Vorlesung Datenbanken

Hinweis

Aufgrund der verfügbaren Menge guter einführender (auch deutschsprachiger) Datenbankliteratur verzichtet das vorliegende Scriptum darauf den in der Literatur verfügbaren Stoff nochmals aufzubereiten, sondern versammelt die Definitionen, Beispiele und Anmerkungen der Vorlesungen in einer übersichtlichen Zusammenstellung.
Für die vertiefende ausführliche lehrbuchartige Darstellung des Stoffes sei auf die empfohlene Literatur verwiesen.

1 Motivation und Einführung

1.1 Begriffsbestimmung: Was ist eine Datenbank?

Motivation für die Einführung einer Datenbank anstatt selbsterstellter Verwaltungs- und Zugriffsroutinen:

- Daten-Programm-Unabhängigkeit.
  Die verwalteten Daten sollen unabhängig vom sie verarbeiteten Programm gespeichert und zugreifbar sein.
  Dies wäre zwar in einem ersten Schritt auch durch die Verwendung des Dateisystems möglich, allerdings würde hierfür ein Programm zur Abbildung der Programmdaten auf die Dateistrukturen benötigt, welches selbst wieder eine Abhängigkeitsbeziehung zwischen Daten und Programm --- nun eben dem Abbildungsprogramm --- darstellen würde.
- Flexible Speicherung.
  Datenbankmanagementsysteme speichern die verwalteten Daten deutlich flexibler als selbsterstellte Routinen und sind somit hinsichtlich der Zukunftsfähigkeit effizienter.
- Verwaltungsfunktionen.
  Datenbankmanagementsysteme bieten in der Regel eine Reihe über die reine Datenverwaltung hinausgehende Funktionen wie Backup-Recovery, Integritätssicherung, Synchronisation gleichzeitiger Zugriffe oder Transaktionskontrolle an, die nicht selbständig implementiert werden

müssen.

Grundlegende Begriffe

Definition 1: *Daten*
Daten sind durch die Maschine verarbeitbare Einheiten.

Definition 2: *Information*
Daten die Bedeutung für den Empfänger besitzen.
Nach Shannon ermißt sich der Wert einer Information durch den Zuwachs der durch den Adressaten nach Kenntnis der Information beantwortbaren Ja/Nein-Fragen.

Mehr zum Unterschied zwischen *Daten* und *Information*:

- Homepage des Arbeitskreises *Bildung*, einem Zusammenschluß von Stipendiaten der Friedrich Naumann Stiftung
- Glossar der deutschsprachigen Anleitung zu PGP

Definition 3: *Datenbank*
Eine Datenbank (engl. *data base*) ist ein integrierter, persistenter Datenbestand einschließlich aller relevanten Informationen über die dargestellten Information (sog. Metainformation, d.h. Integritätsbedingungen und Regeln), der einer Gruppe von Benutzern in nur einem Exemplar zur Verfügung steht und durch ein DBMS verwaltetet wird.

Definition 4: *Datenbankmanagementsystem (DBMS)*
Ein Datenbankmanagementsystem (DBMS) ist die Gesamtheit aller Programme zur Erzeugung, Verwaltung und Manipulation einer Datenbank.
Im Deutschen wird auch der Begriff *Datenbankverwaltungssystem* (DBVS) synonym verwendet.

Beispiele verfügbarer DBMS:

- Die DBMS-Produkte des Herstellers Sybase
- IDMS von Computer Associates
- IMS, DB2 und Informix von IBM
- MySQL des gleichnamigen Herstellers
- Oracle 9i des Herstellers Oracle
- SQLServer und Access des Herstellers Microsoft

Beispiel 1: Am Markt verfügbare DBM-Systeme

Definition 5: *Relationales DBMS*
Ein relationales Datenbankmanagementsystem (RDBMS) ist ein DBMS, welches intern gemäß dem relationalen Modell organisiert ist.

Bei den genannten DBMS *MySQL*, *SQLServer*, *Access*, *DB2* und *Oracle* handelt es sich um relationale Systeme, bzw. Weiterentwicklungen davon.

