritter Normalform**

  = Kein Nichtschlüsselattribut hängt transitiv von einem Schlüssel ab

  $\hat{=}$ Wenn A (nichttrivial) funktional von X abhängt, muß X ein Schlüssel sein

  $\hat{=} \forall X \subseteq R. \forall A \in R.\ (A \text{ NSA} \wedge X \rightarrow A \wedge A \notin X) \Rightarrow X \rightarrow R$
  - Keine funktionalen Abhängigkeiten zwischen Nichtschlüsselattributen
  - 3NF impliziert 2NF (Teilschlüssel hängen vom Schlüssel ab $\mapsto$ transitiver Weg)
- **Transformation in 3NF**
  - Bilde Relation aus Nichtschlüsselattributen und davon abhängigen Attributen
  - Entferne abhängige Attribute aus ursprünglicher Relation

  z.B. `L_City(status,city)` +`Lieferant(s#,sname,city)`

Boyce-Codd Normalform (BCNF)

- **Problem: Transitive Abhängigkeit von Schlüsseln**

  z.B. `PostAdr(PLZ,Stadt,Adresse)` mit Schlüssel {`(Stadt,Adresse)`}

  – 3NF, aber Schlüsselattribut `Stadt` funktional abhängig von `PLZ`

  – Beziehung `PLZ`→`Stadt` nicht unabhängig von `Adresse` zu speichern

- **Relationenschema $\mathcal{R}$ ist in Boyce-Codd Normalform**

  = Kein Attribut hängt transitiv von einem Schlüssel ab

  $\hat{=} \forall X \subseteq R. \forall A \in R. (X \rightarrow A \wedge A \notin X) \Rightarrow X \rightarrow R$

  $\Rightarrow$ BCNF ist Verschärfung von 3NF: alle Schlüssel ohne Redundanzen

  – BCNF $\hat{=}$ jeder Determinant ist Schlusselkandidat

    · $X \subseteq R$ Determinant, falls ein $A \in R\text{-}X$ voll von X abhängt

- **Transformation in BCNF**

  – wie Transformation in 3NF (zusätzliche Betrachtung von Schlüsselattributen)

  z.B. `P_City(PLZ,Stadt)` + `P_Adr(PLZ,Adresse)`

  – Bei mehreren Alternativen unabhängige Projektion bevorzugen

    · FD's nicht als globale Bedingung über mehrere Relationen verstreuen

BCNF vs. 3NF

- Änderung im Leitbeispiel
  - `sname` eindeutige Bezeichnung für Lieferanten
  - `status` interpretiert als Zuverlässigkeit (unabhängig von `city`)

  $\Rightarrow$ neu: {`sname`→`s#`, `sname`→`status`, `sname`→`city`}; `city`→`status` entfällt
- `Lieferanten(s#,sname,status,city)`
  - Einzige Determinanten `s#` und `sname` sind Schlüssel $\Rightarrow$ BCNF
- Neue Relation `Lieferungen(s#,sname,p#,qty)`
  - Determinanten sind `(s#,p#)`, `(sname,p#)`, `s#` und `sname`
  - Schlüssel sind nur `(s#,p#)` und `(sname,p#)` $\Rightarrow$ 3NF aber keine BCNF
  - Transitive FD's: `(s#,p#)`→`sname`→`s#` bzw. `(sname,p#)`→`s#`→`sname`
- Anforderungen der BCNF zu stark?
  - Unterschiede zur 3NF kaum sichtbar, nicht immer einsichtig
  - Überprüfung aufwendig: Test auf BCNF ist NP-vollständig
  - Konflikt mit Abhängigkeitstreue
    - bei Zerlegung von `PostAdr` geht FD `Stadt,Adresse`→`PLZ` verloren
    - Schlüssel zerbrochen! $\Rightarrow$ BCNF nicht immer sinnvoll

3NF ist die praktisch wichtigste Normalform

Dekompositionsanforderung: Minimalität

- **Problem: Globale Redundanz**

  z.B. $\mathcal{S} = \{\mathcal{R}_1(\underline{\text{A}},\text{B}), \mathcal{R}_2(\underline{\text{B}},\text{C}), \mathcal{R}_3(\underline{\text{A}},\text{C})\}$ ist in BCNF

  – Beziehung zwischen A und C auf zwei Arten gespeichert

  – $\mathcal{R}_3$ abhängig von $\mathcal{R}_1$ und $\mathcal{R}_2$

- **Datenbankschema $\mathcal{S}$ ist minimal bezüglich Γ**

  – $\mathcal{S}$ erfüllt alle Bedingungen aus Γ

    · Γ Menge von Forderungen wie 3NF, Verbundtreue,...

  – Keine kleinere Anzahl von von Relationenschemata erfüllt Γ

    · $\forall \mathcal{S}'.\ |\mathcal{S}'| < |\mathcal{S}| \Rightarrow \mathcal{S}'$ erfüllt Γ nicht

Alle Abhängigkeiten durch Schlüssel repräsentiert

- **Problem: Funktionale Abhängigkeiten unsichtbar**
  - $\mathcal{S} = \{$`P_City(PLZ,Stadt), P_Adr(PLZ,Adresse)`$\}$
    ist in BCNF und minimal
  - Abhängigkeit `Stadt,Adresse` $\rightarrow$ `PLZ` nicht durch Schlüssel repräsentierbar
- **Datenbankschema $\mathcal{S}$ abhängigkeitstreu bezüglich $\mathcal{F}$**

  $\hat{=}\ \mathcal{F} \equiv \{\mathrm{K} \rightarrow \mathrm{R} \mid (\mathrm{R}, \mathcal{K}) \in \mathcal{S} \wedge \mathrm{K} \in \mathcal{K}\}$
  - $\mathcal{F}$ ist äquivalent durch die Schlüssel aus $\mathcal{S}$ darstellbar
  - '$\mathcal{S}$ charakterisiert $\mathcal{F}$ vollständig'

  $\Rightarrow$ Nur semantisch sinnvolle und konsistente Abhängigkeiten repräsentiert
  $\hat{=}$ Korrektheit von $\mathcal{S}$
  - oft im Konflikt mit BCNF

Dekompositionsanforderung: Verbundtreue

Originalrelationen durch Verbund wiederherstellbar

- **Problem: Zerlegung erzeugt Fremdtupel**

  z.B. $\mathcal{R}$(A,B,C) zerlegbar in $\{\mathcal{R}_1$(A,B),$\mathcal{R}_2$(B,C)$\}$ wobei $\mathcal{F}$=$\{$A$\rightarrow$B,C$\rightarrow$B$\}$

  – Verbund $\mathcal{R}_1 \bowtie \mathcal{R}_2$ erzeugt Tupel, die nicht in $\mathcal{R}$ waren

  – Dekomposition muß Struktur der FD's beachten

- **Dekomposition $\underline{X \mapsto X_1, \ldots X_n}$ verbundtreu bezüglich $\mathcal{F}$**

  – Alle Anwendungsdaten aus Basisrelationen herleitbar

  $\hat{=}$ Vollständigkeit von $\mathcal{S}$

  – $\forall r \in SAT_X(\mathcal{F}).\ r = \pi_{X_1}(r) \bowtie \ldots \bowtie \pi_{X_n}(r)$

  $\Rightarrow$ Beschränkung auf 'sinnvolle' Zerlegungen

- **Leicht überprüfbare Kriterien**

  – Ein Schema enthält Schlüssel für die Gesamtrelation

  – $\exists i.\ X_i \rightarrow X \in \mathcal{F}^+$ $\underline{\text{Universalschlüssel}}$ X_i

Zerlege initiale Universalrelation

- **Ausgangspunkt: Attributmenge $\mathcal{U}$, FD-Menge $\mathcal{F}$**
- **Verfahren**
  1. Bestimme Schlüsselkandidaten $\mathcal{K}_\mathcal{F} = \{K \subseteq \mathcal{U} \mid K_F{}^+ = \mathcal{U}$ und K minimal$\}$
  2. Fixiere Initialrelation $\mathcal{R} = (\mathcal{U}, \mathcal{K}_\mathcal{F})$
  3. Suche in $\mathcal{R}$ transitive FD's $K \rightarrow Y \rightarrow A$ mit $K \in \mathcal{K}_\mathcal{F}$, $A \notin KY$, $\neg(Y \rightarrow K)$
     zerlege $\mathcal{R}$ in $\{\mathcal{R}_1(R \setminus A, \mathcal{K}_\mathcal{F}), \mathcal{R}_2(Y \cup \{A\}, \{Y\})\}$
  4. Wiederhole Schritt 3 mit enstehenden Relationen bis 3NF erreicht ist
- **Resultat**
  - Verbundtreues Datenbankschema in 3NF
  - Minimalität und Abhängigkeitstreue i.A. nicht gewährleistet
  - Verfahren ist NP-vollständig
- Dekomposition von `Liefert(s#,sname,status,city,p#,qty)`
  $\mathcal{F} = \{$`s#`$\rightarrow$`sname`, `s#`$\rightarrow$`status`, `s#`$\rightarrow$`city`, `city`$\rightarrow$`status`, `s#,p#`$\rightarrow$`qty`$\}$ – $\mathcal{K} = \{\{$`s#,p#`$\}\}$
  $\mapsto \mathcal{S} = \{$`LName(s#,sname)`, `LStat(s#,status)`, `LCity(s#,city)`, `Lieferung(s#,p#,qty)`$\}$

Manipuliere funktionale Abhängigkeiten

- **Ausgangspunkt: Attributmenge $\mathcal{U}$, FD-Menge $\mathcal{F}$**
- **Verfahren**
  1. Ergänze Dummy-FD $\mathcal{U} \rightarrow \delta$
  2. Eliminiere redundante FD's aus $\mathcal{F}$ (f mit $\mathcal{F} \backslash f \vdash f$ – Membership-Test!)
  3. Eliminiere unwesentliche Attribute aus FD's in $\mathcal{F}$
     · A unwesentlich in X→Y, wenn $\mathcal{F} \backslash (X \rightarrow Y) \cup \{X' \rightarrow Y, X \rightarrow Y'\} \vdash X \rightarrow Y$ (X=X'A, Y=Y'A)
  4. Bilde Äquivalenzklassen $\mathcal{F}_X = \{Z \rightarrow Y \mid X \rightarrow Z \in \mathcal{F}^+ \wedge Z \rightarrow X \in \mathcal{F}^+\}$
  5. Bilde Schemata $\mathcal{R}_X = (\mathrm{ATTR}(\mathcal{F}_X), X)$ (d.h. Primärschlüsselmenge X)
  6. Entferne Dummy-Attribut aus der entsprechenden Teilrelation
- **Resultat**
  – Minimales, abhängigkeits- und verbundtreues Datenbankschema in 3NF
  – Verfahren ist polynomial
  – Dummy-Attribut sichert Existenz eines Universalschlüssels ($\mapsto$ Verbundtreue)
- **Synthese im Leitbeispiel**
  – $\mathcal{F} = \{$`s#`→`sname`, `s#`→`status`, `s#`→`city`, `city`→`status`, `s#,p#`→`qty`$\}$
  – `s#`→`status` ist redundant, `sname,status,city,qty` unwesentlich in Dummy-FD
  $\mapsto \mathcal{R}_1(\{$`s#,sname,city`$\},\{$`s#`$\})$, $\mathcal{R}_2(\{$`city,status`$\},\{$`city`$\})$, $\mathcal{R}_3(\{$`s#,p#,qty`$\},\{$`s#,p#`$\})$

Vierte Normalform (4NF)

Komplexe Zusammenhänge durch Schlüssel darstellbar

- **Problem: mehrwertige Abhängigkeiten**
  - `Bücher(ISBN,Autor,Stichwort,...)`: Autorenmenge hängt nur von `ISBN` ab
  - Zusammenhang nicht unabhängig von Stichwörtern speicherbar $\Rightarrow$ Redundanz
- **$X \twoheadrightarrow Y$ 'Y mehrwertig abhängig von X':**
  - Für jeden Wert der Attribute X existiert eine feste Menge von Y-Werten
  - Y-Werte unabhängig von Werten aus $R-(X \cup Y)$
  - 'Relation r genügt der MVD $X \twoheadrightarrow Y$' (MVD $\hat{=}$ Mehrwertige Abhängigkeit)
  - Es gilt $X \twoheadrightarrow Y$, falls $Y \subseteq X$ oder $X \cup Y = R$ (triviale MVD)
  - Nichttriviale MVD's sind beim Entwurf explizit zu deklarieren
- **Relationenschema $\mathcal{R}$ ist in vierter Normalform**
  - $\mathcal{R}$ in BCNF und in jeder nichttrivialen MVD $X \twoheadrightarrow Y$ enthält X einen Schlüssel

  $\Rightarrow$ $X \twoheadrightarrow Y$ ist funktionale Abhängigkeit ($\mapsto$ keine zwei echten MVD's in $\mathcal{R}$)
- **Transformation in 4NF**
  - Analog 3NF: Suche in $\mathcal{R}=(R,\mathcal{K})$ nichttriviale MVD $X \twoheadrightarrow Y$
    - Bilde neue Relationen $\mathcal{R}_1=(R-Y,\mathcal{K})$ und $\mathcal{R}_2=(X \cup Y,X)$

  z.B. `Bücher(ISBN,Stichwort,...)` + `B_Autor(ISBN,Autor)`

Fünfte Normalform (5NF/PJNF)

- **Problem: komplexe ternäre m:n Beziehung**
  z.B. `Mitarbeit(Person,Projekt,Sprache)` mit komplexer Semantik
  - · Person arbeitet in Projekt, Projekt verlangt Sprache, Mitarbeiter muß Sprache können,...
  - – Einfügen/Löschen verlangt komplizierte Aktualisierung der restlichen Relation
    - · Aktualisierung alleine über Schlüssel nicht möglich
- **Verbundabhängigkeit $\bowtie[X_1,\ldots,X_n]$:**
  - – Schema R ist ohne Verluste in Schemata $X_1,\ldots,X_n$ zerlegbar
  - – Für jede Relation r gilt $r = \pi_{X_1}(r) \bowtie \ldots \bowtie \pi_{X_n}(r)$
    - · 'Relation r genügt der JD $\bowtie[X_1,\ldots,X_n]$' (JD $\hat{=}$ Verbundabhängigkeit)
  - – Es gilt $X \twoheadrightarrow Y$, falls $\bowtie[X \cup Y, R\text{-}X]$
  - – Verbundabhängigkeiten sind beim Entwurf explizit zu deklarieren
    - · z.B. $\bowtie$`[{Person,Projekt}, {Person,Sprache}, {Projekt,Sprache}]`
- **Relationenschema $\mathcal{R}$ ist in fünfter Normalform**
  - – $\mathcal{R}$ ist in 4NF und in jede JD $\bowtie[X_1,\ldots,X_n]$ enthält jedes X_i einen Schlüssel

  $\Rightarrow$ Update über Schlüssel eindeutig handhabbar

  $\Rightarrow$ Keine komplex wechselseitig abhängigen Schlüsselattribute

  z.B. `Mitarbeit` hat einzigen Schlüssel {`Person,Projekt,Sprache`} $\Rightarrow$ keine 5NF
- **Transformation in 5NF**
  - – Eliminiere Verbundabhängigkeiten durch Zerlegung in mehrwegige Verbunde

Entwurfstheorie – Zusammenfassung

- **Festlegung funktionaler Abhängigkeiten**
  - Erhältlich aus Analyse der Miniwelt mit ERM
  - Unterstützt Methodik für guten Entwurf,
  - erlaubt semantische Integritätskontrollen im DBS
- **Modellierung als relationales Schema**
  - Ziel: klare, natürliche Zuordnung von Objekt und Datenstruktur
  - Normalisierung existierender Relationen (lokales Verfahren)
    - Schrittweise Elimination von Anomalien,
  - Synthese von 3NF-Relationen (globales Verfahren)
    - ggf. Überprüfung von Schlüsseln (BCNF), MVD's (4NF) und JD's (5NF)
  - Ergänze globale Integritätsbedingungen
- **Probleme**
  - Definition relevanter FD's bei vielen Attributen?
  - Syntheseverfahren liefern i.a. mehrere Alternativen – wie auswählen?
  - Konflikt zwischen BCNF und Abhängigkeitstreue
  - Modellierung von Abstraktionskonzepten?
- **Zusätzliche Aspekte**
  - Effizienz-/Stabilitätsanforderungen können schwächere Normalform erzwingen

  $\Rightarrow$ Der Entwerfer – nicht das Verfahren – bestimmt den endgültigen Entwurf

Grundlagen der Datenbanken

Lektion 10

Das Netzwerkmodell

1. Konzepte
2. Abbildung von ER-Modellen
3. Datenbeschreibung in CODASYL-Netzwerken
4. Datenmanipulation in CODASYL-Netzwerken

ER-Modell mit Einschränkung auf Pointerstrukturen

- **Organisation der Daten in gerichteten Graphen**
  - Schemaebene (Typen): Record Types als Knoten, Set Types als Kanten
  - Instanzenebene: Graph bestehend aus Records und Sets (Links)
- **Navigierender Zugriff**
  - Ein Record als Ausgangsposition
  - Weitere Records durch Verfolgen einer Zeigerkette erreichbar
  - Benutzer muß aktuelle Position im Graphen kennen
- **Implementierungsnahe Betrachtungsweise**
  - Einfach strukturiertes Modell, aber undurchsichtige Semantik
  - Wenig Schutzmechanismen – alle Verantwortung beim Benutzer
  - Theoretisch wenig interessant, da Eigenschaften schlecht nachweisbar
  - Immer noch erfolgreich im Einsatz
- **Zwei wichtige Spezialfälle**
  - Hierarchische Systeme
  - CODASYL/DBTG

Grundkonzepte netzwerkbasierter Datenbanken

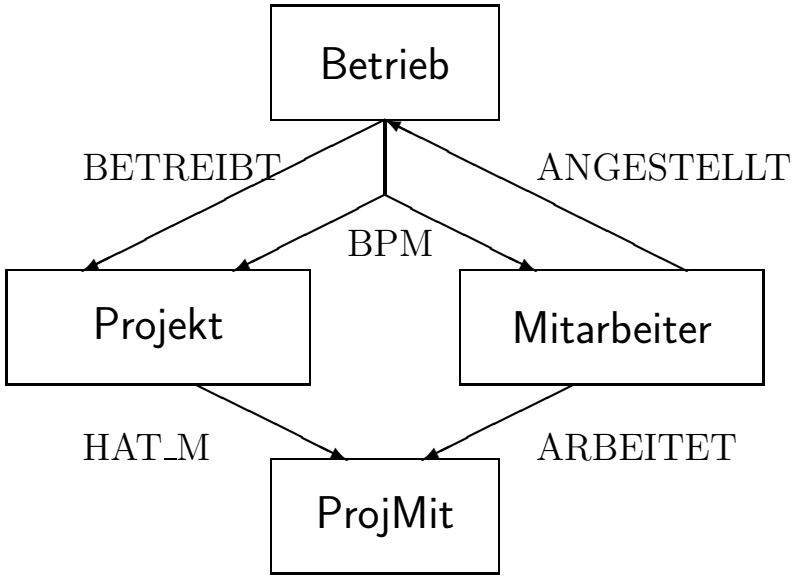
- **Record Type**
  - Besteht aus Feldern (Items) – Struktur entspricht COBOL-Records
  - Wiederholungen und leere Records erlaubt
  - Records benutzen Data Base Key als permanenten internen Identifikator
  - Kein Primärschlüsselkonzept
  - **Record**: Ausprägung eines Record Types $\hat{=}$ Entity
- **Set Type (Link Type)**
  - Struktur zwischen Record Types bestehend aus Owner und Member
  - Owner ist Record Type oder System (für Navigation)
  - Owner kann mehrere Member-Types, Member kann mehrere Owner haben
  - Beliebig viele Set Types zwischen Record Types (auch Zyklen!) erlaubt
  - Nur 1:n und 1:1 Relationships möglich, keine rekursiven Set-Typen
  - **Set**: Ausprägung eines Set Types
    - Instanz des Owner Typs mit allen verketteten Members
- **Area (Realm)**
  - Benannte Speichereinheit zur physischen Unterteilung der Datenbank
- **Schema**
  - Gesamtes Datenbankschema

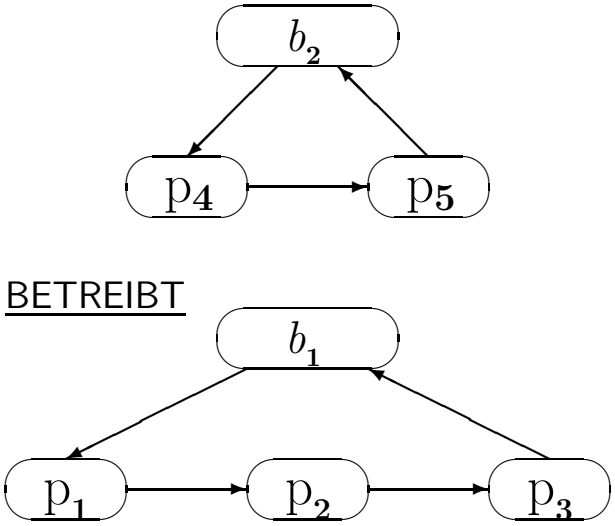
Graphische Beschreibung von Netzwerkmodellen

Schema-Diagramm und Instanzendiagramm

– Pfeile vom Owner zum Member

– *Achtung: Manche Bücher Pfeilrichtung umgekehrt (≙ Funktionalitätspfeil)*





– Simulation einer m:n Beziehung zwischen Projekt und Mitarbeiter
  · Ketten-Record (Typ) ProjMit + Sets HAT_M, ARBEITET

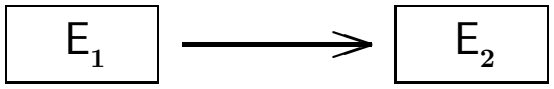
– Multimember Set-Type BPM (ein Owner Typ, zwei Member-Typen)

– Zwei Set-Types BPM und BETREIBT mit Owner Betrieb und Member Projekt

– Zyklus BPM–ANGESTELLT zwischen Betrieb und Mitarbeiter

Abbildung von ER-Modellen

- **Entity-Typ $E(A_1{:}D_1,\ldots,A_m{:}D_m)$**
  - Record-Typ: Felder entsprechen Attributen A_i
- **Binärer funktionaler (1:n) Beziehungstyp $R(E_1,E_2)$**
  - Standard Set-Typ mit Owner E_2 (!)
  - ggf. Attribute von R in den Member-Typ E_1 integrieren
  - 1:1 Beziehungen müssen separat vom Programm überwacht werden
- **m:n Beziehungstyp $R(E_1,..,E_n;\ A_1{:}D_1,\ldots,A_m{:}D_m)$**
  - Ketten-Record Typ R mit Dummy-Attribut A (und ggf. $A_1,\ldots,A_m$)

  \+ Set-Typen $S_1,..,S_n$ mit Ownertyp E_i und Membertyp R
- **Rekursive Beziehungstyp R(E,E)**
  - Ketten-Record Typ R und zwei Set-Typen mit Ownertyp E und Membertyp R
- **IS_A-Beziehung** 

  - Standard Set-Typ mit Owner E_2 und Member E_1
  - Zusatzbedingung muß separat überwacht werden.