Beispiel 2: Am Markt verfügbare RDBM-Systeme

Definition 6: *Relation*
Eine Relation $R(A_1, A_2 \ldots A_n)$ ist eine benannte Menge von *n*-Tupeln, wobei ein *n*-Tupel eine Anordnung von *n* atomaren, d.h. einfachen (nicht weiter zerlegbaren) Attributen $A_1, A_2 \ldots A_n$ ist.

Die Relation *Person* mit den Attributen *Vorname*, *Nachname* und *Geburtsdatum*.
Werteausprägungen davon:
$Person_1$("Meier", "Schorsch", "1955-10-01")
$Person_2$("Huber", "Franz", "1945-08-03")
...

Die Relation *Student* mit den Attributen *Name*, *Matrikelnummer*, *Semester* und *regelmäßigerMensabesucher*.
Werteausprägungen davon:
$Student_1$("Meier Schorsch", "08154711", "WIB 1", "true")
$Student_2$("Müller Xaver", "73619452", "BCM 4", "false")

...

Beispiel 3: Relationen

Definition 7: *Tabelle*
Eine Tabelle unterscheidet sich von einer Relation darin, daß ein Tupel mehrfach auftreten darf; eine Tabelle ist mathematisch keine Menge.

Die Tabelle *Student* mit den Attributen *Name*, *Matrikelnummer*, *Semester* und *regelmäßigerMensabesucher*.
Werteausprägungen davon:
$Student_1$("Meier Schorsch", "08154711", "WIB 1", "true")
$Student_2$("Müller Xaver", "73619452", "BCM 4", "false")
$Student_3$("Müller Xaver", "73619452", "BCM 4", "false")
...

Man beachte, daß der dritte Eintrag doppelt vorkommt, d.h. in all seinen Wertbelegungen mit dem zweiten übereinstimmt.

Beispiel 4: Tabelle

Definition 8: *Modell*
Ein Modell bildet einen existierenden Sachverhalt deskriptiv nach oder nimmt einen Zukünftigen präskriptiv voraus.
Teilweise wird der Begriff *Schema* synonym gebraucht.

Deskriptive Modelle: Modelleisenbahn, Stadtplan, Photo.
Präskriptive Modelle: Bauplan eines Hauses, Skizze eines Gemäldes, maßstäblich verkleinerte Skulptur als Vorbild.

Beispiel 5: Modelle

Definition 9: *Datenbanksprache*
Eine Sprache die zur Erzeugung oder Interaktion mit den Daten bzw. zu deren Verwaltung eingesetzt wird.
Es werden unterschieden:

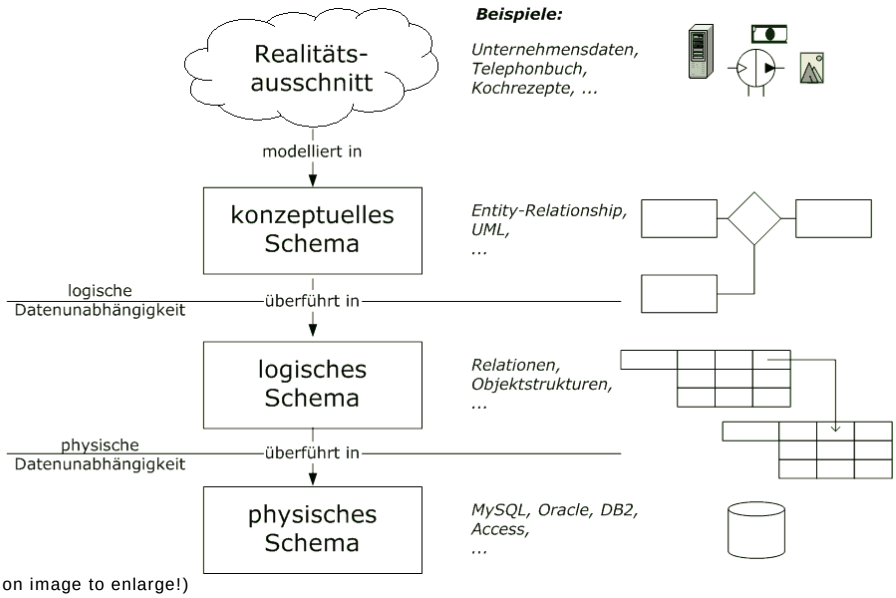
- Data Definition Language (DDL).
  Zur Erzeugung eines Datenmodells.
- Data Manipulation Language (DML).
  Zur Modifikation der verwalteten Daten.
- Data Retrieval Language (DRL).
  Zur Anfrage der in einer Datenbank gespeicherten Daten.
- Data Control Language (DCL).
  Zur Festlegung und Kontrolle von Zugriffsberechtigungen.

Im Verlauf der Vorlesung wird mit *SQL* die bekannteste Sprache im Umfeld relationaler DBMS eingeführt.
Beispiel einer SQL-Anfrage:
```
SELECT FNAME, BDATE FROM EMPLOYEE ORDERED BY BDATE
```

Beispiel 6: Die Datenbanksprache SQL

3-Schema-Architektur

Abbildung 1: *3-Schema-Architektur*



(click on image to enlarge!)

Die Abbildung 1 stellt die 3-Schema-Architektur dar, welche die drei zentralen Modelltypen des Datenbankentwurfsprozesses miteinander in Beziehung setzt.

Definition 10: *Konzeptuelles Schema*
Ein konzeptuelles Schema ist ein Modell, welches den relevanten Realitätsausschnitt (auch *Miniwelt*, *Diskursbereich* oder *Universe of Discourse* genannt) in Struktur und Inhalt beschreibt.

Definition 11: *Logisches Schema*
Ein logisches Schema ist ein Modell, welches paradigmenspezifisch aus einem konzeptuellen Schema abgeleitet wurde.

Die Definition von Relationen als mathematisches Konzept zur Datenstrukturierung stellt ein logisches Schema dar.
Beispielsweise die Festlegung der Struktur der *Person* oder des *Studenten* in Beispiel 3.

Beispiel 7: Relationen sind ein logisches Schema

Definition 12: *Physisches Schema*
Ein physisches Schema ist ein implementierungsspezifisches Modell, welches aus einem logischen Schema abgeleitet wurde.

Definition 13: *Datenunabhängigkeit*
Die Formulierung einer Modellschicht (d.h. eines Datenmodells) ist von den darunter- bzw. darüberliegenden Modellschichten dann datenunabhängig, wenn Änderungen in den „umgebenden“ Modellschichten sich nicht auf die betrachtete Modellschicht auswirken.

Der Vorgang der *Ableitung* zwischen den verschiedenen Modelltypen der 3-Schema-Architektur sollte hierbei idealerweise (aus Gründen der Überprüfbarkeit, Nachvollziehbarkeit, Wiederholbarkeit und Qualitätssicherung) durch einen deterministischen Algorithmus erfolgen.

Die Abbildung 1 zeigt rechts neben den Modelltypen symbolhaft typische graphische Veranschaulichungen der jeweiligen Modellausprägungen.

1.2 Anforderungen an Datenbanksysteme

Allgemein: Speicherung, Verwaltung und Kontrolle der Daten sowie Organisation des u.U. gleichzeitig erfolgenden Zugriffs.
Spezieller:

- Redundanzfreie Datenspeicherung.
  Von dieser Forderung kann bewußt aus Gründen der Geschwindigkeitsoptimierung abgewichen werden.
- Gewährleistung von Integritätsbedingungen und Einhaltung von Regeln.

- Daten-Programm-Unabhängigkeit.