Hierarchisches Modell (IBM, 1969)

Implementierungsnahe Ausprägung des NWM

- **Kommerziell erfolgreichstes Datenbankmodell**
  - IMS: Datenbanksystem der ersten Generation
    - sehr schnell, ausgefeilt, Anwendungen schwer zu entwickeln
  - Große Datenbestände in 70er Jahren hierarchisch erstellt
- **Netzwerkschema als 'Wald' (Menge von Bäumen)**
  - Keine Zyklen
  - Record-Typ kann nur einen Owner haben
    - m:n Beziehungen durch zwei getrennte Hierarchien zu beschreiben
    - Trick: Zeiger (virtual records) als fiktive Kopie eines Records
- **Leicht zu implementieren**
  - Hierarchische Dateien oder Baumstrukturen mit sequentieller Verzweigung
  - 4 Speicherstrukturen
    - HSAM: Sequentieller Zugriff auf Wurzel und Nachfolger
    - HISAM: Indexierter Zugriff auf Wurzel, sequentiell auf Nachfolger
    - HDAM: Hashzugriff auf Wurzel, Pointer auf Nachfolger
    - HIDAM: Indexierter Zugriff auf Wurzel, Pointer auf Nachfolger
  - Nur sehr einfache Navigationsoperationen

CODASYL / DBTG
(**CO**nference on **DA**ta **SY**stem **L**anguages / **D**ata **B**ase **T**ask **G**roup)

- **Prägend für Datenbank-Entwicklung der 70er**
  - DBTG Report 1971: Grundlage für viele Implementierungen
  - 1975–1978: Ergänzungen & Änderungen
  - ANSI/ISO Standard 1981 — bis heute nicht offiziell akzeptiert
- **Sprachen**
  - Schema DDL für konzeptionelle Ebene: COBOL-ähnlich
    - enthält auch viel speicherspezifische Information
  - Subschema DDL für externe Ebene
  - DSDL für interne Ebene: Speicherstrukturen, Record Packing, Recovery
  - Eingebettete DML (COBOL, PL/I, FORTRAN): navigierend, satzorientiert
  - Programmierschnittstelle: Datenaustausch über User Working Area (UWA)

Schema DDL: Beschreibungselemente

- **Schema Entry: Deklaration der Datenbank**
  - `SCHEMA NAME is` *dbname* [*Privacy-Klauseln*].
- **Area Entry: Einheiten der Speicherzuordnung**
  - `AREA NAME is` *aname* [*temporär/permanent-Klausel*].
- **Record Entry: Deklaration von Record-Typen**
  - `RECORD NAME is` *rname* *Record-Klauseln*.

  \+ Angaben über Adressierungsart, Area-Zugehörigkeit
  Feldbeschreibung, Sekundärschlüssel
- **Set Entry: Deklaration von Set-Typen**
  - `SET NAME is` $sname_1$ `OWNER is` $rname_o$ *Set-Klauseln*.
    `MEMBER is` $rname_m$ *Member-Klauseln*.

  \+ Angaben zur Realisierung, Ordnung der Members,
  Details für Zugriffe, Anordnung und Änderungen der Mitgliedschaft

Record Types: Angaben zur Adressierungsart

- **Speicherstrukturen werden pro Record-Typ festgelegt**
  - `LOCATION MODE`-Klausel: wie wird ein Satz gespeichert?
    - Information über Speicherzuweisung nützlich beim Aufsuchen (↦`FIND`)
  - Eigentlich Bestandteil des internen Schemas (DSDL)
- `LOCATION MODE is [` *hash-function* `] CALC USING` *id*
  - Adresse wird durch Hash-Funktion über Seitenidentifikatoren berechnet
    - Record wird auf berechneter Seite abgelegt, falls Platz ist
    - Ansonsten Verkettung und Speicherung auf nächster freier Seite
  - Nutzer kann eigene Hash-Funktion angeben
  - Optionale Duplikateneliminierung mit `DUPLICATES ARE NOT ALLOWED`
- `LOCATION MODE is DIRECT` *Database-Key*
  - Benutzer legt Plazierung über physischen Schlüssel *Data-base-key* fest
  - Direkte Speicherung und Zugriff über abgelegte Adresse im *Data-base-key*
- `LOCATION MODE is VIA SET` *Set-Name*
  - Speicherung möglichst nahe bei anderen Members des gleichen Sets
  - Zugriff navigierend über Elemente des Set-Typs *Set-Name*
- `LOCATION MODE is SYSTEM`
  - System bestimmt Speicherplatz – Zugriff durch allgemeinen `FIND`-Befehl.

Record Types: sonstige Angaben

- **Beschreibung einzelner Felder eines Record-Typs**
  - Angabe von Level, Feldname und Typ
    - z.B. `02 Autor PICTURE is` *COBOL-Pattern*
    - z.B. `02 Preis TYPE is` *COBOL-Typ*
  - Anzahl Vorkommnisse bei Arrays: `OCCURS` n `TIMES`
  - ggf. Herkunft der Feldwerte (physisch oder virtuell)
    - z.B. `ISBN is ACTUAL AND SOURCE is ISBN of OWNER of EX_VON`
  - Validierungsangaben mit Wertebereichen: `CHECK is ...`
- **Zuordnung der Record-Typen zu Areas (optional):**
  - Klausel `WITHIN` $area_1$ [$area_2, \ldots$]
  - Bei mehrere Alternativen muß vor Speicherung eine Area spezifiziert werden
- **Sekundärschlüssel (Erweiterung von 1978)**
  - `SEARCH-KEY is` *Data-base-key* `USING {INDEX|CALC}`
  - Spezifikation zusätzlicher Zugriffspfade
  - Implementiert durch Hash-Struktur oder B*-Bäume
  - Beschleunigt Zugriff über Feldwerte (Angabe von Inhalt + Search Key)

Definition von Set-Types

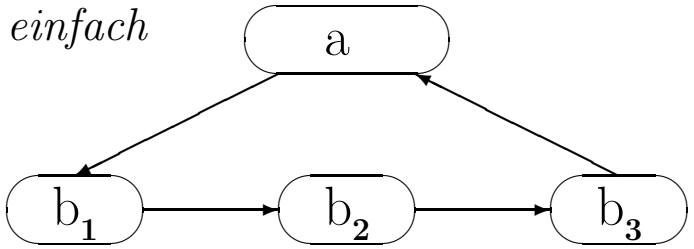
```
SET NAME is sname_1 OWNER is rname_o   Set-Klauseln.
                MEMBER is rname_m   Member-Klauseln.
```

- **Angabe von Set-Name, Owner- und Member-Typen**
- **Wichtigste Angaben**
  - Welche Speicherungsstruktur wird verwendet? $\mapsto$ `SET MODE`-Klausel
  - An welcher Stelle werden Members eingefügt? $\mapsto$ `ORDER`-Klausel
  - Wie wird ein Record Member eines Sets? $\mapsto$ `MEMBER`-Klausel
  - Wie soll die Suche nach Members unterstützt werden? $\mapsto$ `SEARCH`-Klausel
  - Welche Set-Ausprägung soll bei Einfügen/Suche benutzt werden? $\mapsto$ `SET-SELECTION`-Klausel
- **Weitere Optionen**
  - Dynamischer Set-Typ: Record-Typen zur Laufzeit veränderbar
  - Singulärer Set-Typ: nur eine Set-Ausprägung existiert
    - erlaubt sequentiellen Scan über alle Elemente eines Record-Typs

Set-Types: Angaben zur Speicherungsstruktur

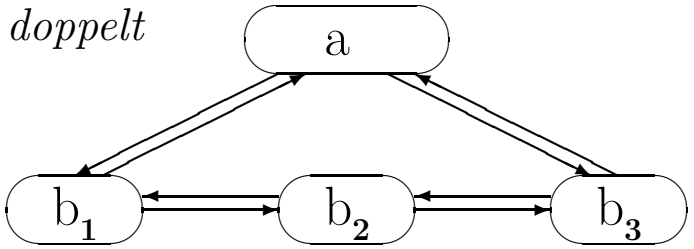
- Einfache und doppelte Verkettung

einfach




```
SET NAME is AB;
MODE is CHAIN;
OWNER is A.
MEMBER is B.
```

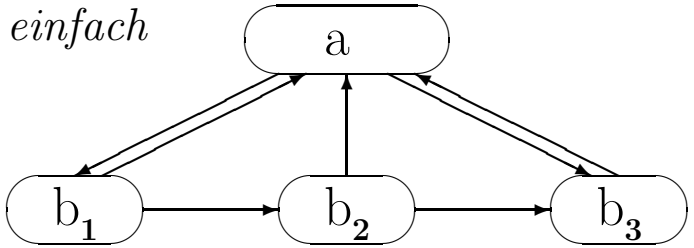
doppelt




```
SET NAME is AB;
MODE is CHAIN LINKED TO PRIOR;
OWNER is A.
MEMBER is B.
```

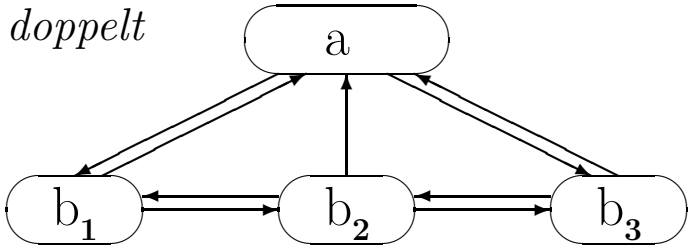
- Kette mit Verbindung zum Owner ($\mapsto$ `MEMBER`-Klausel)

einfach




```
SET NAME is AB;
MODE is CHAIN;
OWNER is A.
MEMBER is B LINKED TO OWNER.
```

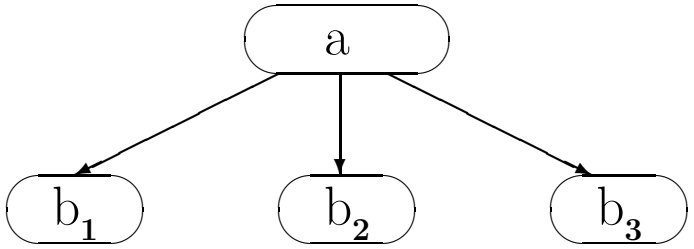
doppelt




```
SET NAME is AB;
MODE is CHAIN LINKED TO PRIOR;
OWNER is A.
MEMBER is B LINKED TO OWNER.
```

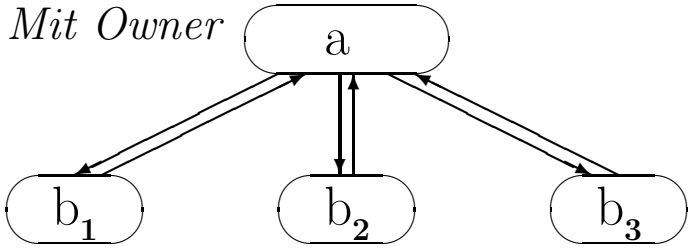
Set-Types: Angaben zur Speicherungsstruktur II

- Pointer Array




```
SET NAME is AB;
MODE is POINTER ARRAY;
OWNER is A.
Member is B.
```

Mit Owner




```
SET NAME is AB;
MODE is POINTER ARRAY;
OWNER is A.
Member is B LINKED TO OWNER.
```

- `INDEXED`: Für jede Set-Instanz wird ein eigener kleiner Index gehalten
- Eigentlich Bestandteil des internen Schemas ($\mapsto$ DSDL)

Set-Types: Angaben zur Mitgliedschaft

- Storage Class: `MEMBER is` $rname_m$ { `AUTOMATIC` | `MANUAL` }
  - Wie wird ein Record Member eines Sets?
  - `AUTOMATIC`: beim Speichern eines neuen Records mit `STORE`
    - Bei nichtsingulären Sets ist `SET SELECTION`-Klausel erforderlich
  - `MANUAL`: explizit durch das Anwenderprogramm mit `CONNECT`
- Removal Class: `MEMBER is` $rname_m$ { `MANDATORY` | `OPTIONAL` | `FIXED` }
  - Wann wird ein Record aus einem Set entfernt?
  - `FIXED`: nur beim Löschen des Records mit `ERASE`
  - `MANDATORY`: Record muß in irgendeinem Set Member sein
    - Entfernung nur beim Löschen des Records
    - oder Wechsel in anderen Set mit `MODIFY MEMBERSHIP`
  - `OPTIONAL`: Mitgliedschaft jederzeit durch `DISCONNECT` widerrufbar
- Beispiel: Beziehungen zwischen Fachbereich und Student
  - Eingeschrieben-in: `AUTOMATIC MANDATORY`, da FB-Wechsel möglich
  - Hiwi-bei: `MANUAL OPTIONAL`, nur wenige und temporäre Mitglieder
  - Diplom-bestanden-bei: `FIXED OPTIONAL`, da unwiderruflich
- Record-Typ-übergreifende Integritätsbedingung
  - Detaillierter und komplexer als im Relationemodell

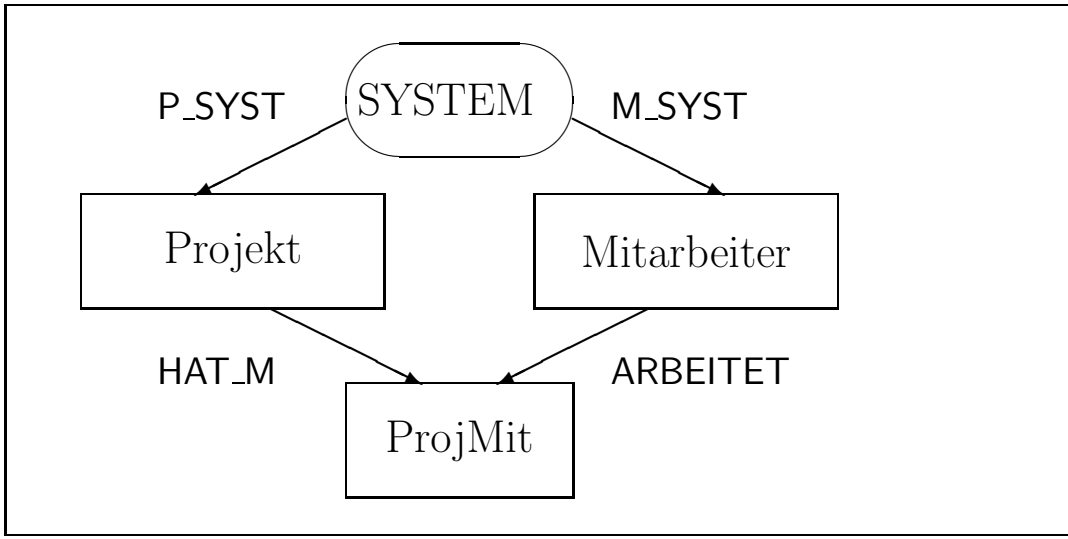
Set-Types: Auswahl einer Set-Ausprägung

- Auswahl konkreter Sets beim Aufsuche/Einfügen von Records
  - `SET SELECTION is THRU` *set-type* `OWNER IDENTIFIED by ...`
  - Wählt Set vom Typ *set-type*, dessen Owner wie folgt bestimmt ist...
  - Kurz: `SET SELECTION is THRU ...`, falls *set-type* definierender Set-Typ
  - Navigation durch Auswahlkaskade mit `THEN THRU ...`
- `CURRENT OF SET`
  - Explizite Anwahl durch Anwenderprogramm (CRS wird gesetzt)
- `DATA-BASE-KEY [EQUAL to` *db-id* `]`
  `CALC-KEY [EQUAL to` *db-ids* `]`
  - Owner entsprechend `LOCATION-MODE`-Klausel des Owner-Typs festgelegt.
    - Schlüssel werden entsprechend dem `DIRECT` bzw. `CALC`-Modus bestimmt
  - Durch `EQUAL to` werden weitere Alternativschlüssel zum Aufsuchen angegeben
- `MEMBER` *rname* `SELECTION`
  - Übernehme Selektionsmechanismus, der für Member *rname* angegeben ist
    - Weniger Schreibarbeit, erkennbarere Koppelung
- `SYSTEM`
  - Nur eine (singuläre) Set-Ausprägung existiert
- Achtung: wenig standardisiert!!
  - Alternative Bezeichnungen und Klassifikationen im Einsatz

Set-Types: sonstige Angaben

- **Speicherungsreihenfolge der Mitglieder**
  - `ORDER is { TEMPORARY | PERMANENT } INSERTION is ...`
  - Angabe, wo ein neues Mitglied in eine Kette eingefügt wird
    - `FIRST, LAST, NEXT, PRIOR`: Stack, Liste, relativ zum letzten Einfügen
    - `SORTED [ INDEXED ] by { `*Key*` | `*Record*` }`: Einfügen abhängig vom Wert
    - `IMMATERIAL`: Systembestimmt
  - Ordnung darf ggf. (nicht) verändert werden
- **Spezifikation zusätzlicher Zugriffspfade**
  - `SEARCH-KEY is `*Data-base-key*` USING { INDEX | CALC }`
  - Beschleunigt Zugriff auf Members eines konkreten Sets
  - Effizienter Zugriff unterstützt durch sortierte Speicherung