Wünschenswerte Eigenschaften:

- Leistungsfähigkeit
- Skalierbarkeit
- Benutzerfreundlichkeit
- Flexibilität
- ... spezifische Anforderungen, die sich aus der Anwendungssituation ergeben

1.3 Typen von Datenbankmanagementsystemen

Datenbanken werden heute vielfälltig in Wirtschaft, Technik und Wissenschaft eingesetzt.
Für verschiedene Anwendungsgebiete und Strukturen der verwalteten Daten haben sich daher spezifische DBMS-(Unter-)Typen herausgebildet, die diese Anwendungsfelder besonders gut unterstützen:

- **Deduktive Datenbanken**
  Ein um eine Menge von Regeln (Deduktionskomponenten) erweitertes Datenmodell welches logische Schlüsse auf Basis der hinterlegten Fakten ziehen kann.
- **Multimedia Datenbanken**
  Ein System, welches sich besonders zur Verwaltung großer Bild-, Audio- oder Videodaten eignet.
- **Objektdatenbanken**
  Ein System zur Speicherung von Strukturen gemäß dem logischen Objektmodell.
- **Geographische Datenbanken**
  Ein System das sich besonders zur Verwaltung geographischer Daten (z.B. Landkarten) eignet.
- **XML-Datenbanken**
  Ein System zur Speicherung gemäß dem logischen Modell des XML Information Sets.
- **Aktive Datenbanken**
  Ein System zur selbständigen Reaktion auf externe Ereignisse.
- **Temporale Datenbanken**
  Ein System, welches neben den reinen Datenbeständen auch die Zeit des Datenzustandes mitverwaltet.