Beispiel eines Schemas in CODASYL




```
SCHEMA NAME is PROJEKT_MANAGEMENT

AREA NAME is PROJ_EMP

RECORD NAME is PROJEKT;
 LOCATION MODE is CALC USING P#
          DUPLICATES ARE NOT ALLOWED;
 WITHIN PROJ_EMP.
     02 P#         PICTURE is  999.
     02 PNAME      PICTURE is  X(30).
     02 STATUS     PICTURE is  X(2).
     02 BUDGET     PICTURE is  99999.

RECORD NAME is MITARBEITER;
 LOCATION MODE is CALC USING MNAME
          DUPLICATES ARE NOT ALLOWED;
 WITHIN PROJ_EMP.
     02 MNAME      PICTURE is  X(20).
     02 KAT        PICTURE is  99.
     02 DM_HR      PICTURE is  999.

RECORD NAME is PROJMIT;
 LOCATION MODE is VIA HAT_M SET
     02 WSTD       PICTURE is  99.
```

```
SET NAME IS P_SYST;
 OWNER is SYSTEM;
   MODE is POINTER ARRAY;
   ORDER is PERMANENT FIRST.
 MEMBER is PROJEKT AUTOMATIC MANDATORY;
   SET SELECTION is THRU SYSTEM.
    ⋮
    ⋮
SET NAME IS HAT_M;
 OWNER is PROJEKT;
   MODE is CHAIN LINKED TO PRIOR;
   ORDER is NEXT.
 MEMBER is PROJMIT MANUAL OPTIONAL
             LINKED TO OWNER;
   SET SELECTION is THRU CURRENT OF SET.
```

Datenmanipulation: Verarbeitungskonzept

- **DML-Operationen beziehen sich auf genau einen Satz**
  - Parameter aus UWA übernommen, Ergebnisse in UWA abgelegt
  - DBS merkt sich letzte Verarbeitungsposition durch Aktualisierungsindikatoren
  - Analog zum Cursor-Konzept des Relationemodells
- **Aktualisierungsindikatoren**
  - Ermöglichen Bestimmung von Set-Ausprägungen bei `STORE, CONNECT, ...`
  - Bei Ablauf implizit für alle Objekttypen eines Subschemas definiert
    - CRU (current-of-run-unit) — ein Pointer pro Anwendungsprogramm
    - CRR (current-of-record-name) — ein Pointer pro Record-Typ
    - CRS (current-of-set-name) — ein Pointer pro Set-Typ
    - CRA (current-of-area) — ein Pointer pro Area
  - Markieren jeweils letztbesuchte Positionen im Instanzennetzwerk
- **(Selektive) Unterdrückung der Aktualisierung möglich**
  - `RETAINING CURRENCY for {REALM|RECORD|SETS` *set-typen* `}`
  - Spezielle Klausel im Rahmen des DML-Kommandos `FIND`

CODASYL: Datenmanipulationsprache

- **Kommandos zum Aufsuchen**
  - `FIND` (Record), `GET` (Record/Feld), `FETCH` ($\hat{=}$ `FIND+GET`)
  - `ACCEPT`, `SET`
- **Änderungsoperationen**
  - `STORE` (Record), `ERASE` (Record), `MODIFY` (Record/Feld)
  - `CONNECT`, `DISCONNECT` (Member–Set), `ORDER` (Set)
- **Satzschutz**
  - `KEEP`, `FREE` (für Concurrency)
  - `OPEN`, `CLOSE` (Area)
- **Transaktionssteuerung**
  - `READY`, `FINISH`
  - `USE`
- **Records zuerst müssen gefunden werden**
  - `FIND`-Kommando muß oft vor anderen eingesetzt werden

Datenzugriff: FIND *format* [RETAINING CURRENCY for ...]

- **Lokalisieren eines Records**
  - Record wird CRU und kann dann durch GET in UWA gebracht werden

7 Formate zur Art der Identifikation

1. *rname* DATA-BASE-KEY is *Data-base-key*

- Direkter Zugriff wenn Schlüsselwert des Records bekannt ist

2. { ANY | DUPLICATE } *rname*

- Absoluter Zugriff mittels CALC-Schlüssel ($\mapsto$ LOCATION MODE is CALC)
- Mögliche Duplikate durch DUPLICATE-Option aufspürbar

3. DUPLICATE WITHIN { *rname* | *set-name* } [USING *ids*]

- Aufspüren von Duplikaten eines Suchbefehls
  - gleicher Suchschlüssel wie im vorhergehenden FIND
- Ermöglicht Durchsuchen aller Records eines Typs bzw. eines Sets

Weitere Suchformate

4. { `NEXT` | `PRIOR` | `FIRST` | `LAST` | *int* | *id* }
{ *rname* | `RECORD` } [`WITHIN` { *set-name* | *area* }]

- Navigation innerhalb der aktuellen Set-Ausprägung oder Area
- Zugriff relativ zum `CRS` / `CRA`
- Angabe von *rname*, falls mehrere Membertypen im angegebenen Set-Typ
- Verlangt Kenntnis der Speicherstruktur innerhalb des Set-Typs (bzw. der Area)

5. `CURRENT` *rname* [`WITHIN` { *rname* | *area* }]

- Aktualisierung der Indikatorposition (nach vorherigem `RETAINING ...`)

6. `OWNER WITHIN` *set-name*

- Findet Owner der aktuellen Set-Ausprägung

7. *rname* [`WITHIN` *set-name* [`CURRENT`]] `USING` { *ids* | *expr* } ...

- Suchen über Werte (`USING ...` $\hat{=}$ `where` in SQL)
- Steuerungskommandos: `RESULT in` $set\text{-}name_2$ `LIMITED by` $set\text{-}name_3$
`TALLYING` *id*

Änderungsoperationen

- `STORE` *rname*
  - Plazierung einer Satzausprägung der UWA entsprechend dem CRR
  - Verkettung mit `AUTOMATIC` Sets, bei denen *rname* Member ist
- `CONNECT` *rname* `TO` *set-name* [`,` $\mathit{set\text{-}name}_2$`, ...`]
  - Aufnahme eines Records (CRU) in Sets bei `SET MODE is MANUAL`
  - Vorheriges Setzen der entsprechenden CRS-Indikatoren nötig
- `DISCONNECT` *rname* `FROM` *set-name* [`,` $\mathit{set\text{-}name}_2$`, ...`]
  - Aufgabe der Mitgliedschaft eines Records in `OPTIONAL`-Sets
  - Neupositionierung der CRS-Indikatoren nötig
- `MODIFY` *rname* `ONLY` *set-name* `MEMBERSHIP`
  - Wechsel der Mitgliedschaft eines Records entsprechend der CRS-Indikatoren
- `ERASE` *rname* [`PERMANENT` | `SELECTIVE` | `ALL`]
  - Löscht Ausprägung von *rname*, sofern diese keine Members hat
  - `PERMANENT`: entfernt CRU, permanente Members, löst Verbindung zu anderen
  - `SELECTIVE`: entfernt auch optionale Members, die sonst isoliert sind
  - `ALL`: entfernt CRU und alle damit verbundenen Members

Transaktionen

- CODASYL überläßt Concurrency-Management dem Benutzer
  - Transaktionen beginnen mit `READY`, enden mit `FINISH`
  - `USAGE MODE`-Klausel bestimmt Synchronisationsmaßnahmen für alle Areas des Subschemas

- `USAGE MODE is [EXCLUSIVE|PROTECTED] {RETRIEVAL|UPDATE }`
  - `RETRIEVAL` beschränkt eigene Zugriffe auf Suchoperationen
  - `EXCLUSIVE` blockiert gesamte Area vor Fremdzugriff
  - `PROTECTED` erlaubt Fremdzugriff aber keine Fremdänderung
    - erzwingt Einbenutzerbetrieb auf Area
  - Ansonsten freier Fremdzugang (explizites `KEEP/ FREE` im Programm nötig)

- Lokale Synchronisation

  `KEEP` *rname* `{RETRIEVAL|UPDATE}`: Anforderung einer Sperre für Record *rname*

  `FREE` *rname* `{RETRIEVAL|UPDATE}`: Freigabe der Sperre für Record *rname*
  - CRU (Record unter aktuellem Zugriff) ist immer implizit gesperrt
  - geänderte Sätze müssen bis Ende der Transaktion gesperrt bleiben

  ↦ *Lektion über Transaktionsverwaltung*

Netzwerkmodell: Zusammenfassung

- **Informationsdarstellung: Record- und Set-Typen**
  - Beziehung $\hat{=}$ Verbindung zwischen Datensätzen, berücksichtigt Speicherstruktur
  - Semantikarmes Modell – keine Abstraktionskonzepte
- **Prozedurale Datenbanksprache**
  - Satzweiser Zugriff über vorhandene Zugriffspfade
  - Programmierer als Navigator verantwortlich für Leistungsaspekte
  - Komplexe DDL/DML
  - Keine typübergreifenden Operationen
- **Geringer Grad an Datenunabhängigkeit**
  - Abhängigkeit von (logischen) Zugriffspfaden
  - Hohes Leistungsvermögen durch spezialisierte Zugriffspfade möglich
  - Änderungen im konzeptionellen Schema beeinflussen Anwendungsprogramme
- **Evolutionspad zu objektorientierten Datenmodellen**
  - Ansatz von Objektidentität, gezieltere Navigation, Satzorientierung
  - Aspekte der Verarbeitung komplexer Objekte durch Pointerstrukturen

Grundlagen der Datenbanken

Lektion 11

Grundlagen des objektorientierten Datenmodells

1. Notwendigkeit für Objektorientierung
2. Grundkonzepte der Objektorientierung
   - Objekte und Identität, Klassen und Typen
   - Datenkapselung und abstrakte Datentypen
   - Komposition von Objekten
   - Vererbung, Overriding und dynamisches Binden
3. Entwurfskonzepte

Beschränkungen klassischer Datenmodelle

- **Einfach strukturierte Datenobjekte**
  - Datensatzorientiert: festes Format, nur einfache Datentypen
  - Darstellung von Objektstrukturen nur über Fremdschlüssel
- **Geringe semantische Ausdrucksfähigkeit**
  - Fehlende Abstraktionskonzepte (Aggregation, Generalisierung, ...)
  - Begrenzte Auswahlmächtigkeit der Anfragesprachen
  - Nur einfache Integritätsbedingungen
- **Uniforme Operationen für alle Datenstrukturen**
  - Nur Einfügen, Löschen, Ändern, Suchen von Werten
  - Keine Datenkapselung (Einschränkung erlaubter Operationen) möglich
  - Keine spezialisierten (benutzerdefinierten) Operationen
    - Keine Unterstützung von Zeit, räumlichen und logischen Beziehungen
- **Umständliche Einbettung in Programmiersprachen**
  - Impedance Mismatch: relationale DB-Sprache $\leftrightarrow$ Programmiersprache
  - Nicht mengenorientierte Operationen einzeln durch AP auszuführen
- **Zugeschnitten auf kurze Transformationen**
- **Ineffizient bei Nicht-Standard-Anwendungen**
  - CAD / CAM / CIM / Graphische Informationssysteme (GIS)
  - Multimediale Datenbanksysteme, Office-Automation, ...

Objektorientierte DBMS

Doppelrolle: Datenbanksystem + objektorientiertes System

- **Aspekte von Datenbanken**
  - Persistenz + Externspeicherverwaltung
  - Object Sharing
  - Synchronisation (Concurrency) + Recovery-Mechanismen
  - Ad-hoc Anfragesprachen
- **Aspekte objektorientierter Systeme**
  - Objektidentität + Typ/Klassenstruktur mit Datenkapselung
  - Komplexe Objekte
  - Typhierarchie, Vererbung, Überladung, dynamisches Binden
  - Operationale Vollständigkeit und Erweiterbarkeit
- **Möglichkeiten der Konzeption von OO-DBMS**
  - Erweiterung eines DBMS um OO-Konzepte ($\mapsto$ SQL-3, POSTGRES, DATALOG ...)
  - Erweiterung einer OOPL um DB-Konzepte ($\mapsto$ ObjectStore, ...)
  - Völlige Neuentwicklungen ($\mapsto$ O_2, ITASCA, ORION ...)
    - langfristig der sinnvollste Weg

Ebenen der Objektorientierung

- **Satzorientierung**
  - Einfache Objekte: atomare oder zusammengesetzte Elementtypen
  - Begrenzte Anzahl von Zusammensetzungsstufen
  - Fest vordefinierte Typkonstruktoren (set, bag, list, array, ...)
  - Fest vordefinierte Operationen: Werte suchen, einfügen, löschen, ändern
  - DB-Sprache ohne Unterstützung von Objektorientierung
- **Strukturelle Objektorientierung**
  - Komplexe Objekte mit unbegrenzte Anzahl von Zusammensetzungsstufen
  - Beschränkung von Typ-Überlappung, Rekursion, Beziehungen
  - Vordefinierte Operationen passend zu Typkonstruktoren
- **Verhaltensmäßige Objektorientierung**
  - Einfache (satzorientierte) Objektstrukturen
  - Benutzerdefinierbare Typen und Operatoren (auf Satzstrukturen)
- **Volle Objektorientierung**
  - Komplexe benutzerdefinierbare Objektstrukturen und Operatoren
  - Datenkapselung, Typhierarchie, Overriding, dynamisches Binden

  $\Rightarrow$ Effiziente Behandlung komplexer Strukturen und semantischer Bezüge

Grundkonzepte objektorientierter Systeme

1. **Objekte mit eigener Identität**
   - Mehr als eine Sammlung von Daten (Wertegleichheit $\neq$ Identität)
2. **Typ- und Klassenstruktur**
   - Gemeinsame Struktur und Charakteristika (Operationen) von Objekten
   - Typen als abstraktes, Klassen als Implementierungskonzept
3. **Datenkapselung**
   - Trennung von Schnittstelle und Implementierung ($\mapsto$ abstrakte Datentypen)
4. **Typ-Komposition**
   - Zusammensetzung von Objekten durch (selbstdefinierbare) Typkonstruktoren
5. **Klassen und Typ-Hierarchien**
   - Vererbung von Struktur, Methoden, Integritätsbedingungen und Defaultwerten
   - Subklassen/-typen können eigene Struktur, Methoden, etc. ergänzen
6. **Überladung, Überschreiben und dynamisches Binden**
   - Overriding: Methoden in Subklassen redefinierbar (Struktur + Axiome gelten)
   - Late binding: Bindung der Implementierung an das Objekt zur Laufzeit
7. **Operationale Vollständigkeit**
   - Turingmächtigkeit: größere Funktionalität als herkömmliche DB-Sprachen
8. **Erweiterbarkeit vordefinierter Klassen**
   - Kein Unterschied zwischen System- und benutzerdefinierten Typen

Objekte und Identität

- **Objekte sind mehr als beschreibende Werte**
  - Objekte haben einen eindeutigen Identifikator (Objekt-ID)
  - Objekte haben einen Zustand, der durch Attributwerte beschreibbar ist
    - Attributwerte können Referenz auf andere Objekte sein
  - Objekte haben Operationen, welche ihre Schnittstelle zur Außenwelt definieren
  - Objekte kommunizieren miteinander durch Nachrichten
- **Objekte existieren unabhängig von ihren Werten**
  - Objektidentifikation durch unveränderlichen Identifikator (Surrogat)
    - systemweit eindeutig, zustandsunabhängig
    - intern verwaltet, ohne eigene Semantik
  - Änderungen von Attributwerten ergeben dasselbe Objekt (anderer Zustand)

  $\Rightarrow$ Gleichheit von Attributwerten $\neq$ Identität von Objekten
  - Kontrast zum wertorientierten Relationenmodell

$\Rightarrow$ **Andere Verarbeitungsformen**
  - Suchen + Aktualisieren durch Aufruf von Operationen des Objekts
  - Objekt kontrolliert Zugriffe, Autorisierungen und Integrität
  - Redundanzfreie Realisierung semantischer Zusammenhänge

Objektidentität

- **Mehrere Arten von Gleichheit**
  - o_1`==`o_2: o_1 und o_2 bezeichnen *dasselbe Objekt* (Identität)
  - $o_1 =_s o_2$: Objekte o_1 und o_2 haben *denselben Zustand* (Flache Gleichheit)
  - $o_1 =_d o_2$: Alle *Werte* von o_1 und o_2 sind im gleichen Zustand (Tiefe Gleichheit)
    - (d.h. Komponenten*objekte* von o_1 und o_2 sind in der Tiefe gleich)
  - Es gilt $o_1 == o_2 \Rightarrow o_1 =_s o_2 \Rightarrow o_1 =_d o_2$
- **Mehrere Arten von Kopieroperationen**
  - o_1`:=`o_2 (Identitätserhaltende Zuweisung): o_1 zeigt auf *dasselbe Objekt* wie o_2
  - o_1`:=`o_2`.shallowCopy` : erzeugt einen Clon von o_2 (Flache Kopie)
    - d.h. ein neues Objekt mit gleichem Zustand wie o_2
  - o_1`:=`o_2`.deepCopy`: erzeugt neues Objekt mit gleichen Werten (Tiefe Kopie)

    und jeweils eine tiefe Kopie der Komponentenobjekte

Typen und Klassen

- **Typen definieren Struktur + Operationen**
  - Abstrakte Beschreibung gleichartiger Objekte im Typ-Interface (Schnittstelle)

  + Menge möglicher Implementierungen, welche der Schnittstelle genügen
- **Interface $\hat{=}$ Signatur + Axiome**
  - Signatur: Namen + Typstruktur von Attributen und Operationen
    - partielle Operationen durch Eingabebedingungen beschränkt
  - Axiome: Unveränderliche Eigenschaften von Attributen und Operationen
    - Ausgabebedingungen von Operationen, globale Invarianten
- **Implementierung $\hat{=}$ Objektrepräsentation + Methoden**
  - Repräsentation: konkrete Darstellung durch Menge von Datenstrukturen
  - Methode: Prozedurkörper für jede im Interface definierte Operation
  - auch interne Methoden und Datenstrukturen (ohne Interface nach außen)

  Achtung: Begriffe 'Methode' und 'Operation' im OOP oft umgekehrt
- **Klasse $\hat{=}$ Typ-Interface + (eine!) Implementierung**
  - Implementierungsnahes Konzept: konkrete Datenstrukturen + Operationen

  Achtung: Begriffe 'Klasse' und 'Typ' noch umstritten – oft umgekehrt
- **Extent $\hat{=}$ Menge aller Instanzen des Typs**
  - Wird durch die Typdefinition deklariert
  - Wird vom OODBMS bei Einfügen und Löschen von Objekten verwaltet

Datenkapselung und Abstrakte Datentypen

- **Interne Repräsentation von Objekten unsichtbar**
  - Zugriffe und Änderungen nur über Operationen der Schnittstelle
  - Attribute nicht direkt zugreifbar
  - Interne Attribute und Code der Operationen nach außen unsichtbar

  $\hat{=}$ mathematisches Konzept der abstrakten Datentypen
- **Objektspezifische Menge von Operationen**
  - Verhalten des Obektes aussschließlich durch Operationen bestimmt
  - Interne Struktur nach außen ohne Bedeutung

  $\Rightarrow$ Objekte sind gekapselt: erhöhte (semantische) Datenunabhängigkeit
- **Benutzerdefinierte Basistypen möglich**
  - Erzeugung problembezogener Klassen (z.B. Vector, Matrix, Dreieck, Kreis, ...)
  - Speziell zugeschnittene Operationen möglich

  $\Rightarrow$ geringerer Kommunikationsaufwand mit dem DBMS
- **Komplexere Anfragen einfacher zu realisieren**

  z.B. 'Finde alle Rechtecke , welche das Rechteck ((0,0),(1,1)) schneiden'
  - ADT `Box` mit Funktionen `intersect, contains, area, move, ...`
  - Typ `R-Eck(R-Nr:Int, Beschr: Box)`
  - Anfrage: `select R-Nr from R-Eck where intersect(Beschr,(0,0,1,1))`

Typen-Komposition

- **Erzeugung strukturierter Objekte und Datentypen**
  - Konstruktoren zur Komposition existierender Typen
  - Objektwertige Attribute
  - RM beschränkt sich auf Tupelbildung und Relationen
- **Wünschenswerte Konstruktoren**
  - Array (Vector) $\hat{=}$ Datenspeicher mit Direktzugriff
    - Zugriff auf Komponenten über Position, Speichergröße meist fest
  - List (Sequence) $\hat{=}$ unbegrenzter geordneter Datenspeicher
    - Einfügen und Lesen nur an aktuellem Element (oft als Stack)
  - Bag (Multimenge) $\hat{=}$ unbegrenzter ungeordneter Datenspeicher
    - Elemente erscheinen ggf. mehrfach, Reihenfolge ohne Bedeutung
  - Set $\hat{=}$ Menge (Assoziation)
    - Elemente erscheinen maximal einmal, Reihenfolge ohne Bedeutung
  - Record (Tupel) $\hat{=}$ Komposition (Aggregation) verschiedener Typen
    - PART-OF Semantik,
  - Beliebige (rekursive) Kombination existierender Konstruktoren
- **Generische Typen**
  - Benutzerdefinierte Typkonstruktoren $\hat{=}$ Typen mit Typ-Parameter

Typhierarchie und Vererbung

- **Kennzeichnung von Abhängigkeiten zwischen Typen**
  - Anordnung in Vererbungs-/Generalisierungs-/Spezialisierungshierarchie
  - IS-A Beziehung zwischen Subtyp (Spezialfall) und übergeordnetem Supertyp
  - Subtypen erben alle Attribute, Methoden, Integritätsbedingungen
  - Einfache oder multiple Vererbung – Disjunkte oder überlappende Spezialisierung
- **Erlaubte Modifikationen im Subtyp**
  - Hinzunahme neuer Attribute und Methoden
  - Verschärfung von Integritätsbedingungen
  - Umbenennung, Unterdrückung und Redefinition (Overriding) von Merkmalen
- **Vererbungsarten ($\hat{=}$ intendierte Semantik)**
  - Inklusionsvererbung: Vererbung basiert auf Strukturgleichheit
    - T Subtyp von T', wenn jedes Objekt von T auch eines von T' ist
  - Constraintvererbung: Inklusionsvererbung mit bennennbarer Einschränkung
    - Bestimmte T'-Merkmale sind in T durch eine Bedingung eingeschränkt
  - Spezifikationsvererbung: T-Objekte sind T'-Objekte mit zusätzlichen Attributen
  - Substitutionsvererbung: auf T-Objekte sind zusätzliche Methoden anwendbar
- **Vorteile**
  - Code-Wiederverwendung bei Repräsentation zusätzlicher Semantik
  - Modellierungsdisziplin (schrittweise Verfeinerung von Klassen)

Overriding und Dynamisches Binden

- **Methodenaufruf mit Pfadausdruck** `obj.methode(args)`
  - Sendet Nachricht an benanntes Objekt
  - Objekt führt seine Methode (mit Argumenten) gemäß seiner Typdefinition aus
- **Overloading (Überladen von Operatoren)**
  - Verschiedene Methoden in verschiedenen Klassen mit gleichem Namen
  - Implementierung durch Typ des benannten Objekts zur Compile-Zeit bekannt
- **Overriding (Überschreiben von Implementierungen)**
  - Neuimplementierung des Operationsrumpfes im Subtyp
  - Wahl der Implementierung zur Compile-Zeit nicht möglich
- **Spätes (dynamisches) Binden**
  - Bindung der Implementierung an ein Objekt zur Laufzeit
  - Tatsächlicher Typ des Laufzeitobjekts bestimmt gewählte Implementierung
  - Ermöglicht polymorphe Operationen (Deklaration in abstraktem Supertyp)

- **Problem: multiple Vererbung gleichnamiger Merkmale**

  $\Rightarrow$ Umbenennung in Erbenklasse oder automatische Vorrangregelung

  · Möglich, wenn Namenskonflikt 'zufällig'

  $\Rightarrow$ Von Hand Selektion einer Merkmalversion bei Präzedenzkonflikten

  · Nötig bei wiederholtem Erben auf verschiedenen Wegen mit Redefinition

Operationale Vollständigkeit

- **Herkömmliche Anfragesprachen sind unvollständig**
  - Aus Effizienzgründen nur Teilsprache einer Programmiersprache
  - Nicht alle Berechnungen in Datenbanksprache durchführbar
  - Anwendungen erfordern Einbettung in allgemeine Programmiersprache
- **Impedance Mismatch zwischen zwei Sprachen**
  - Verschiedene Typ-Systeme in DB-Sprache und Programmiersprache
    - Nur begrenzte Typ-Prüfung möglich
    - Typkonversion erforderlich
  - Verschiedene Programmierparadigmen: deklarative DML $\leftrightarrow$ prozedurale PL
  - Verschiedene Verarbeitungsformen: mengenorientierte DML $\leftrightarrow$ satzorientierte PL
    - Cursorkonzept erforderlich

$\Rightarrow$ **Umständliche, fehleranfällige Programmierung**

- **Ziel: einheitliche DB-Programmiersprache**
  - Objektorientierte Sprache mit persistenten Datenstrukturen
  - Macht einen Standard erforderlich $\mapsto$ ODMG-93

Das O_2-Objektmodell

- **Hybrides kommerzielles Objektmodell**
  - Notation angelehnt an C^{++} und SMALLTALK
- **Trennung zwischen Objekten und Werten**
  - Werte können über primitive Operationen bearbeitet werden
  - Objekte $\hat{=}$ Paare (Identifikator, Wert)
    - können nur über Methoden einer Klasse bearbeitet werden
  - Benutzer definiert, was Wert oder Objekt sein soll
- **Werte definierbar als Instanzen von Typen**
  - Atomare Werte (int, float, double, char, string, boolean, bit)
  - Objekte (intern dargestellt über Identifikatoren)
  - Zusammensetzbar durch Bildung strukturierter Werte (Listen, Tupel, Mengen)
- **Objekte definierbar als Instanzen von Klassen**
  - Klassen haben Strukturteil (Typdefinition) + Verhaltensteil (Methoden)

```
class Hotel
  type tuple (name:string, address:Address, partners:set(Hotel), rate:float)
  method price(days:int): float
```

  - Methoden können public (defaukt) oder private sein
  - Signatur der Methode wird getrennt vom Rumpf angegeben

```
method body price(days:int):float in class Hotel {return (self→rate)*days
```

Objektorientierter Datenbankentwurf

- **Einstiegspunkt: Abbildung von ER-Modellen**
  - Entity-Typ als Klasse (ggf. Aggregation bei strukturierten Attributen)
  - 1:n Relationship-Typ als Objektreferenz
  - m:n Relationship-Typ als zwei symmetrische Objektreferenzen
  - Relationship-Typ mit eigenen Attributen als Klasse mit Referenzen (wie NWM)
  - Kardinalitätsrestriktionen durch Zugriffsmethoden kontrollierbar
  - IS_A-Beziehung und PART-OF-Beziehung direkt modellierbar
- **Integritätsbedingungen**
  - Lokal: Nutzung von Typrestriktion, Interface-Axiomen
  - Global: Einsatz von Benutzermethoden (Wartung problematisch!)
- **Verhaltenspezifikation direkt im Modell ausdrückbar**
- **Schrittweise Verfeinerung des Entwurfs**
  - Einsatz von Generalisierung, Overriding, Hinzunahme weiterer Methoden

$\Rightarrow$ **Objektorientierte Entwurfsmodelle erforderlich**

- (erweitertes) Entity-Relationship Modell zu ausdruckschwach

$\Rightarrow$ Softwareentwurfstechniken wie OMT als Entwurfshilfe einsetzen

Beurteilung objektorientierter DBMS

- **Vorteile gegenüber relationalen Datenbanken**
  - Adäquatere Modellierung eines Umweltausschnitts (ERM direkter umsetzbar)
  - Leistungsfähige Konzepte für Umgang mit komplexen Objekten
    - Individuelle Methoden für verschiedene Arten von Daten
    - Benutzerdefinierte Typen/Klassen, Methoden und Strukturierungskonzepte
    - Datenkapselung durch abstrakte Datentypen
    - Vererbung, Overloading und Overriding, dynamisches Binden
  - Besondere Stärken bei 'Nichtstandard'-Anwendungen
- **Nachteile**
  - Laden objektorientierte Datenbanken mit Daten noch nicht gut unterstützt
  - Projektionsoperationen auf Objekten erheblich komplizierter
  - Anfrageoptimierung, Synchronisation, Recovery etc. noch deutlich schwächer
  - Namenskonflikte bei multipler Vererbung nicht automatisch auflösbar
  - Bisher kein allgemeines Sichtenkonzept
  - Entwurf komplexer, da Berücksichtigung von Struktur + Methoden
- **Es gibt nicht <u>das</u> Objektmodell** (anders als im RM)
  - Verschiedene Modelle haben individuelle Stärken, Eigenschaften wandeln sich

  $\Rightarrow$ ODMG-Standardisierungsprojekt

  $\Rightarrow$ SQL-3 Projekt übernimmt 'brauchbare' Aspekte in relationales Modell

Grundlagen der Datenbanken

Lektion 12

Das Objektorientierte Datenmodell: ODMG-93

1. Konzeption
2. Typen, Implementierungen und Klassen
3. Objekte, Literale, Struktur
4. Zustand und Verhalten von Objekten
5. Transaktionen, Einbettung in Wirtssprachen

ODMG-93: Standard für Objektorientierte DBMS?

- **Object Data Management Group**
  - Zusammengesetzt aus den wichtigsten kommerziellen OODBMS-Herstellern
  - Ziel: de facto Standard als systemübergreifendes Datenmodell für OODBMS
  - Verpflichtung: Verwirklichung von ODMG-xx 18 Monate nach Erscheinen
- **Eingebettet in OMG Standardisierungsaktivitäten**
  - Entwicklung einer verteilten objektorientierten Betriebssystemarchitektur
  - CORBA: Common Object Request Broker Architecture
  - OMTF/OSTF: Object Model/Services Task Force
- **Bestandteile des ODMG Konzeptes**
  - Object Model: Modell für OODBMS, OOPL + Anwendungen (CORBA Erweiterung)
  - **O**bject **D**efinition **L**anguage: Weiterentwicklung der CORBA IDL
  - **O**bject **Q**uery **L**anguage: deklarative (nicht prozedurale) OODML
  - C++ OML: Anbindung an C++ (OQL+ODL mit C++ Syntax)
  - SMALLTALK-OML: OQL+ODL mit SMALLTALK-kompatibler Syntax
  - Differenzenbeschreibung zu OMG und Adapter für Übersetzung von Objekten
  - Vorschläge für ANSI-C++ Erweiterungen

Typen deklarieren Attribute, Operationen, Beziehungen

- Typ-Eigenschaften: Vererbung, Name des Extents (optional), Schlüsselattribute
- Eigenschaften der Instanzen: Signaturen von Attributen und Beziehungen
- Operationen auf Instanzen: Signatur
- Integritätsbedingungen (Eindeutigkeit, Namen inverser Beziehungen, ...)

```
interface Lecture
 { type properties
      supertype:        Atomic_Object
      extend:           lectures
      key:              (taught_by,lecture_number)
   instance properties
      lecture_number:   String; // unique
      days_offered:     Set<Struct<day:Weekdays, from:Time, duration:Time>>;
      students:        Set<Students>  inverse Student::take;
      taught_by:        Professor      inverse Professor::teaches;
   instance operations
      cancel ();
      reschedule(from:Struct<date:Date,time:Time>,
                 to:Struct<date:Date,time:Time>)
 }
```

Typen, Implementierungen und Klassen

- **Ein Typ kann mehrere Implementierungen haben**
  - Objektrepräsentation durch Datenstrukturen für alle Attribute
  - Prozedurkörper (Methoden) für alle Operationen
  - Referenzen (Links) für Beziehungen
  - Implementierungen werden benannt
- **Klassen $\hat{=}$ Typ-Interface und eine Implementierung**
  - Mehrere Klassen für ein Typ-Interface möglich
  - Ermöglicht heterogene Systeme ohne Mißbrauch der Vererbung
- **Abstrakte (virtuelle) Typen $\hat{=}$ Typen ohne Instanzen**
  - Definieren Charakteristika eines Typs (Attribute, Beziehungen, Operationen)
  - Bieten keine vollständige Implementierung an
  - Können keine eigenen Instanzen haben
  - Liefert einheitliches Interface für mehrere Subtypen
    - Subtyp ergänzen Implementierung, die das Interface erweitert

1. Modellierung abstrakten Begriffs, der nur in speziellen Ausprägungen auftritt
   - Keine Attribute, aber einheitliche Deklaration erlaubter Operationen
2. Modellierung einer universellen Datenbank ohne eigene Zugriffe
   - Alle Attribute — Operationen nur in Subklassen (Sichten) implementiert
3. Vereinfachung eines Modelles durch künstlichen Oberbegriff als Interface

Grundkonstrukte objektorientierter Modellierung

- **Mutable Object $\hat{=}$ veränderbare Objekte**
  - Objekte mit unveränderlicher Identität, Zustand und Verhalten
  - Zustand (Attributwerte und Teilnahme an Beziehungen) ist veränderbar
  - Zustände sind atomar oder strukturiert
  - Identifikation durch interne OID (oder Beschreibung von Schlüsselwerten)
  - Objekte werden erzeugt und haben eine Lebenszeit
- **Literale (immutable Objects) $\hat{=}$ unveränderbare Werte**
  - Atomare oder strukturierte Sammlung von Daten
    - Atomar: `Integer, Float, Boolean, Character`
    - Sammlung: `Set<`L`>, Bag<`L`>, List<`L`>, Array<`L`>, Enumeration (`$e_1,..,e_n$`), String, Bit_String`
    - Struktur: `Structure<`a_1`:`$L_1,..,a_n$`:`L_n`>, Date, Time, Interval,...`
    - Benutzer darf eigene Untertypen von Literalen erzeugen
  - Literale werden nicht erzeugt sondern existieren in sich
  - Veränderung der Werte ändert die Identität ($\hat{=}$ ein anderes Literal 'entsteht')
  - Kein Objekt im eigentlichen Sinne (keine echte Objektidentität)

Grundeigenschaften von Objekten

- **Vordefinierte Eigenschaften und Operationen**
  - `has_name?: Boolean`
  - `names: Set<String>`
  - `type:Type`
  - `create()`→`oid:Object_id`: weist Speicherplatz zu, generiert OID
  - `delete()`: entfernt Objekt aus Datenbank, Beziehungen, und Extent
  - `same_as?(oid:Object_id)`→`b:Boolean`:
    - testet (flache) Gleichheit mit anderem Objekt
  - `equal?(`o_1`:Object,`o_2`:Object)`→`b:Boolean`:
    - Identitätstest, definiert für Objekte und Literale
- **Lebensdauer von Objekten**
  - coterminus-with-procedure: nur zur Laufzeit einer Operation existierend
  - coterminus-with-process: wird im Prozess generiert
  - coterminus-with-database:
    - dauerhaft, Laufzeitspeicher wird von DBMS zugewiesen

Strukturierte Objekte

- **Sammlung (Datenbehälter)**
  - Beliebige Anzahl von Elementen ohne eigene Namen
  - Zugriff (Einfügen, Entfernen) an fester Position (Anfang, Ende, Iteratorposition)
  - Geordnet oder ungeordnet, mit und ohne Duplikate
  - Vordefiniert: `Set<`T`>, Bag<`T`>, List<`T`>, Array<`T`>`
  - Benutzerdefinierte Sammlungen durch parametrische Typen
- **Struktur**
  - `Structure<`a_1`:`T_1`, . . . ,`a_n`:`T_n`>`
  - Feste Anzahl benannter Felder für Objekte oder Literale
  - Zugriff auf Felder (Einfügen, Entfernen) durch Feldnamen im Pfadausdruck
  - Kopieroperation ist flache Kopie
- **Strukturierte Objekte haben unveränderliche Identität**
  - Identität eines Mengenobjekts bleibt, auch wenn Elemente sich ändern
  - Mathematische Sicht entspricht immutable collection / immutable structure
    - Strukturierte Konstanten (unveränderlich)
- **Beliebige Komposition von Strukturen erlaubt**
  - Mengen von Mengen von Studenten

Benutzerdefinierte Strukturen

- **Parametrische Typen**
  - `type` *collection*`<`T`>` { `...el:`T `...`}
  - Definiert Kollektion von Elementen eines Typs T
    - Prädikative Definition einer Teilmenge des Extent von T
    - oder benutzerdefinierte Einfüge- und Zugriffsoperationen
  - Typprüfung zur Compile-Zeit möglich
- **Iteratoren (Cursor) in Sammlungen**
  - Verwaltung einer aktuellen Zugriffsposition im Datenbehälter

  $\Rightarrow$ aktive Gestaltung des Datenbehälters
  - Iterator kann Sammlung vorwärts, rückwärts oder beliebig durchlaufen
  - Vordefinierte Eigenschaften und Operationen
    - `stable?:Boolean`
    - `iteration_order:Enumeration(fwd,bwd)`
    - `next()`$\rightarrow$`el:`T
    - `first()`$\rightarrow$`el:`T
    - `last()`$\rightarrow$`el:`T
    - `more?`$\rightarrow$`b:Boolean`
    - `reset()`
    - `delete()`::

Modellierung von Objektzuständen

Ein Zustand ist beschrieben durch Attributwerte und Beziehungen

- **Attribute beschreiben abstrakte Zustände**
  - Attributwertebereiche stellen mögliche Werte dar
    $\Rightarrow$ Attribute sind Teil des abstrakten Typ-Interfaces
  - Implementiert als Datenstrukturen oder Methoden ($\hat{=}$ abgeleitete Attribute)
  - Vordefinierte Operationen
    - `set_value(new:Literal)`: Änderung des Zustands ($\hat{=}$ flache Kopie)
    - `get_value()`$\rightarrow$`val:Literal`: Lesen des Zustands
- **Beziehungen**
  - Nur binäre Beziehungen zwischen Objekten
    - Dargestellt als benannte Traversionsfunktion
    - Traversion in beide Richtungen gekennzeichnet durch `inverse`
  - Referenzintegrität wird vom System gewartet
  - Vordefinierte Operationen
    - `delete, add_one_to_one, add_one_to_many, remove_all_from, ...`
  - Objektwertige Attribute als Beziehung modellieren?

Modellierung von Verhalten

Das Verhalten eines Objekts wird durch Operationen beschrieben

- **Operationen**
  - Können nicht losgelöst von einem Objekttyp existieren
  - Werden durch ihre Signatur (Name, Ein- und Ausgabetypen) beschrieben
  - Namen müssen nur innerhalb eines Typs eindeutig sein
    - Überladen ist möglich
    - Dynamisches Binden an den spezifischsten Typ eines Objekts
  - Ausnahmebehandlung und -erzeugung ist möglich
    - Kontrollierte Verarbeitung ungewöhnlicher (Fehler-)situationen
- **Vordefinierte Operationen auf Operationen**
  - `invoke()`: Auslösen der Operation
  - `return()`: Beenden der Operation
  - `return_abnormally(e:Exception)`: Beenden mit Ausnahmezustand

Struktur des ODMG-Modells

Darstellung als Vererbungshierarchie von Meta-Typen

Denotable_Object
- *Object*
  - *Atomic_Object*
    - Type | Exception | Iterator
  - *Structured_Object*
    - *Collection*<T>
      - Set<T> | Bag<T> | List<T> | Array<T> | String | Bit_String
    - Structure<a_1:T_1,..,a_n:T_n>
- *Literal*
  - *Atomic_Literal*
    - Integer | Float | Character | Boolean
  - *Structured_Literal*
    - *Immutable_Collection*<T>
      - Immutable_Set<T> | Immutable_Bag<T> | Immutable_List<T> | Immutable_Array<T>
      - Immutable_String | Immutable_Bit_String
    - Immutable_Structure<a_1:T_1,..,a_n:T_n>
      - Date | Time | DateTime | Interval
- *Characteristic*
  - *Property*
    - Attribute | Relationship
  - Operation

Begriffe in Schrägschrift sind abstrakt und nicht direkt instantierbar

Transaktionen im ODMG Modell

- **Manipulation persistenter Daten**
  - Innerhalb von Transaktionssgrenzen
    - `Transaction::begin()->t:Transaction ...t.commit()`
  - Limitiert Fremdzugriffe auf Objekte der Klasse bis zum `commit`-Befehl
- **Geschachtelte Transaktionen möglich**

```
Transaction::begin()->t:Transaction
  ...
  Transaction::begin()->x:Transaction
    ...
    Transaction::begin()->y:Transaction
      ...
      if minor_error y.abort()
      if major_error y.abort_top_level()
      ...
    y.commit()
    ...
  x.commit()
  ...
t.commit()
```

  - Nur hierarchische Schachtelungen möglich
  - ODMG-Modell verwaltet die Anwendungsbereiche der Transaktionen

Details $\mapsto$ *Lektionen über Transaktionen*

Einbettung in C++

ODMG Modell ist (fast) kompatibel mit C++

- **ODMG Objekttypen**
  - Lassen sich direkt auf C++-Klassen abbilden
  - Umkehrung: Instanzen von C++-Klassen müssen analysiert werden
    - Abbildung in ODMG Objekt oder ODMG Literal
    - Instanzen einer C++-Top-Level Klasse werden als Literale behandelt
  - Nur eine C++ Implementierung pro Interface erlaubt
- **Beziehungen müssen in C++ simuliert werden**
  - Darstellung durch Methoden, welche Traversionsfunktionen implementieren
    - 1:1 Beziehungen als Objektreferenzen dargestellt
    - 1:n Beziehungen als Sammlung von Objektreferenzen
- **Extents und Schlüssel in C++ nicht unterstützt**
  - Extent: Benutzer muß Sammlung der Instanzen definieren und selbst verwalten
  - Schlüssel müssen durch Strukturen und Indexe direkt verwaltet werden
- **Arrays in C++ nicht direkt vorhanden**
  - Müssen durch indizierte Folge von Objekten simuliert werden
- **Statische Einbettung in C++ als Wirtssprache**
  - ODL Preprocessor erzeugt ODMG-Prozeduraufrufe

ODMG: Zusammenfassung

- **Ehrgeiziges Standardisierungsprojekt**
  - Wirtschaftliches, nicht (nur) wissenschaftliches Interesse
  - 18-Monate Verpflichtung für ODMG-93 nur teilweise eingehalten
- **Noch unausgereift**
  - Standard entspricht noch nicht dem Stand der Forschung
  - Noch keine ernstzunehmende Konkurrenz des SQL-Standards
  - Anfragesprache OQL sehr gewöhnungsbedürftig
- **Geplante Erweiterungen**
  - Objekte sollen zu mehreren Typen gehören dürfen (Objektmigration)
    - z.B. wenn Student nach Diplom zum Mitarbeiter wird
  - Automatische Verwaltung von Subextents (Anfrage-Ergebnisse) als Sichten
  - Hinzunahme allgemeinerer Integritätsbedingungen
  - Default-Werte für Attribute
  - Semantische Eigenschaften von Beziehungen (z.B. transitiv/reflexiv)

  $\mapsto$ ODMG-9x ???