Exkurs

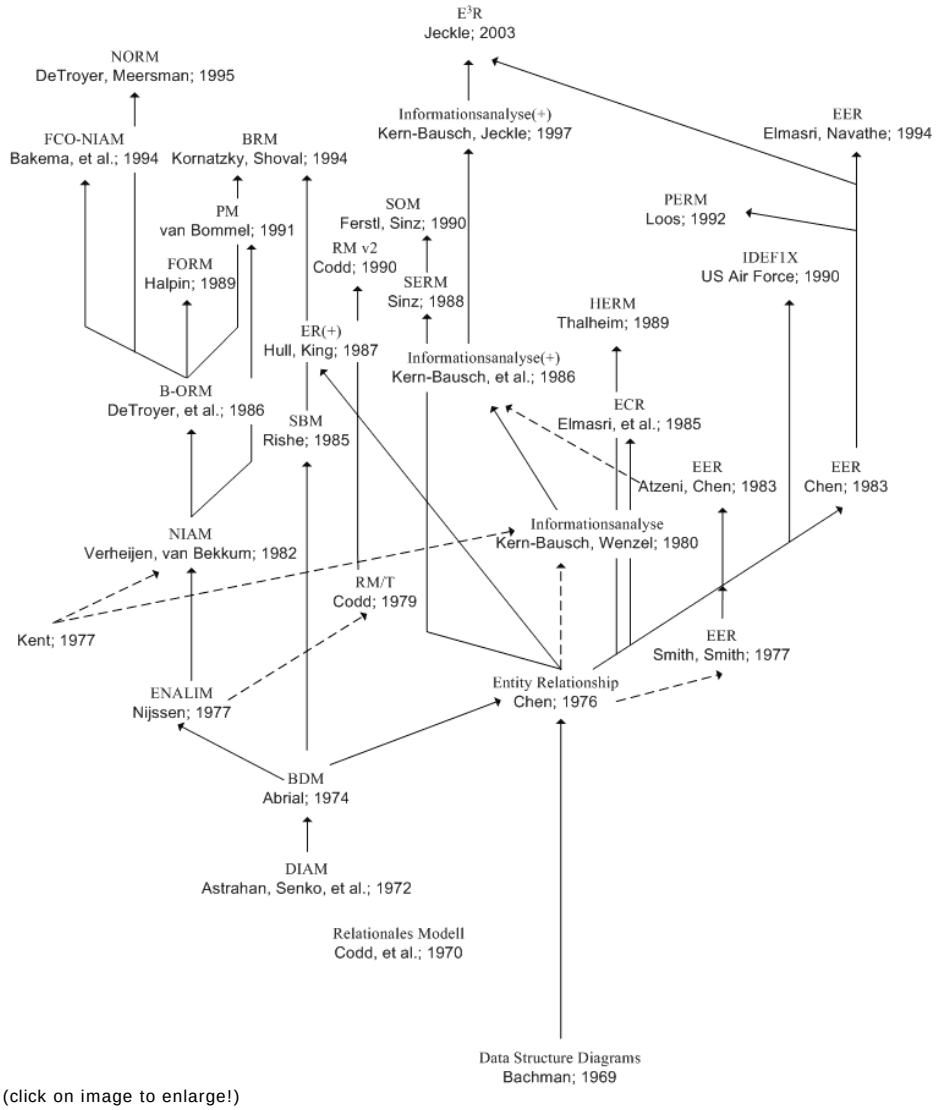
Erste Gehversuche mit dem RDBMS MySQL

2 Entwurf einer Datenbank

2.1 Graphischer Entwurf des konzeptuellen Schemas mit dem Entity-Relationship Modell

Seit der wirkungsmächtigen Erstveröffentlichung des *Entity Relationship Modells* (ERM) durch P. Chen 1976 kommt dieser Modellierungssprache zur Erstellung des konzeptuellen Schemas die uneingeschänkt größte Bedeutung in der Praxis zu.
In der Folgezeit wurden verschiedene Weiterentwicklungen des ursprünglichen ERM vorgeschlagen, die das Originalmodell in verschiedenen Richtungen erweitern. Hierunter fallen die Einführung von Konstrukten zur Abbildung hierarchischer Beziehungen ebenso wie Primitive zur Darstellung von Aggregationsbeziehungen.
Die Graphik der Abbildung 2 zeigt eine Auswahl verschiedener Entwicklungen rund um das initiale ERM sowie einige zentrale Weiterentwicklungen. Innerhalb der Abbildung ist unterhalb des Namens der Modellierungssprache (sofern vorhanden, bei Weiterentwicklungen ohne eigenständige Namensgebung ist zur Unterscheidung vom Vorgängermodell ein geklammertes Pluszeichen angetragen) der Autor sowie das Jahr der Erstveröffentlichung dargestellt.

Abbildung 2: *Entwicklungslinien des ER-Modells*



(click on image to enlarge!)

Im oberen Bereich der Abbildung ist das *Semantically Enriched Extended Entity Relationship Model* (E^3R) dargestellt, welches im Rahmen dieser Vorlesung behandelt wird.
Es stellt eine kompatible Entwicklung dar, die versucht die datenorientierten Aspekte der ER-Nachfolgemodelle mit denen der semantischen Datenmodellierung zu vereinigen.

Die Grundkonzepte des E^3R-Modells sind:

Definition 14: *Entität*
Eine Entität ist ein eindeutig identifizierbares und daher wohlunterscheidbares „Ding".
Im graphischen E^3R-Modell werden Entitäten durch benannte Rechecke dargestellt, die im Zentrum den unterstrichenen Namen der Entität, gefolgt vom -- in Klammern angegebenen -- Namen des Entitätstypen, tragen.

Anmerkung: Der Begriff *Ding* wird hierbei in seiner Bedeutung als Synonym von *Seiendes, Gegenstand* oder *Objekt* gebraucht. Die philosophische Terminologie detailliert den Begriff zusätzlich hinsichtlich seiner Verwendung zur Beschreibung raumzeitlicher Gegenstände mit festgelegten charakteristischen (substantiellen) und zufällig anhaftenden Eigenschaften (Akzidenzien) aus.

- Die Tafel direkt vor ihnen.
- Sie selbst.
- Dieser Hörsaal.