Grundlagen der Datenbanken

Lektion 13

Transaktionsmanagement I

1. Transaktionen
   - Grundkonzepte
   - Zustände und Kontrollfluß
   - Konsistenzgrade
2. Datensicherung – Recovery
   - Aufgabe und Klassifizierung
   - Physische Protokolle
   - Logische Protokolle

- **Sicherung der Datenbank-Konsistenz**
  - Datensicherung (Recovery): Konsistenzsicherung beim Auftreten von Fehlern
  - Concurrency Control: Synchronisation nebenläufiger Transaktionen
    - z.B. Platzreservierung in Flugzeugen, Zugriff während Statistikberechnung
- **Transaktion**
  - Minimale (atomare) Prozeßeinheit im DB-System
    - führt DB von konsistentem Zustand in neuen konsistenten Zustand über
  - Gekennzeichnet durch Anfangs- und Endmarken
    - BOT(T), EOT(T): Begin/End of Transaction T
  - Beliebige korrekte DML/PL-Statements zwischen BOT- und EOT-Marke
  - `commit`: Normales Transaktionsende, Änderungen permanent in DB
  - `abort`: Anormales Ende, BOT-Zustand wiederherstellen (Rollback)
    - Kann vom Benutzer, Anwenderprogramm oder System veranlaßt werden
- **2 Arten von Konsistenz**
  - Datenbankkonsistenz: alle auf DB definierte Konsistenzbedingungen gelten
  - Transaktionskonsistenz: korrekter Ablauf nebenläufiger Transaktionen
  - DB muß vor Anfang und nach Ende der Transaktion konsistent sein

Eigenschaften von Transaktionen

ACID: Atomicity + Consistency + Isolation + Durability

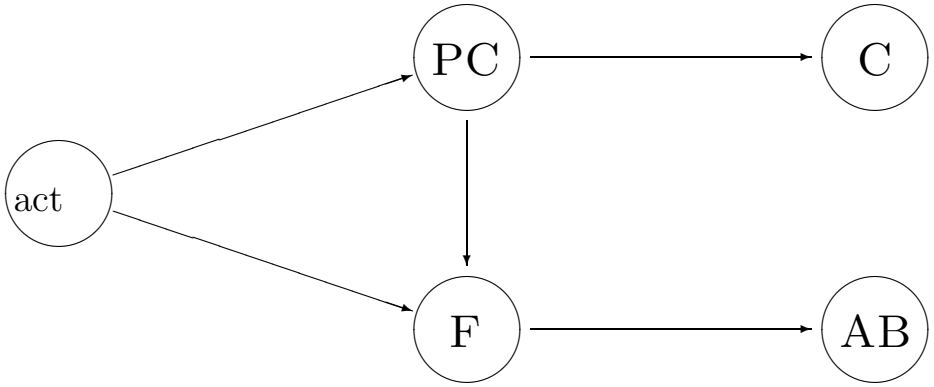
- **Atomicity $\hat{=}$ Ununterbrechlichkeit**
  - Alle Aktionen der Transaktion werden ausgeführt oder keine
  - Zwischenzustände dürfen auch im Fehlerfall nicht hinterlassen werden
- **Consistency $\hat{=}$ Konsistenzerhaltung**
  - Transaktion ist 'korrektes' Programm (bzgl. BOT/EOT)
  - Datenbank darf zwischen BOT und EOT inkonsistent sein
  - Konsistenzerhaltung bzgl. Nebenläufigkeit muß garantiert werden
- **Isolation $\hat{=}$ Isolierter Ablauf**
  - Jede Transaktion muß konsistenten DB-Zustand sehen
  - Transaktionsergebnisse erst am Ende sichtbar machen (`commit`)

  $\not\rightarrow$ lost updates: Änderungen können verloren gehen
  - Fremdtransaktion liest alten Wert, überschreibt geänderten Wert

  $\not\rightarrow$ cascading abort: Eingabedaten für Fremdtransaktionen bei Abbruch ungültig
  - Fremdtransaktion müsste ebenfalls zurückkgesetzt werden
- **Durability $\hat{=}$ Dauerhaftigkeit der Ergebnisse**
  - Ergebnisse müssen permanent in DB gespeichert sein

Ablauf einer Transaktion

- **Transaktion kann 5 Zustände annehmen**
  - Active: laufende Transaktion
  - Pre-Commit: gelaufen, aber noch nicht permanent
  - Failed: fehlgeschlagen, aber noch nicht zurückgesetzt
  - Committed: permanent beendet
  - Aborted: fehlgeschlagen und zurückgesetzt



- **Zurückgesetzte Transformationen können**
  - neu gestartet werden: wenn Abbruch durch Hardware-/Systemfehler
  - entfernt werden: fehlerhafte Transaktion wird eliminiert

Kontrollfluss zwischen AP und DBMS

Anwenderprogramm		**DBMS**
BOT	$\longrightarrow$	Vorbereitende Maßnahmen für
	$\longleftarrow$	Rücksetzbarkeit v. Änderungen
Folge korrekter DML-Befehle	$\longrightarrow$	Ausführung der DML Befehle
		Überprügung unverzögerter Integritätsbedingungen
	$\leftarrow$	(ggf. Fehlermeldung)
EOT	$\longrightarrow$	Überprügung verzögerter Integritätsbedingungen
	$\leftarrow$	ggf. Fehlermeldung + Rücksetzen)
		falls OK
		Sicherstellung der Wiederholbarkeit aller Änderungen der Transaktion
		Aufhebung von Rückstellungsmaßnahmen
	$\longleftarrow$	Bestätigung des `commit`
Weiterarbeit AP		

4 Konsistenzgrade für Transaktion T

- **Grad 3 Konsistenz (strict):**
  - T überschreibt keine schmutzigen Daten anderer Transaktionen
    - Dirty Data (schmutzige Daten) $\hat{=}$ Daten vor einem `commit`-Befehl
  - T liest keine schmutzigen Daten anderer Transaktionen
  - T macht alle write-commits am EOT
  - Fremdtransaktionen beschmutzen keine von T gelesene Daten vor dem commit
- **Grad 2 Konsistenz (no cascading aborts):**
  - T überschreibt keine schmutzigen Daten anderer Transaktionen
  - T liest keine schmutzigen Daten anderer Transaktionen
  - T macht keine commits vor dem EOT
- **Grad 1 Konsistenz:**
  - T überschreibt keine schmutzigen Daten anderer Transaktionen
  - T macht keine commits vor dem EOT
- **Grad 0 Konsistenz (no lost updates):**
  - T überschreibt keine schmutzigen Daten anderer Transaktionen

Datensicherung: Aufgaben

- **Wiederherstellung von DB-Konsistenz im Fehlerfall**
  - Logische Fehler: Input- und arithmetische Probleme, fehlende Daten, ...
    - Transaktion wird gestoppt ohne neu zu starten
  - Systemfehler: Deadlock, ...
    - Transaktion wird gestoppt und automatisch neu gestartet
  - System-Ausfall: Inhalt von Hauptspeicher verloren / verfälscht
    $\Rightarrow$ Recovery mit Log-Buch, Schattenspeicer, Recovery Protokoll, ...
  - Platten-Crash: Plattenfehler beim Schreiben, Head crash, unlesbare Blöcke
    $\Rightarrow$ Recovery mit Backups, Archiven, Plattenspiegelung, ...

- **Transaktionsmanagement unabhängig von Pufferverwaltung**
  - Daten beendeter Transaktionen können noch im Puffer sein
  - Daten unbeendeter Transaktionen können schon in der Datenbank sein

- **Recovery Manager muß garantieren**
  - Transaktionen gehen nicht verloren
  - Transaktionen werden nicht nur teilweise abgearbeitet
  - Effekt eines REDO ist der eines einmaligen Ablaufs
  - Datenbank jederzeit konsistent (relativ zum Transaktionsablauf)

Voraussetzungen für Recovery

- **Redundante Informationsspeicherung**
  - Archiv-Kopien (meist Bänder) für permanente Medien
    - müssen regelmäßig angefertigt werden (täglich)
    - wie bei Backup-Erstellung durch Betriebssysteme
  - Protokolldateien (Log-Buch)
    - Historie: Änderungen in Datenbank / Statusänderungen von Transaktionen
    - temporäre Protokolldatei oder Archiv-Protokolldatei
    - phyische (Speichereinheit) oder logisch (Parameter)
- **Log-Buch Einträge**
  - BOT(T): Beginn einer Transaktion `T`
  - A(Z,T): Zustand nach einer Änderung durch Transaktion `T` (After-Image)
    - bezogen auf Datenbankobjekte oder physische Seiten
    - wird für effizientes REDO benötigt
  - B(Z,T): Zustand vor einer Änderung durch Transaktion `T` (Before-Image)
    - wird für effizientes UNDO benötigt
  - EOT(T): Erfolgreiches Ende der Transaktion `T` (mit `commit`)
  - Informationen müssen Recovery im schlimmstmöglichen Fall ermöglichen
    - abhängig von Art des Recovery

Checkpointing

- **Protokollierung konsistenter DB-Zustände**
  - Regelmäßige Absicherung eines stabilen Zustands auf externem Speicher
  - Gesamter Pufferinhalt wird auf Platte gezwungen
  - aktionskonsistent: alle aktiven Transaktionen werden blockiert

  $\Rightarrow$ materialisierte Datenbank enthält alle geschriebenen Seiten

- **Verfahren zur Erzeugung von Checkpoints**
  - Protokolldatei-Puffer wird in permanente Protokolldatei geschrieben
  - Checkpoint-Record wird in permanente Protokolldatei geschrieben
  - Datenbank-Puffer werden auf permantente Medien gezwungen
  - Adresse des Checkpoint-Record wird in permanentes Restart-File geschrieben

- **Checkpoint-Record**
  - Liste aller unbeendeten Transaktionen zum Zeitpunkt des Checkpoint
    - ermöglicht korrektes Wiederaufsetzen im Fehlerfall
  - Adresse des letzten Protokollsatzes für jede aktive Transaktionen

Recovery-Verfahren

- **Backward-Recovery (UNDO)**
  - Rücksetzen von Änderungen bis Konsistenz erreicht
  - Benötigt Before-Image aller gestarteten Transaktionen
  - Neuausführung mit `restart` möglich
- **Forward-Recovery (REDO)**
  - Nachvollziehen erfolgreicher Transaktionen (ohne `restart`)
  - Benötigt After-Image der mit `commit` beendeten Transaktionen
- **Generelle Vorgehensweise**
  - Suche im Log-Buch letzten konsistenten Zustand der permanenten Datenbank
  - Stelle Zustand durch unbeendeter Transaktionen wieder her
    - UNDO-Schritte beim Rückwärtslesen des Log-Buchs
    - Verwendet Daten aus Before-Images
  - Mache Effekte erfolgreicher Transaktionen permanent
    - REDO-Schritte beim Vorwärtslesen des Log-Buchs
    - Verwendet Daten aus After-Images
  - Neustart unbeendeter Transaktionen oder Warnung

Recovery-Klassen

- **R1-Recovery (partielles Zurücksetzen)**
  - Bei logischen Fehlern, `abort` oder Deadlock
  - Isoliertes Rücksetzen einzelner Transaktionen
- **R2-Recovery (partielles Wiederholen)**
  - Bei Systemausfall (Zielzustand beendeter Transaktionen ist konsistent)
  - REDO abgeschlossener Transaktionen, deren Daten nur im Puffer waren
- **R3-Recovery (vollständiges Zurücksetzen)**
  - Bei Systemausfall (Zielzustand unbeendeter Transaktionen ohne Wirkung)
  - In DB ausgelagerte Daten laufender Transaktionen werden entfernt
- **R4-Recovery (vollständiges Wiederholen)**
  - Bei Defekt persistenter Externspeicher
  - Kopieren einer Archiv-Kopie auf neuen Datenträger
  - REDO aller Transaktionen seit letzter Transaktion auf Archiv-Kopie

Physisches Protokollieren

- **Protokoll physischer Speichereinheiten**
  - Before-Image $\hat{=}$ Seite, die zum Ändern in Puffer geholt wird
    - Nur älteste Version (bezogen auf Transaktion) für UNDO nötig
    - Meist Speicherung nur in temporärer Protokolldatei
  - After-Image $\hat{=}$ Seite, die in Datenbank zurückgeschrieben wird
    - Meist Speicherung in temporärer und permanenter Protokolldatei
- **Protokoll mit direkter Seitenadressierung**
  - Pufferseite der Datenbank wird bei Änderungen direkt überschireben
  - Volles Before-Image muß vor Änderung in Protokolldatei geschrieben werden
- **Protokoll mit indirekter Seitenadressierung**
  - Seitentabelle verweist auf physikalische (meist permanente) Pufferseiten
  - Änderungen werden in freie Pufferseite geschrieben und Seitentabelle geändert
  - Vorteil: Before-Image besteht nur aus alter Seitentabelle
  - Nachteil: Speicher wird fragmentiert (kein Clustering), Garbage collection nötig
- **Schattenspeicher-Verfahren**
  - Permanenter Schattenspeicher enthält Kopie des aktuellen Puffers
  - Aktueller Puffer wird durch Transaktionen verändert
  - Puffer wird bei Transaktionsende im Schattenspeicher gesichert
  - Effizient im Zusammenhang mit indirekter Seitenadressierung

Logisches Protokollieren

- **Protokoll logischer Parameter von Änderungen**
  - Transaktion, Datenbank, Relation, Record
  - Stand der 'Pointer' vor Transaktionsbeginn
  - Geänderte Felder: Id, alter Wert, neuer Wert
- **Elegant und logisch sauber**
  - DML-Statements können logisch nachvollzogen werden
  - Gut geeignet im Zusammenhang mit relationalen Datenbanken
- **Problematisch**
  - Inverse Operation (UNDO) zu DML-Statements nicht trivial (außer RM)
  - R2/R4-Recovery nur im Einbenutzerbetrieb möglich
    - um genauen Zustand der Datenbank wiederherstellen
  - Datenbank muß vor Recovery in speicherkonsistentem Zustand sein
    - Kombination mit Schattenspeicherverfahren

Grundlagen der Datenbanken

Lektion 14

Transaktionsmanagement II: Concurrency Control

1. Serialisierbarkeit paralleler Transaktionen
2. Sperrung von Fremdzugriffen
   - Zweiphasen-Sperrprotokolle
   - Behandlung von Deadlocks
   - Hierarchisches Sperren

Korrektheit paralleler Transaktionen

- **Gleichzeitige Transaktionen sind Normalfall**
  - DBMS muß konsistenten Ablauf sicherstellen
    - Aufstellung von (verschränkten) Ablaufplänen
    - Korrektheit: paralleler Ablauf muß einem seriellen entsprechen
  - Wichtig: Reihenfolge der Lese- und Schreibzugriffe

- **Schedule: (verschränkter) Ablaufplan für Transaktionen**

T_1	T_2	—	T_1	T_2
read(A)		—	1000	
A:=A-50		—	950	
write(A)		—	950	
	read(A)	—		950
	tmp:= A*0.1	—		95
	A:=A-tmp	—		855
	write(A)	—		855
read(B)		—	2000	
B:=B+50		—	2050	
write(B)		—	2050	
	read(B)	—		2050
	B:=B+tmp	—		2145
	write(B)	—		2145

  - Bei n Transaktionen sind n! serielle Schedules möglich
  - Ergebnisse unterschiedlicher Schedules können verschieden sein
  - $S_1 \hat{=} T_1;T_2$: (1000,2000) $\mapsto$ (950,2050) $\mapsto$ (855,2145) — A+B=3000
  - $S_1 \hat{=} T_2;T_1$: (1000,2000) $\mapsto$ (900,2100) $\mapsto$ (850,2150) — A+B=3000

Serialisierbarkeit

- **Korrekte Ausführung nicht garantierbar**
  - Nicht alle parallelen Schedules sind serialisierbar

T_1	T_2	—	T_1	T_2
read(A)		—	1000	
A:=A-50		—	950	
	read(A)	—		1000
	tmp:= A*0.1	—		100
	A:=A-tmp	—		900
	write(A)	—		900
	read(B)	—		2000
write(A)		—	950	
read(B)		—	2000	
B:=B+50		—	2050	
write(B)		—	2050	
	B:=B+tmp	—		2100
	write(B)	—		2100

  - "Lost update" Problem wegen Lesen schmutziger Daten
- **Äquivalenz von Schedules: $S_1 \equiv S_2$**
  - S_1 und S_2 enthalten genau dieselben Transaktionen T_1,..,T_n
  - In S_1 liest Transaktion T_i den Wert von Objekt X, der von T_j geschrieben wurde, gdw. dies auch in S_2 gilt
  - Wurde für X in S_1 der letzte Wert von T_i geschrieben dann auch in S_2
- **Serialisierbarkeit eines parallelen Schedules S**
  - es gibt einen seriellen Schedule S' mit S ≡ S'

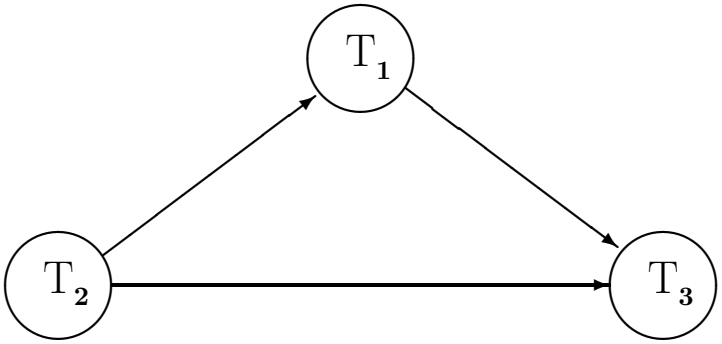
Testen auf Serialisierbarkeit

- **Uneingeschränktes Schreiben**
  - Ein Datenbankobjekt darf jederzeit beschrieben werden
  - Allgemeine Serialisierbarkeitsprüfung ist NP-vollständig

  $\Rightarrow$ Kein effizienter Test auf Serialisierbarkeit

- **Schreiben nur nach Lesen**
  - Ein Datenbankobjekt muß vor dem Überschreiben gelesen worden sein

  $\Rightarrow$ Serialisierbarkeit von S $\hat{=}$ Präzedenzgraph von S zyklenfrei
  - Zyklentest in gerichteten Graphen mit n Knoten in $\mathcal{O}(n^2)$
  - Zyklenfreier Präzedenzgraph liefert möglichen sequentiellen Ablauf



Serieller Schedule: $T_2 - T_1 - T_3$

- **Präzedenzgraph eines Schedule S**
  - Darstellung von Transaktionen T_i als Knoten
  - Kante $T_i \rightarrow T_j$ falls T_i ein Objekt Q beschreibt bevor T_j es liest
  - Kante $T_i \rightarrow T_j$ falls T_i ein Objekt Q liest bevor T_j es beschreibt

Realisierung von Concurrency Control

- **Exklusiver Zugriff durch Sperren von Datenobjekten**
  - Paralleler (unveränderter) Schedule beim tatsächlichen Ablauf
  - Objekte werden vor Benutzung gesperrt und danach freigegeben
  - Befehle `lock` / `unlock` ($\mapsto$ Betriebssysteme / parallele Programmierung)
- **Granularität von Sperren**
  - Größe der gesperrten Objekte (Tupel / Relationen / Datenbanken, ...)
  - Grobe Granularität leicht zu verwalten
  - Feine Granularität ermöglicht hohes Maß an Parallelität
- **Arten von Sperren (Sperr-Modi)**
  - Exclusive: Fremdzugriff verboten, da Schreiben beabsichtigt
  - Shared: Fremdzugriff erlaubt, da nur Lesen vorgesehen
  - Kurze Sperre: Sperre wird unmittelbar nach letztem Gebrauch freigegeben
  - Lange Sperre: Sperre wird bis zum `commit` gehalten
- **DBMS verwaltet Sperren**
  - Kompatibilitätsprüfung bei Anforderung von Sperren
  - Verwendet Lock Table für jedes Objekt und Kompatibilitätsmatrix

$T_1 \backslash T_2$	S	X
S	+	-
X	-	-

- **Konsistenzebenen einer Transaktion T**

  0: kurze X-Sperren auf veränderten Objekten

  1: lange X-Sperren auf veränderten Objekten

  2: lange X-Sperren auf veränderten, kurze S-Sperren auf gelesenen Objekten

  3: lange X-Sperren auf veränderten, lange S-Sperren auf gelesenen Objekten

  – Kommerziell: meist Ebene 2 + Möglichkeit, Ebene 3 zu erzwingen

- **Fundamentalsatz des Sperrens**

  – **5 Voraussetzungen garantieren Serialisierbarkeit**

  – Jedes von einer Transaktion benutzte Objekt wird zuvor entsprechend gesperrt

  – Jede Transaktion beachtet die Sperren der anderen Transaktionen

  – Keine Transaktion fordert eine Sperre, die sie hat, nochmals an

  – Jede Transaktion gibt alle Sperren bis spätestens zum `commit` zurück

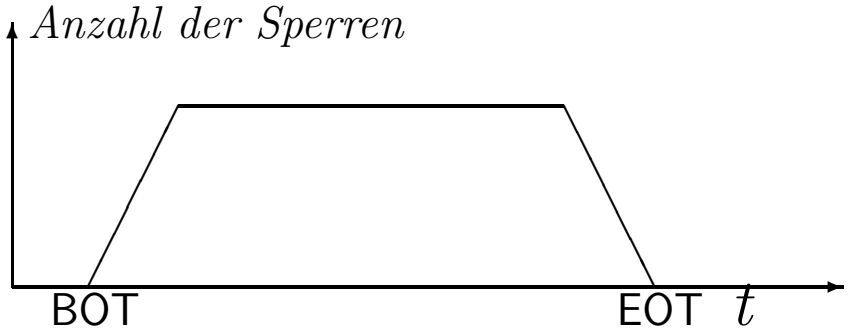
  – Jede Transaktion folgt dem Zweiphasensperrprotokoll (2PL)

  · Sperren werden nur in einer Wachstumsphase angefordert

  · Sperren werden nur in einer Schrumpfphase aufgehoben

  $\Rightarrow$ Kein Test auf Serialisierbarkeit zur Laufzeit erforderlich

Zweiphasen-Sperrprotokoll (2PL)

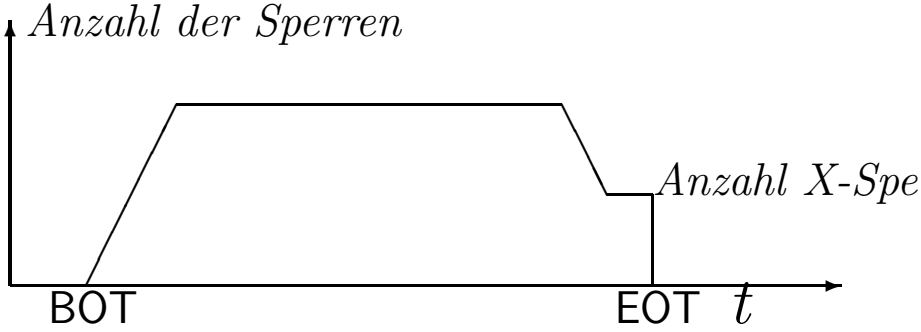


- Wachstumsphase: Sperren werden angefordert, aber keine freigegeben
- Schrumpfphase: Sperren werden freigegeben, keine neuen angefordert

- **Ausreichend für korrekten Transaktionsablauf**
  - Zyklus im Sperrgraphen $\hat{=}$ Verletzung des 2PL
  - 2PL auch notwendig für Serialisierbarkeit

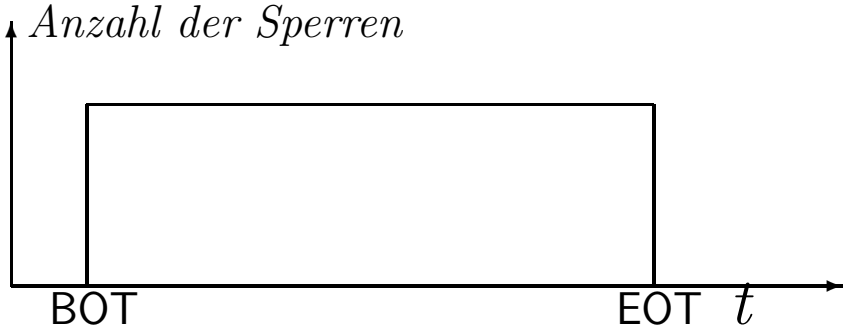
- **Mögliche Probleme**
  - Kaskadierender Abbruch bei Systemausfall ($\mapsto$ Striktes 2PL)
    - UNDO beendeter Transaktionen, die freigegebene Daten gelesen haben
  - Deadlock: Gegenseitige Blockade durch Anforderung von Betriebsmitteln
  - Phantom Problem ($\mapsto$ Hierarchisches Sperren)
    - Lesende Transaktion T_1 sieht durch T_2 gesperrte Tupel nicht

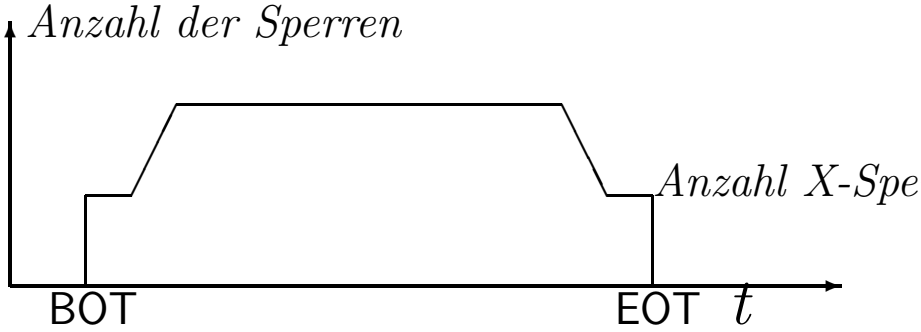
Vermeide Cascading Abort



1. Alle Sperren dürfen nicht vor dem `commit` freigegeben werden
   - Anforderung unnötig streng
2. Alle X-sperren dürfen nicht vor dem `commit` freigegeben werden

- **Striktes 2PL mit Preclaiming**
  - Sperranforderung symmetrisch zur Freigabe
  - Alle (X-)Sperren werden zu Beginn der Transaktion angefordert





Deadlocks

- **Parallele Transaktionen benötigen Betriebsmittel**
  - T_1 hält X-Sperre auf A und benötigt B zum Beenden
  - T_2 hält X-Sperre auf A und benötigt B zum Beenden
  - Zyklischer Wartegraph
- **Voraussetzung für Entstehung von Deadlocks**
  - Paralleler Ablauf von Transaktionen
  - Betriebsmittel mit X-Sperren anforderbar
  - Transaktionen besitzen X-gesperrte Betriebsmittel und fordern weitere an
  - Transaktionen geben Betriebsmittel nicht vorzeitig frei
  - Es bestehen zyklische Wartebeziehungen
- **Behandlung von Deadlocks** ($\mapsto$ Betriebssysteme)
  - Verhinderung zyklischer Wartebeziehungen beim Scheduling
  - Aufbrechen entstandener Zyklen durch Rücksetzen einer Transaktion
    - Verlangt effiziente Zyklenerkennung (nicht immer möglich)
  - Time-Out für Laufzeit oder Inaktivität einer Transaktion
  - Vergabe von Zeitstempeln passend zu einem möglichen seriellen Ablauf
    - Transaktionen, die zu früh auf Betriebsmittel zugreifen, werden zurückgesetzt

Hierarchisches Sperren

- **Probleme bei einfachen X-/S-Sperren**
  - Aufwendige oder ineffiziente Verwaltung
    - Große Sperrtabelle bei feiner Granularität
    - Wenig Parallelität bei grober Granularität
  - Phantom Problem: Anomalie zwischen Lese- und Schreibttransaktionen
    - Lesende Transaktion T_1 sieht durch T_2 gesperrte Tupel nicht
    - Lesende Transaktion T_1 bemerkt Veränderungen der Gesamtrelation nicht

    z.B. T_1 summiert Gehälter und vergleicht mit explizit gespeichertem Wert

    T_2 fügt neuen Angestellten ein und ändert explizit gespeicherten Wert

    $\Rightarrow$ Scheinbare Inkonsistenz, wenn T_2 während T_1 abläuft

    Konflikt zwischen T_1 und EOT(T_2) über einfache X-/S-Sperren nicht erkennbar

- **Hierarchisches Sperren von Betriebsmitteln**
  - Sperren eines Tupels sperrt übergeordnete Betriebsmittel
    - Relation, Datenbank
  - Macht neue Sperrmodi nötig
    - X-Sperre würde sonst Einbenutzerbetrieb erzwingen
    - S-Sperre würde sonst jede gleichzeitige Änderung verbieten

Sperrmodi beim hierarchischen Sperren

IS: Absicht, niedrigere Objekte zu lesen (für diese ist IS/S vorgesehen)
IX: Absicht, niedrigere Objekte zu ändern (IS/IX/S/SIX/X vorgesehen)
S: Absicht, Knoten und Nachfolger zu lesen
SIX: Absicht, Knoten zu lesen und Nachfolger (IX/SIX/X vorgesehen) zu ändern
X: Absicht, Knoten und Nachfolger zu ändern

Kompatibilitätsmatrix

$T_1 \backslash T_2$	IS	IX	S	SIX	X
IS	+	+	+	+	-
IX	+	+	-	-	-
S	+	-	+	-	-
SIX	+	-	-	-	-
X	-	-	-	-	-

- **Konsistentes Sperren bei exklusivem Zugriff**
  - X-Sperre auf Tupel $\Rightarrow$ IX-Sperre auf Relation und Datenbank
  - Verhindert S/X/SIX-Sperre auf Relation und Datenbank
- **Verhinderung des Phantomproblems**
  - S-Sperre auf Relation, IS-Sperre auf Datenbank
  - Blockiert X-Sperre auf Tupel der Relation
  - Blockiert Änderungen der Relation durch Einfügen oder Löschen

Transaktionsmanagement: Zusammenfassung

- **ACID Konzept ermöglicht saubere Handhabung**
  - keine schmutzigen Daten, Änderungsverluste oder Datenverluste
  - Konsistenzbedingungen für Fehlersituationen und Parallelität formulierbar
- **Recovery**
  - Verdeckung zu erwartender Fehlersituationen
  - Führung eines Log-Buchs als Voraussetzung (physisch oder logisch)
  - UNDO unbeendeter Transaktionen, REDO vollständiger Transaktionen
  - Checkpointing zur Effizienzsteigerung
- **Concurrency Control**
  - Serialisierbarkeit als Kriterium
  - Zweiphasen-Sperrprotokolle garantieren Korrektheit
  - Striktes 2PL vermeidet kaskadische Abbrüche im Fehlerfall
  - Hierarchisches verhindert Phantomprobleme

Grundlagen der Datenbanken

Lektion 15

Wahrung von Sicherheit und Integrität

1. Sichten
   - Definition, Klassifizierung, Problembehandlung
2. Semantische Integritätssicherung
   - Klassifikation und Formulierung von Integritätsbedingungen
   - Kontrolle und Wiederherstellung von Integrität
3. Datensicherheit und Datenschutz
   - Zugangskontrolle: Identifikation und Authentifikation
   - Vergabe und Kontrolle von Zugriffsrechten
   - Inferenz- und Datenflußkontrolle

Sichten

- **Strukturierung und Präsentation von Daten**
  - Externe Ebene zur Erreichung logischer Datenunabhängigkeit
  - Abgeleitete (virtuelle) Datenbank – feste Berechnungsvorschrift
  - Ausblendung von Daten und Präsentation in neuer Form
- **Vorteile des Sichtkonzepts**
  - Vereinfachung von Anfragen, Übersichtlichkeit
  - Strukturierung der Datenbeschreibung,
    - zugeschnitten auf Benutzerklassen und bestimmte Anwendungen
  - Stabile Schnittstelle für Anwendungen (auch bei konzeptueller Reorganisation)
  - Beschränkung von Zugriffen $\mapsto$ Datenschutz
- **Probleme der Realisierung**
  - Automatische Anfragetransformationen in nicht-orthogonalen Sprachen (SQL)
  - Umsetzung von Änderungen auf Sichten in tatsächliche DB-Änderungen
- **Definition im Relationenmodell**
  - Name (und Schema) der virtuellen Relation, Berechnungsvorschrift
  - create view V [(A_1,...,A_n)] as (select ...from ...) [with check option]

Sichten: Klassifizierung und typische Probleme

- **Projektionssicht: Ausblendung von Attributen**

  `create view MA as select Mitarbeiter, Abteilung from MGA`

  – Einfügen: Werte für unsichtbares Attribut `Gehalt` fehlen
    · Nullwerte (Integritätsverletzung?) oder Defaultwerte einsetzen
  – Ändern: ein Sichttupel kann mehreren Basistupeln entsprechen

- **Selektionssichten: Ausblendung von Tupeln**

  `create view MG as select Mitarbeiter, Gehalt from MGA where Gehalt>2000`

  – Änderung von `Gehalt` kann Tupel unsichtbar machen (Tupelmigration)
  – Veränderung kann unsichtbares Tupel ebenfalls ändern

- **Verbundsicht: Kombination mehrerer Relationen**

  `MGAL := MGA(Mitarbeiter,Abteilung,Gehalt)` $\bowtie$ `AL(Abteilung,Leiter)`

  – Mehrdeutigkeiten – welche Tupel der Originalrelationen sind betroffen?
  – Einfügen: ist neues Tupel `(Abteilung,Leiter)` Duplikat?
  – Ändern: neues Tupel `(Abteilung,Leiter)` einfügen oder altes ändern?
  – Löschen: Tupel `(Abteilung,Leiter)` wirklich entfernen?

- **Aggregierungs- und berechnete Sichten**

  `create view AS(Abt,G-Sum) as select Abteilung,sum(Gehalt) from MGA group by Abteilun`

  – Löschen, Einfügen, Ändern i.a. nicht sinnvoll übersetzbar
    · Welche Auswirkung hat Änderung von `G-Sum`?

Sichten: Behandlung von Problembereichen

- **Integritätsverletzungen**
  - Einfügen von Nullwerten bei Projektionssichten verletzt Schemadefinition
  - Einfache Schemaverletzungen durch Angabe von Defaults vermeidbar
  - *SQL weist integritätsverletzende Änderungen generell ab*
- **Seiteneffekte im unsichtbaren Teil der Datenbank**
  - Tupelmigration bei Selektionssichten verletzt Datenschutz
  - *SQL überläßt explizite Behandlung dem Benutzer (*with check option*)*
- **Mehrere oder keine Transformationsmöglichkeiten**
  - Auswahlproblem: welche Umsetzung einer Sichtänderung wird gewählt?
  - Änderung eines Wertes in Aggregierungs- oder berechneten Sichten?
  - *SQL klassifizert derarige Sichten als nicht änderbar*
- **Keine 1:1 Beziehung Sichttupel $\leftrightarrow$ Basistupel**
  - Elementare Sichtänderungen betreffen viele Tupel der Basisrelationen
  - Projektionen mit Schlüsselteilen und distinct
  - *SQL verbietet* distinct *in Beschreibung änderbarer Sichten*

Sichtbehandlung in SQL

create view `AS(Abt,G-Sum)`
as select `Abteilung,`**sum**`(Gehalt)` **from** `MGA` **group by** `Abteilung`

- **Syntaktisches Mischen**
  - Sichtdefinition wird in Anfrage eingesetzt
    - Sichtattribute in **select**-Liste ggf. umbenannt
    - Originalrelationen im **from**-Teil
    - Konjunktive Verknüpfung der **where**-Klauseln
  - select `Abt` from `AS` where `Abt` like `A%`

  $\mapsto$ select `Abteilung` from `MGA` where `Abteilung` like `A%` group by `Abteilung`

- **Unerwartete Probleme durch Schachtelungsverbote**
  - select `Abt` from `AS` where `G-Sum` > `500`

  $\mapsto$ select `Abteilung` from `MGA` where sum`(Gehalt)` > `500` group by `Abteilung`

  Nicht möglich, da Verbot von Schachtelungen im where*-Teil*

  korrekt: select `Abteilung` from `MGA` group by `Abteilung` having sum`(Gehalt)` > `500`
  - select avg`(G-Sum)` from `AS` $\mapsto$ select avg(sum`(Gehalt)`) from `MGA` group by `Abteilung`

    Völlig unmöglich, da Verbot geschachtelter Aggregatfunktionen

Sichtbehandlung in SQL: Restriktionen

- **Einschränkungen an Beschreibung änderbarer Sichten**
  - Reine Selektion (kein Verbund, Schnitt, Vereinigung)
  - Kein **distinct**, keine Gruppierungen ($\mapsto$ 1:1 Bezug bleibt erhalten)
  - Keine Arithmetik und Aggregation im **select**-Teil
  - Maximal eine Referenz auf eine Relation im **from**-Teil
  - Relationennamen äußerer SWF-Blocks nicht in **from**-Teil von Unterabfragen
- **SQL differenziert nicht**
  - Änderungen und Löschungen gleichermaßen verboten
  - Auch in Fällen, in denen Löschen unproblematisch wäre

  $\Rightarrow$ keine gute Lösung für einen Standard

Sicherung semantischer Integrität

- **Integrität**
  - Inhaltliche Übereinstimmung zwischen Datenbank und Miniwelt
  - Semantischer Begriff – eigentlich nicht überprüfbar
  - Das schwierigste Problem der Datenbankforschung
- **Konsistenz**
  - Korrektheit Datenbank-interner Strukturen und Verwaltungsinformationen
  - DBMS kann nur Datenkonsistenz sichern
    - Physische Konsistenz von Geräten, Speicherstrukturen, Zugriffspfaden
    - Konsistenz beim Transaktionsablauf $\mapsto$ Concurrency Control/Recovery
- **Logische Konsistenz**
  - Modellinhärente Bedingungen
  - Syntaktische Simulation von Integrität durch benutzerdefinierte Bedingungen
  - Nicht wirklich identisch mit semantischer Integrität
    - Konsistente DB kann semantisch unsinnige Informationen enthalten
    - Dennoch 'Integritätsbedingungen' um Zusammenhang hervorzuheben
  - Ziel: hohe Datenqualität (Übereinstimmung Datenbank $\leftrightarrow$ Miniwelt)