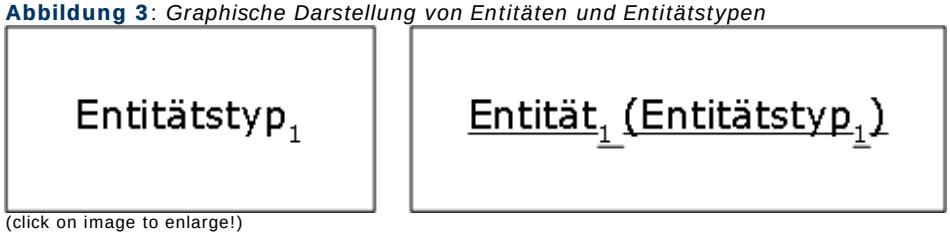
Beispiel 8: Beispiele für Entitäten

Definition 15: *Entitätstyp*
Ein Entitätstyp ist eine ungeordnete und duplikatfreie Sammlung von als logische zusammengehörig betrachteten Entitäten.
Im graphischen E^3R-Modell werden Entitätstypen durch benannte Rechtecke dargestellt. Der Name eines Entitätstyps muß dabei schemaweit eineindeutig sein.

- Tafel.
- Person.
- Student.

Beispiel 9: Beispiele für Entitätstypen

Abbildung 3: *Graphische Darstellung von Entitäten und Entitätstypen*



(click on image to enlarge!)

Soll ein Hinweis auf eine spätere physische Realisierung (d.h. die gewählte Form der Abspeicherung von Entitäten in der Datenbank) gegeben werden, so kann einem Entitätstypen ein Repräsentationstyp zugeordnet werden, bzw. einer Entität eine Repräsentation.

Definition 16: *Repräsentationstyp*
Ein Repräsentationstyp führt einen physischen Typ in das konzeptuelle Schema ein, der zur technischen Implementierung eines durch ihn annotierten Entitätstypen herangezogen werden kann.
Als Repräsentationstypen sind beliebige atomare (d.h. in ihrer Semantik nicht weiter verlustfrei zerlegbare) Datentypen eines logischen oder physischen Modells zugelassen.

- Integer
- Datum
- Money
- String

Beispiel 10: Beispiele für Repräsentationstypen

Definition 17: *Repräsentation*
Eine Repräsentation ist eine Ausprägung genau eines Repräsentationstypen.

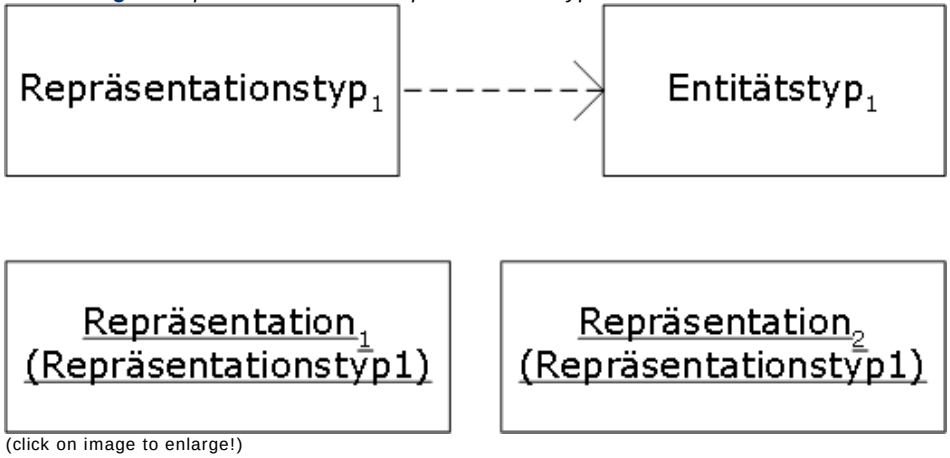
Der zugehörige Repräsentationstyp ist in Klammern angegeben.

- 42 (Integer)
- 2004-06-08 (Datum)
- €99,95 (Money)
- "Hallo Welt!" (String)

Beispiel 11: Beispiele für Repräsentationen

Die graphische Darstellung erfolgt durch benannte Rechtecke. Repräsentationen werden unterstrichen mit der geklammerten nachfolgenden Angabe des Repräsentationstypen dargestellt. Repräsentationstypen werden durch eine gerichtete Kante mit unterbrochener Linienführung mit dem durch sie repräsentierten Entitätstypen verbunden.