Integritätsbedingungen: Beispiele

1. Das Konto von Schmidt darf nicht überzogen werden
   – Unmittelbare Bedingung für ein einzelnes Tupel einer Relation
2. Kein Konto darf unter -1000.- absinken
   – Lokale Bedingung für alle Tupel einer Relation
3. Kein Kundenname darf mehrfach vorkommmen
   – Bedingung für alle Paare von Tupeln einer Relation
4. Der Erbsenpreis muß im Schnitt unter dem von Spargel liegen
   – Arithmetische Bedingung für Paare von Teilmengen von Tupeln einer Relation
5. Zu jeder Ware muß ein Lieferant existieren, der sie liefert
   – Relationenübergreifende Bedingung (modelliert über Fremdschlüssel)
6. Der Brotpreis darf nicht erhöht werden
   – Bedingung über Zustandsübergänge
7. Kunden dürfen nur gelöscht werden, wenn sie keine Waren bestellen
   – Bedingung über Zustandsübergänge und Operationen
8. Der Mietpreis darf in 3 Jahren höchstens 20% steigen
   – Langfristig zu überprüfende Bedingung
9. Kunden, die keine Waren mehr bestellen, müssen gelöscht werden
   – Integritätsregel: auszulösende Aktion

Klassifikation von Integritätsbedingungen

- **Reichweite (Granularität)**
  - Bedingung an Einzelattribute eines Tupels
  - Satzausprägungsbedingung an einzelne Tupel (mehrere Attribute)
  - Satztypbedingungen an mehrere Tupel einer Relation (Paare, Mengen)
  - Satztypübergreifende Bedingung an mehrere Tupel verschiedener Relationen
- **Art der Überprüfbarkeit**
  - Statische Zustandsbedingungen
  - Transitionale Bedingungen an (unmittelbare) Zustandsübergänge
  - Temporale (langfristige) Bedingungen
- **Zeitpunkt der Überprüfbarkeit**
  - Unmittelbar: sofort nach einzelner Änderungsoperation
  - Verzögert: am Ende einer Transaktion (komplex, mehrere Objekte)
- **Art der Reaktion auf Verletzung**
  - Reject: Zurückweisung der gesamten Transaktion
  - Repair: Korrigierende Maßnahmen

SQL-Formulierung von Integritätsbedingungen

- **Integritätsregel (SQL 92)**
  - Integritäts-**B**edingung
  - Betroffene **O**bjekte
  - **A**uslöser für Überprüfung
  - **R**eaktion auf Verletzung

  create assertion `a` **check** `(P)` — **B,O**
  [**immediate**|**deferred**] [**on update**|**on delete**] — **A**
  [**cascade** | **set null** | **set default** | **no action**] — **R**
  - Spezifischere Reaktionsmöglichkeiten durch Verwendung von Triggern

- **Übergangsbedingungen nicht formulierbar**
  - Ursprünglicher Vorschlag:
    - old x für Wert von `x` vor, `x` (oder new x) für Wert nach Änderung

- **Trigger (SQL 3)** $\mapsto$ aktive Datenbanken

  create trigger `t` [**before**|**after**] [**insert**|**delete**|**update**] **of** $A_1, \ldots, A_n$ — **A**
  on `r` **referencing old** `old-r` **new** `new-r` — **O**
  when P (*update-Anweisungen*) — **B,R**
  - Automatisches Starten von Folgeänderungen (immediate)
  - Weitere Trigger können aktiviert werden

Formulierung von Integritätsbedingungen: Beispiele

1. Das Konto von Schmidt darf nicht überzogen werden
   - `...check not exists(select * from Kunde where Kname='Schmidt' and Kto<0)`
   - Besser als lokale Integritätsbedingung formulieren
2. Kein Konto darf unter -1000.- absinken
   - Lokale Integritätsbedingung: `...check Kto` $\geq$ `-1000`
3. Kein Kundenname darf mehrfach vorkommmen
   - Umständlich, besser lokale Schlüsselbedingung: `primary key (KName)`
4. Der Erbsenpreis muß im Schnitt unter dem von Spargel liegen
   - `...check (select avg(Preis) from Lieferant where Ware = 'Erbsen')`
     $\leq$ `(select avg(Preis) from Lieferant where Ware = 'Spargel') deferred`
5. Zu jeder Ware muß ein Lieferant existieren, der sie liefert
   - Fremdschlüsselbedingung: `foreign key (Ware) references Lieferant(Ware)`
6. Der Brotpreis darf nicht erhöht werden
   - Lokale Bedingung: `...check not (Ware='Brot' and Preis>old Preis) on update`
7. Kunden dürfen nur gelöscht werden, wenn sie keine Waren bestellen
   - `...check not exists(select * from Auftrag where Kname = old Kname) on delete`
8. Der Mietpreis darf in 3 Jahren höchstens 20% steigen
   - Nicht formulierbar
9. Kunden, die keine Waren mehr bestellen, müssen gelöscht werden
   - Trigger verwenden

Überwachung von Integritätsbedingungen

- **Code-Erweiterung durch Pre-Compiler**
  - Einfügen von Kontrollbefehlen, wenn Operation Bedingung berührt
  - Verlangt Namensbindung zur Übersetzungszeit
  - Nicht geeignet für Ad-hoc Anfragen und -Änderungen
- **Query-Modifikation bei Interpretierung**
  - Hinzufügen einer where-Bedingung zur Anfrage ($\mapsto$ immediate)
  - Effizient, aber nur für einfache Integritätsbedingungen
- **Separate Überprüfung durch DBMS**
  - Unmittelbar nach Änderung oder bei Transaktionsende
  - Rückgabe eines Fehlercodes
- **Kosten für Mehraufwand noch unklar**
  - Viele DBMS unterstützen nur einfachste Bedingungen

Schutz vor unberechtigtem Zugriff und Manipulation

- **Datenschutz (Gesetze)**
  - Festlegung, welche Daten in welchem Umfang schutzbedürftig sind
  - Vorschriften, die Mißbrauch entgegen wirken sollen
    - Erlaubte Speicherungen, Zugriffe und Weitergabe von Daten
  - Schutz für Belange der Betroffenen
    - Verbotsprinzip mit Erlaubnisvorbehalt für personenbezogene Daten
    - Rechte der Betroffenen: Auskunft, Berichtigung, Sperrung, Löschung
    - Besondere technische und organisatorische Maßnahmen
- **Datensicherheit**
  - Schutz vor Verlust oder Manipulation von Daten
    - Beabsichtigte Verletzungen $\mapsto$ Kontrollmaßnahmen
    - Unbeabsichtigte Verletzungen $\mapsto$ Recovery-Mechanismen
- **Technische und organisatorische Probleme**
  - Zugangskontrolle für Benutzer und Daten
  - Isolation der Benutzer und Betriebsmittel
  - Zugriffskontrolle auf gemeinsame Daten
  - Datenflußkontrolle beim Datentransport
  - Inferenzkontrolle bei statistischen Datenbanken

Zugangskontrolle

- **Identifikation der Benutzer**
  - Anmeldung des Benutzers unter einer dem System bekannten Kennung
- **Authentifikation der Benutzer**
  - Nachweis der angegebenen Identität durch persönliche Merkmale
  - Wissen: Paßwörter, PIN, ...
  - Gegenstände: Schlüssel, maschinenlesbare Ausweise, Chip-Karte, ...
  - Charakteristika: Fingerabdruck, Stimme, Unterschrift, ...
- **Verantwortlichkeit beim Betriebssystem**
  - Basisprüfung mit Vergabe grundsätzlicher Rechte
  - Zusätzliche Prüfung durch DBMS bei besonders geheimen Daten
  - Zusätzliche kryptographische Maßnahmen
    - Verschlüsselung von Daten, Nachrichten und Programmen
    - Verhindern unerlaubte Zugriffe durch Lücken im Betriebssystem

Zugriffskontrolle

- **Isolation von Daten**
  - Jeder besitzt alle Rechte auf eigenen Daten
  - Keine gemeinsamen Nutzung
  - Widerspricht Hauptziel großer Datenbanksysteme
  - Kontrollprobleme bei gemeinsamer Nutzung
    - Zugang durch Paßwort bietet nur Eingangskontrolle
    - Erlaubnis zu unspezifisch: alles oder nichts
- **Individuelle Autorisierung**
  - Zugriff auf DB-Dateien nur durch DBMS-Funktionen
  - Vergabe von Zugriffsrechten abhängig von Subjekt, Objekt und Operation
    - Subjekte: Benutzer, Terminals
    - Objekte: Relationen, Sichten, Anwendungs- und Dienstprogramme
    - Operationen: Lesen, Schreiben, Ändern, Ausführen, Weitergabe von Rechten
  - DBMS verwaltet wertunabhängige Zugriffsrechte in Berechtigungsmatrix
    - Problem: trojanische Pferde (Erschleichen von Nutzungsprivilegien)
  - Sichtkonzept erlaubt wertabhängige Zugriffsrechte

Vergabe von Zugriffsrechten

- **Vergabekonzepte**
  - Prinzip des kleinstmöglichen Privilegs
    - Nur, was unbedingt benötigt wird
  - Hierarchische Ordnung der Nutzungsprivilegien von Benutzern
  - Hierarchische Ordnung der notwendigen Rechte für Operationen
  - Hierarchische Ordnung der Schutzbedürftigkeit von Objekten
    - Kein Zugriff auf Objekte höheren Schutzgrades als Nutzerprivilegien
    - Kein Schreiben von Objekten niedrigeren Schutzgrades als Privilegien
- **Weitergabe von Rechten**
  - Nutzer kann Rechte individuell weitergeben (grant-Befehl)
    - Recht zur Weitergabe kann ebenfalls weitergereicht werden
  - Nutzer kann weitergegebene Rechte widerrufen (revoke-Befehl)
    - Zustand sollte so sein, als ob Recht nie vergeben worden wäre
    - Problem, wenn Rechte von mehreren Nutzern weitergegeben wurden
  - Autorisierungsgraph (mit Zeitablauf) erforderlich
    - Verwaltung des Widerrufs bei Graphen mit Zeitablauf aufwendig

Datenfluss- und Inferenzkontrolle

- **Datenflußkontrolle**
  - Berechtigte Nutzer könnten geheime Daten durch Kopie zugänglich machen
    - unberechtigte Nutzer bekommen Zugriff auf geheime Daten
  - Kontrolliere Verbleib und Verwendung von Daten nach Zugriffen
  - Beschreibe vorgesehene Transport- und Verarbeitungswege
- **Inferenzkontrolle in statistischen Datenbanken**
  - Datenbank erlaubt statistische Zugriffe auf geschützte Daten
    - Einzeldaten sind geschützt, Statistiken erlaubt
    - Volkszählung, medizinische Statistiken, Steuerschätzung, ...
  - Durch geschickte Abfragen könnte man individuelle Informationen erschließen
    - Eingrenzung auf die eine Person die ich kenne

    Wieviele Ingeneure zwischen 35 und 40 mit mehr als 2 Kindern sind rauschgiftsüchtig?
  - Abhilfemöglichkeiten
    - Keine Ausgabe von weniger als x Werten (reicht nicht: stelle x Anfragen)
    - Überprüfung, ob Anfragen aufeinander aufbauen (kaum durchführbar)
    - Gezielte Einstreuung kleiner statistischer Ungenauigkeiten

Resümee: Sicherheit und Konsistenz

- **Logische Konsistenz (Semantische Datenqualität)**
  - Syntaktische Mechanismen zur Erhaltung semantischer Integrität
  - Schutz vor (absichtlichen) Verletzungen durch Benutzer
  - Überprüfung und Wiederherstellung der Bedingungen der Miniwelt
- **Transaktionskonsistenz**
  - Vermeidung unbeabsichtigter Nebenwirkungen des Mehrbenutzerbetriebs
  - Concurrency Control zur Erhaltung der Ablaufintegrität
- **Datensicherheit und Datenschutz**
  - Schutz vor (beabsichtigter) Manipulation oder Zerstörung von Daten
  - Schutz vor unerlaubtem Zugriff auf geschützte Daten
  - Zugriffskontrolle durch Authentifikation und Autorisierung
- **Speicherkonsistenz**
  - Schutz vor unvorhersehbarem Verlust oder Verfälschung von Daten
  - Recoverymechanismen zur Wiederherstellung der Vollständigkeit

Grundlagen der Datenbanken

Lektion 16

Aktuelle Entwicklungen

1. Aktive Datenbanken
2. Weitergehende Ansätze
   - Heterogene Datenbanken
   - Deduktive Datenbanken

Aktive Datenbanken

- **Passive Überprüfung von Integrität**
  - Liefert nur Fehlermeldung bei Verletzung
  - Verlangt Wiederherstellung von Integrität durch Anwendungsprogramm
    - insbesondere Redundanz-Nachführung bei abgeleiteten Daten

  $\Rightarrow$ Ineffizient und fehleranfällig

- **Deskriptive Beschreibung von Sachverhalten**
  - Explizite Regeln für Aufrechterhaltung von Datenqualität
  - Anwendungsunabhängige Spezifikation und Handhabung von Aktionen
    - zentrale (redundanzfreie) Verwaltung von Daten und Regeln
    - verbindlich für alle Benutzer
    - vereinfacht Wartung und Anwendungsentwicklung
  - Problem: Prüfung von Widerspruchsfreiheit und Vollständigkeit
    - Aktionen sind Prozeduren mit operationaler Semantik

- **Aktives Verhalten sinnvoll zur**
  - Integritätserhaltung
  - Automatische Wartung abgeleiteter Daten
  - Allgemeine Überwachungs- und Kontrollaufgraben (Trigger, Alerter)

Aktives Verhalten: Erhaltung der Datenqualität

- **Integritätserhaltung**
  - Typbedingungen, Schlüsselbedingungen, Referentielle Integrität
    - Leicht überwachbar, Defaultverhalten zur Erhaltung definierbar
  - Wertebereichseinschränkungen, Aggregatbedingungen, allgemeine Bedingungen
    - `17≤Student.Alter≤80`, 'maximal 10 Personen pro Abteilung'
    - 'Vier Vordiplomprüfungen besser als 4.0 $\Rightarrow$ Vordiplom bestanden'
    - keine universelle Methode zur Integritätserhaltung möglich
- **Abgeleitete Daten**
  - Müssen bei Änderung von Basisdaten automatisch nachgeführt werden
  - Standardtechnik für virtuelle Daten in vielen DBMS
    - Sichten: Berechnung auf Anforderung
    - bei häufigen Updates der Basisrelation, wenig Sichtanfragen
  - Konsistenzerhaltung schwierig bei materialisierten Daten
    - abgeleitete Daten als spezielle Relation abgespeichert
    - bei häufiger Nutzung abgeleiteter Daten, seltenen Basisupdates

Aktives Verhalten: Trigger, Alerter

- **Regeln für komplexere Zusammenhänge**
  - Erfassung erwünschter Reaktionen auf Situationen und Ereignisse
    - Wird Gehalt um mehr als 10% erhöht, benachrichtige Manager
    - Verliert Abteilung mehr als 5 Angestellte, so kürze Budget um 25%
  - Zusammenhang nicht durch statisches Prädikat beschreibbar
    - mehr als nur Qualität der Daten
  - Realisierung sollte durch einheitlichen DBMS-Mechanismus geschehen
    - Auftreten der Ereignisse erkennen und Reaktion ausführen
    - Realisierung durch einzelnes AP wäre schlechtes Software-Engineering
- **Trigger**
  - Auslöser (Ereignis): ausgeführte Datenbankoperation, BOT, EOT
  - Reaktion: Folge von Datenbankoperationen
    - Ziel meist Wiederherstellung von Integrität
  - Problem: Priorität (mehrere Trigger), Terminierung (gekoppelte Trigger)
- **Alerter**
  - Auslöser: beliebige Signale (Zeit, Anwenderprogramm, Dialog,...)
  - Anwendung: automatische Nachbestellung, Benachrichtigung, Dialog
    - Entwicklung in Richtung auf offene Datenbanken
  - Problem: Semantik der Reaktion bei Auslösung durch Anwenderprogramm

ECA-Regeln als allgemeines Konzept

- **Allgemeine Struktur von Regeln**
  - **E**vent: Auslöser für Regel
  - **C**ondition: zusätzliche Bedingung an Regelausführung
  - **A**ction: auszuführende Operationen
- **Angabe von Events** (SQL-3 Notation)
  - Bezug auf Operation: [before | after | instead of]
  - Bezug auf Relation: [insert | update | delete | read] of (A_1..A_n) on r
  - Bezug auf Transaktion: on [bot | commit | abort]
  - Zeitgesteuert: [at t | during t_1–t_2 | repeat each t]
  - Benutzerdefiniert: on event *name* (*parameter*)
- **Angabe von Conditions** (optional)
  - Boolescher Ausdruck über allen Daten der Datenbank ($\hat{=}$ select)
  - Überprüfungszeitpunkt: [immediate [not deferrable] | deferred]
  - Bezug auf Event: [coupled [not decoupable] | decoupled]
- **Angabe von Actions**
  - Granularität: [for each [statement | row]]
    - Ausführung für einzelnes Tupel oder alle betroffenen Tupel auf einmal
  - DML-Anweisungen (ggf. auch DDL=Befehle und externe Funktionsaufrufe)
  - Durchführungszeitpunkt: [immediate [not deferrable] | deferred]
  - Koppelung: [coupled [not decoupable] | [dependent|independent] decoupled]

Bezug zwischen Event – Condition – Action

- **Trennung Event / Condition wichtig**
  - Event: wann soll überprüft werden
  - Condition: was soll überprüft werden
  - Erlaubt Aktionsauslösung aufgrund von Signalen und Operationen
  - Unterstützt asymmetrische Regeln (z.B. für Sicherung einer Invariante `A=B`)
  - Ermöglicht Optimierung: Überprüfung nur zu speziellen Events
  - Ermöglicht flexible Ausführung
    - Auswertung der Condition zu späterem Zeitpunkt (deferred)
    - Auswertung der Condition in späterer Transaktion (decoupled)
- **Zeitlicher Bezug: immediate / deferred**
  - Sofort oder am Ende der auslösenden Transaktion
  - Bezug Event – Condition / Condition – Action
- **Verarbeitungskontext coupled / decoupled**
  - In derselben oder einer separaten Transaktion
  - Bezug Event – Condition / Condition – Action
- **Übergabe von Parametern**
  - vom Event an Condition und Action
- **Übergabe des Überprüfungsergebnisses**
  - von Condition an Action

Spezifikation von Ereignissen

- **Ereignisse sind grundsätzlich Zeitpunkte**
  - Eintreten muß vom DBMS entdeckt und signalisiert werden
  - Zeitspanne zwischen Eintreten und Entdeckung muß klein sein
    - Echtzeitanforderung unverträglich mit Rücksetzbarkeit von Transaktionen
  - Ereignis kann in mehreren Regeldefinitionen vorkommen
    - Separate Definition von Ereignis und Identifikation sinnvoll
- **Ereignisklassen**
  - Regeln spezifizieren Klassen gleichartiger Ereignisinstanzen
  - Parametrisierung für Weitergabe von Informationen an Bedingung/Aktion
    - wichtig bei Modellierung zusammengesetzter Ereignisse
- **Primitive Ereignisse**
  - Zeitereignis, Methodenereignis, Wertereignis, Transaktionsereignis
  - Abstraktes Benutzerereignis: muß deklariert und explizit erzeugt werden
- **Zusammengesetzte Ereignisse** — Ereignisalgebra
  - 3 Verknüpfungsoperatoren
    - Disjunktion `(E1 E2)`, Konjunktion `(E1,E2)`, Sequenz `(E1;E2)`
  - 3 Überwachungsoperatoren für Ereignisintervalle `[E1-E2]`
    - Negation `not E [E1-E2]`: `E` tritt zwischen E_1 und E_2 nicht auf
    - Wiederholung `*E [E1-E2]`: nur das erste Auftreten wird signalisiert
    - Zählung `times(`n`,E) [E1-E2]`: das n-te Auftreten wird signalisiert

Spezifikation von Aktionen

- **Granularität**
  - Triggerauslösung für jedes einzelne (geänderte) Tupel
  - Geänderte Tupelmenge (bzw. Transaktion) löst einen Trigger aus
  - Unterschiedliche Effekte bei kaskadischer Triggerauslösung
- **Verhalten bei Anwendbarkeit mehrerer Regeln**
  - Bestimmung einer Reihenfolge bei sequentieller Ausführung
  - Korrektheitskriterien für parallele Ausführung
- **Gegenseitige Auslösbarkeit – Selbstreferenz**
- **Bezug zu auslösender Transaktion**
  - Zeitlicher Bezug: immediate / deferred
  - Verarbeitungskontext: coupled / decoupled
  - Äbhangigkeit von erfolgreicher Beendigung: dependent / independent
    - independent: kein Zurücksetzen bei Abbruch der auslösenden Transaktion

Implementierungsaspekte

- **Zusatzschicht oder ins DBMS integriert ?**
- **Effizienzfragen**
  - Erkennen von Events, Auswahl anwendbarer Regeln
  - Auswertung von Bedingungen, Ausfrührung von Aktionen
- **Integration des Regelkonzepts mit DBMS-Konzepten**
  - Recovery und Fehlerbehandlung
  - Synchronisation von Mehrbenutzerbetrieb, Autorisierung
- **Kontrolle der Regelausführung**
  - Ausführung ist prozedural und unstrukturiert
  - Semantik und Interaktion mit anderen DB-Operationen
  - Sicherung von Terminierung
  - Begrenzung von nichtdeterministischem Verhalten (Konfluenz)
  - Tracing und Debugging
- **Systeme**
  - Starburst: relationaler Prototyp (IBM)
  - HiPAC: objektorientiertes System (XEROX)
  - Eingeschränkte Regelsysteme in Ingres 6.0, Sybase

Klassifikation aktiver Systeme

- **update → update**
  - Aktualisierung löst weitere Aktualisierung aus
  - Kontrollfluß für Terminierung des forward chaining nötig
- **update → retrieve**
  - Regel wirkt als Alerter (Anzeige aufgesuchter Information)
  - Nachricht, wenn etwas 'Interessantes' passiert
- **retrieve → retrieve**
  - Oft Ersatzanfrage für tatsächliche Anfrage

    $\hat{=}$ Bereitstellung virtueller Daten oder Sichten
  - Kontrollfluß für Terminierung des backward chaining nötig
- **retrieve → update**
  - Überwachung von Anfragen durch Audit-file
  - Nur in wenigen Systemen angeboten

Anwendungsgebiete für ECA-Regeln

- **Erhaltung allgemeiner Integritätsbedingungen**
  - Nachträgliche Überprüfung und `ROLLBACK` von Änderungsbefehlen
  - Automatisches Update bei Umbenennung von Fremdschlüsseln
- **Konsistenz materialisierter abgeleiteter Daten**
  - Automatisches Update abgeleiteter Daten nach Änderungsbefehlen
  - Selbstauslösende Regeln für rekursiv abgeleitete Daten
- **Trigger / Alerter**
  - Überprüfung der Trigger-Bedingung nach Änderungsbefehlen
  - Warten auf Alerter-Event
  - Auslösen der gewünschten Aktion
- **Sonstige Anwendungen**
  - Autorisierung, Verwaltung von Abhängigkeiten
  - Kooperationsunterstützung (benachrichtigung)
  - Leistungsüberwachung (Automatische Statistiken, Lastbalancierung)
  - Fabrikautomatisierung
- **Probleme**
  - Operationale Semantik mit kaum vorhersagbaren Ergebnissen

  $\Rightarrow$ Entwicklung einer korrekten Menge von Regeln schwierig

Verteilte und heterogene Datenbanken

- **Ablösung der monolithischen zentralen Datenbank**
  - Verteilte Datenbank senkt Kommunikationsaufwand und Verwundbarkeit
  - Höhere Verarbeitungskapazität + bessere Konsistenz zwischen Datenbanken
- **Client-Server Architektur**
  - Zentrale Datenhaltung, verteilte DBMS-Funktionalität
  - Clients fordern Datenbestände beim Server an und verarbeiten sie
  - Server kann vereinfachte DBMS-Struktur besitzen (geringere Belastung)
- **Verteilte homogene Datenbanken**
  - Verteilter Datenbestand mit einheitlicher Datenschnittstelle
    - 'Zentrale' Regelung der Datenpartitionierung auf Knotenrechner
    - Kontrollierte Redundanz zur Steigerung von Effizienz und Verfügbarkeit
  - Optimierung von Anfragen noch möglich?
- **Heterogene verteilte Datenbanken**
  - Verteilter Datenbestand mit verschiedenen Schnittstellen
    - Koppelung unterschiedlicher, historisch gewachsener DBS möglich
  - Realisierung einer einheitlichen Zugriffsmethodik auf Datenbestände
    - Korrekte Auflösung von Konflikten in der Datenbeschreibung?
- **Föderative Datenbanken** $\hat{=}$ WWW-Datenbank
  - Autonome Datenbanken ohne zentrale Kontrolle
  - Unterstützung globaler Anwendungen und Exportschemata

Deduktive Datenbanken

- **Regelbasierte Datenbanken** $\mapsto$ Expertensysteme
  - Gespeicherte Daten entsprechen Fakten und wenn-dann-Beziehungen
    - festes Verarbeitungsmodell der Prädikatenlogik
  - Anfrage auf Gültigkeit einer vorgegebenen Formel
  - Bottom-Up Auswertung: Transitive Hülle der Fakten unter Regeln
  - Effizient bei starkem Übergewicht der Fakten über Regeln
  - Behandlung von Negation schwierig
- **Wissensbanken**
  - Ziel: explizite Verwaltung von Wissen in persistentem Datenmodell
  - Unterstützen zusätzlich verschiedene Verarbeitungsstrategien
    - unscharfes Schließen (Fuzzy-Datenbanken)
    - Default Schließen
  - Verlangt zusätzlich Speicherung von Verarbeitungsinformationen

Grundlagen der Datenbanken

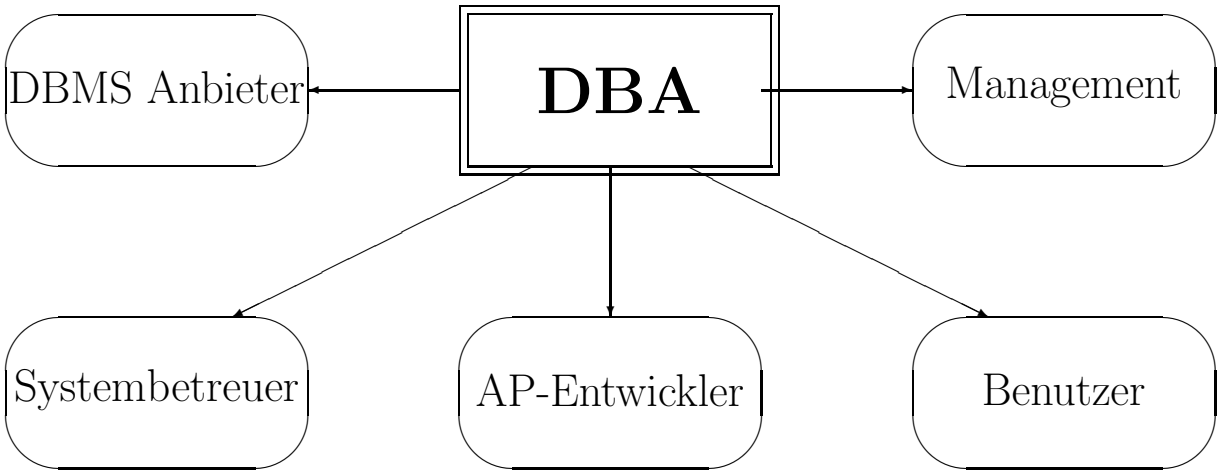
Lektion 17

Datenbankadministration

1. Der DBA als Vermittler zwischen Interessengruppen
   – Management, Benutzer, Entwickler, Systembetreuer, Anbieter
2. Schritte beim Übergang auf Datenbanksysteme
   – Analyse, Ziele, Alternativenbewertung, Kosten, Einführung
3. Resümee und Zukunftstendenzen

Aufgabenfelder eines Datenbankadministrators

- **DBA $\hat{=}$ Verwalter der Datenressourcen**
  - Daten sind eine der wichtigsten Ressourcen eines Unternehmens
  - DBA kontrolliert Datenbestand und seine Qualität
    - Niemand außer dem DBA fühlt sich hierfür zuständig
  - DBA ist Vermittler zwischen Parteien mit widersprüchlichen Interessen



- **DBA Verantwortung ist abteilungsübergreifend**
  - DBA-Funktion muß hoch in Unternehmenshierarchie sein
    - Verhandlung mit Direktionsebene muß möglich sein
  - DBA muß guter Diplomat und guter Manager sein
  - DBA muß Unternehmen gut kennen und technische Kenntnisse besitzen
  - DBA-Funktion oft von Gruppe von Personen ausgeübt
  - DBA-Funktion gut durch Informationsfluß charakterisierbar

Interaktion DBA ⟷ Management

- **Management → Datenbankadministrator**
  - Prioritäten des Unternehmens
  - Management Vorstellungen / Zeitplan für DB-Entwicklung
  - Budget
  - Zusagen an Dritte (Datenbereitstellung, Performance, ...)
  - Strategische Ziele und Pläne (Umstrukturierung, Wachstumserwartung, ...)

  Ist DBA nicht hoch genug, wird vertrauliche Information nicht weitergegeben
- **Datenbankadministrator → Management**
  - DBA sammelt Projektinformationen und leitet diese weiter
  - DBA muß DB-Bedeutung und -Problematik verständlich machen
    - Zeitpläne, Personal
    - Budget mit Details von Hardware/Software und Manpower
    - Projektanalysen (Einflüsse auf DB)
    - Datenschutz und Datensicherungsmaßnahmen
    - Raumanforderungen (z.B. für Hardware)

Interaktion DBA ⟷ Benutzer

- **DBA ist wichtigster Ansprechpartner**
  - DBA muß Benutzer Vertrauen einflößen
    - Datenschutz, Korrektheit und Sicherheit der Daten
- **Benutzer → Datenbankadministrator**
  - Datenanforderungen, Datenbeschreibung, Integritätsbedingungen
  - Datennutzung und Performance-Anforderungen
  - Archivierungsanforderunge
  - Prioritäten der Anwendungsprogramme in DB-Umgebung
  - Verknüpfung von Anwendungsprogrammen
  - Zuständigkeiten
- **Datenbankadministrator → Benutzer**
  - Vorschriften/Richtlinien zur Datenhaltung in DBS
  - Kontrollmechanismen für Datenänderung
  - Datensicherung, Datenschutzmaßnahmen, DB-Status
  - Abweichungen von Spezifikationen
  - Prozeduren, Vorschriften und Warnungen zum Löschen der DB
  - Phase-In Pläne für neue Datenbanken und Tools
  - Training

Interaktion DBA ⟷ Anwendungsentwickler

- **DBA konsolidiert Entwicklungsinformationen**
  - für alle Anwenderprogramme, auf die Datenbanken Auswirkung haben
- **Anwendungsentwickler → Datenbankadministrator**
  - Zeitpläne für AP-Entwicklung (besonders der DB-relevante Teil)
  - DB-Anforderungen des Anwenderprogramms
  - Update-Prozeduren
  - Datenvolumen, Speicheranforderungen, Performanz-Anforderungen
  - Testpläne
  - Test-Datenbank: Aufbau, Wartung, Verifikation
- **Datenbankadministrator → Anwendungsentwickler**
  - Wichtige relevante Informationen über Datenbank
  - Datensicherung, Datenschutzmaßnahmen
  - Schnittstelleninformation, Integritätsregeln
  - Details der Datenbankwartung

Interaktion DBA ⟷ Systembetreuer

- **DBA koordiniert Anwender und Systembetreuer**
  - Systembetreuer verantwortlich für physische Plattform
  - Systembetreuer brauchen nformation über Anforderungen von DBMS und AP
- **Systembetreuer → Datenbankadministrator**
  - Kompatibilitätsanalysen – Inkompatibiltätsmeldungen
  - Technische Lösungen und Alternativen zu DBA-Anforderungen
  - Kapazitätserweiterungen
  - Wartungspläne, Datensicherungspläne
  - Performanzmessungen, Fehlermeldungen
- **Datenbankadministrator → Systembetreuer**
  - Software- und Hardware-Installationsanforderungen und -Zeitpläne
  - Datenschutz- und Datensicherungsmaßnahmen
    - Checkpointfrquenz, Backup-Pläne, Archivkopien, Off-Site Lagerung
  - DB-Verfügbarkeit
  - Prioritäten der Anwendungsprogramme und Benutzer

Interaktion DBA ⟷ DBMS-Anbieter

- **DBA verantwortlich für Kontakte**
  - insbesondere DBMS-bedingte Hardware-Spezifikationen
- **DBMS-Anbieter → Datenbankadministrator**
  - Dokumentation, Kontakte zu anderen Lizenznehmern, Training
  - Hardware Anforderungen
  - Tuning Information, Wachtumsgrenzen, Kompatibilitätsinformationen
  - Tools
  - Upgrades, Bug-Fixes
- **Datenbankadministrator → DBMS-Anbieter**
  - Trainingerfordernisse
  - Datenschutz- und Datensicherungsanforderungen
  - Schnittstellenanforderungen
  - Performanz-Engpässe

Umstellung auf Datenbanksysteme

- **Globales System für Gesamtunternehmen**
  + Gute Integration, globale Planung möglich
  - Konfliktträchtig, langwierige Vorbereitung
  $\Rightarrow$ nur bei kleinen Unternehmen praktikabel
- **Unabhängige Teilsysteme**
  + Schnelle Implementierung, Systeme lokal optimal konfigurierbar
  - Gefahr von Inkompatibilität und Redundanz und Daten und Funktionen
  Tendenz ist "Departmental Computing"
- **Erweiterbares Teilsystem**
  – globale Planung, aber schrittweise Implementierung
- **Schritte bei Neubeschaffung und Konversion ähnlich**
  – Voranalyse
  – Zielsetzung und Bedürfnisse
  – Spezifikation von Alternativen / DBMS Selektion
  – Bewertung von Alternativen
  – Systementwurf und -implementierung
- **Häufigster und schwerwiegendster Fehler**
  – Schwache Zielsetzung, Bewertung von Alternativen vor Abschluß der Spezifikation

Anschaffung von DB-Systemen: Analyse I

- **Voranalyse: Durchführbarkeit**
  - Unterstützung auf hoher Management-Ebene
  - Unzufriedenheit mit gegenwärtigem System
  - Qualifikation des EDV-Personals
    - Übergangs-DB-Team bilden, DBA bestimmen
    - Kompetenzen abgrenzen
- **Zielsetzung und Bedürfnisanalyse**
  - Spezifische Ziele setzen
  - Benutzeranforderungen identifizieren
    - Analyse bestehender Reports und Formulare
    - Analyse von Wartungsaktivität, Benutzertätigkeiten und -wünschen
  - Benutzergespräche erst nach Auswertung ansetzen
    - Verfrühte Gespräche erzeugen nur Frust (keiner weiß Bescheid)
  - Anforderungen trennen in notwendig und erwünscht
  - In Datenwörterbuch einführen und auswerten
  - Kosten spielen in dieser Phase keine Rolle!

Anschaffung von DB-Systemen: Analyse II

- **Spezifikation von Alternativen / DBMS Selektion**
  - Identifizierung von Kandidatensystemen
  - Fixiere Grobarchitektur des Systems (Monolith $\leftrightarrow$ Client-Server?)
  - Fixiere Datenmodell (Relational, Objektorientiert, ...)
  - Fixiere Datenzugriff (Mengenorientiert $\leftrightarrow$ Navigierend)
  - Schnittstellen (Query-Sprache, Programmiersprachenschnittstellen, ...)
  - Benötigte Hard- und Software (Netz,...),
  - Kompatibiltät zur Hard-/Software des gegenwärtigen Systems
  - DBMS-Anbieter und lokale Vertreter
    - Seriösität, technische Kapazität des Personals
    - Dokumentation, Wartungspläne, Support

Anschaffung von DB-Systemen: Alternativenbewertung

- **Eliminierung nicht akzeptabler Produkte**
- **Formulierung von Selektionskriterien**
  - Kriterien, Gewichtung, K.O.-Kriterien
- **Kriterium: Einhaltung der Spezifikation**
  - Performanz bei Retrieval und Update, On-Line Performanz
  - Datenschutz, Datensicherheit, Integritätssicherung & Konsistenzgrade
  - Recovery Mechanismen
- **Kriterium: Benutzerfreundlichkeit**
  - Installieren, Modellieren, Benutzen
  - Entwicklung neuer AP, Erweiterbarkeit des Schemas
  - Dateneingabe und -konversion
- **Kriterium: Software und Tools**
  - DB-Design Tools, Wartungstools, Datenwörterbuch, Verfügbare Schnittstellen
  - Report-Generator, 4GL-Tools, Formbasierte Tools, Query-Sprachen
  - Benötigte Sprachen und Compiler

Anschaffung von DB-Systemen: Alternativenbewertung II

- **Kriterium: Support und Training**
  - Dokumentation, Technische Beratung, Hotline
  - Training: Niveau für DB-Personal und Benutzer
- **Bedeutung von Benchmarks**
  - Umstrittener Wert – repräsentativer durch Benchmarkstandards
  - Nötig, wenn Performanz kritisch
  - Formulierung guter Benchmarkprobleme schwer
  - Benchmarktests teuer
- **Absprache mit Benutzern**
  - Nur Benutzer von Systemen, welche die Kriterien überlebt haben
  - Vollständige Liste anfordern und ggf. ergänzen
  - Benutzer mit ähnlichen Anforderungen aussuchen
  - Benutzer sollten mindestens 1 Jahr Erfahrungen haben

Anschaffung von DB-Systemen: Kostenanalyse

- **Feste Kosten**
  - Durchführbarkeitsstudie
  - DBMS, Tools, Hardware, Raumausstattung, Aufbaukosten
  - Einführung des DBMS, Datenbankentwurf, Testen, Dokumentation
  - Datenkonversion, Programmkonversion, Training
  - Arbeitsausfall
- **Variable Kosten**
  - Personal, Rechnerzeiten
  - Wartung: Hardware, Software, Daten
  - Back-Up, Checkpoints, Recovery
  - Benutzerberatung und Nachschulung
- **Probleme**
  - Preisverfall in der Computerindustrie
  - Schätzung der Benefits

$\mapsto$ Nutzenanalyse

Anschaffung von DB-Systemen: Nutzenanalyse

- **Benefits**
  - Personaleinsparung (weniger Benutzer durch höhere Produktivität)
  - Einsparungen bei Daten- und Programmkonversion,
  - Einsparungen bei Datenwartung, Entwicklung neuer Anwenderprogramme
  - Weniger Programmierung durch Ad-hoc queries
  - Einsparung und Fehlervermeidung durch Datenkonsistenz
  - Neue Informationen durch Querverbindungen
- **Empfehlung**
  - Tendenzen beim Schätzen vermeiden

    $\cdots$ Tendenzfreie Fehler gleichen sich besser aus
  - Zahlungszeitpunkte sind wichtig (inkrementelles Wachstum bevorzugen)
  - Benutzer sollten Benefits ihrer Wünsche quantifizieren

Anschaffung von DB-Systemen: Systemeinführung

- **Kritische Anwendungen identifizieren**
  - Kritische Anwendungen parallel fahren ($\mapsto$ echte Belastung)
  - Faktor 2-2.5 Sicherheitsmarge kalkulieren
- **Anwendungsprogramme**
  - Erfolg beim ersten Anwendungsprogramm ist wichtig
  - Zuerst Anwendungen 'einfacher' Benutzer
  - Keine Erfahrungen an sichtbaren AP's (Payroll) sammeln
- **Benutzer beim Entwurf mit einbeziehen**
  - Pilot-System benutzen lassen
  - Anerkennung zukommen lassen !!

Datenbanksysteme: Resümee und Zukunftstendenzen

- **Relationenmodell für Standardanwendungen**
  - Netzwerkmodell wird sich auslaufen
  - Kommerzielle Entwicklungen meist 6-8 Jahre hinter dem Standard
- **Architekturen**
  - Client-Server Systeme
  - Homogene und heterogene verteilte Systeme
- **Konsistenzkriterien**
  - Globale (statt lokale) Kriterien werden wichtig
  - Sofortige (statt differierte) Konsistenz

  $\Rightarrow$ neue Transaktionsmodelle, aktive Datenbanken
- **Tendenz zu OO-DBMS in der Zukunft**
  - Zunehmende Tendenz zu Nichtstandardanwendungen
  - Verteilte aktive OO mit Zugriff auf herkömmliche Systeme
- **Weiterführungsmöglichkeiten**
  - Praktikum, Wintersemester 1996/97
  - Lehrbücher: Elmasri/Navathe, Datenbank-Handbuch, Date, ...