Abbildung 4: *Repräsentation und Repräsentationstyp*

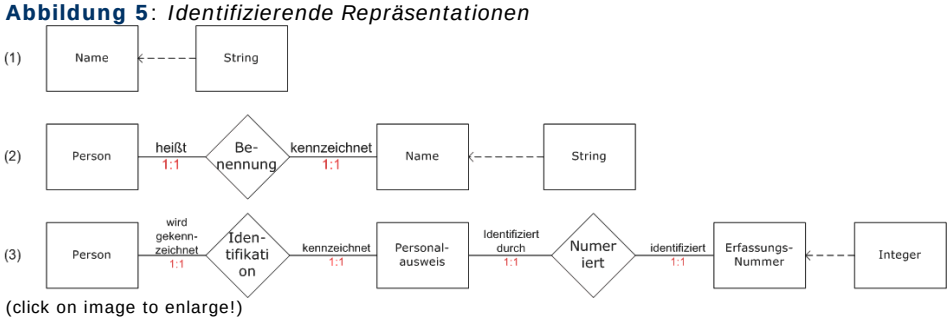


(click on image to enlarge!)

Das Beispiel der Abbildung 5 zeigt verschiedene Beispiele für die Verknüpfung von Entitätstypen

mit ihren zugehörigen Repräsentationstypen.
Im Teilbeispiel (1) wird der Entitätstype `Name` unmittelbar durch den Repräsentationstypen `String` repräsentiert, d.h. die spätere physische Realisierung des Entitätstypen `Name` wird durch den Datentyp `String` erfolgen.
Teilbeispiel (2) zeigt eine transitive Repräsentation (genaugenommen eine transitive Repräsentation erster Ordnung). Hier ist der Entitätstyp `Person`, welcher selbst über keinen Repräsentationstypen verfügt, in eindeutiger Weise (d.h. über einen Assoziationstyp der ausschließlich über Kardinalitsintervalle von `1:1` verfügt (nach Maßgabe der Anmerkung zur Struktur der Kardinalitätsintervalle kann es sich daher nur um einen binären Assoziationstypen handeln)) mit dem Entitätstypen `Name` verknüpft, der über die Repräsentation `String` verfügt.
Abschließend zeigt das Teilbeispiel (3) die transitive eindeutige Assoziierung des Entitätstypen `Person` mit dem Entitätstypen `Personalausweis` durch den Assoziationstypen `Identifikation`, wobei `Personalausweis` seinerseits in eindeutiger Weise mit der durch `Integer` repräsentierten Erfassungsnummer assoziiert ist.

Abbildung 5: *Identifizierende Repräsentationen*



(click on image to enlarge!)

Definition 18: *Assoziation*
Eine Assoziation ist eine benannte n-äre Beziehung (n>1) zwischen Entitäten.
Die Semantik jeder durch eine Assoziation verbundenen Entität wird durch Angabe einer innerhalb einer Assoziation für jede verbundene Entität eindeutigen Rolle konkretisiert.
Im graphischen E^3R-Modell wird eine Assoziation durch eine Raute dargestellt, die durch ungerichtete Kanten mit Entitäten verbunden ist. Im Zentrum wird der Name der Assoziation, gefolgt vom in Klammern geschriebenen Namen des Assoziationstypen plaziert. Zusätzlich sind die beiden Namen unterstrichen dargestellt.

Definition 19: *Assoziationstyp*
Ein Assoziationstyp ist eine duplikatfreie ungeordnete Sammlung von logisch als zusammengehörig betrachteten Assoziationen.
Jede zu einem Assoziationstypen beitragende Rolle wird durch ein Kardinalitätsintervall ergänzt.
Im graphischen E^3R-Modell wird ein Assoziationstyp durch eine Raute dargestellt, die durch ungerichtete Kanten mit Entitätstypen verbunden ist. Im Zentrum des Assoziationstypen wird sein schemaweit eineindeutiger Name plaziert.

- Arbeitsverhältnis.
- Ehe.
- Verwandschaft.

Beispiel 12: Beispiele für Assoziationstypen

Definition 20: *Kardinalitätsintervall*
Ein Kardinalitätsintervall legt die Anzahl derjenigen Entitäten fest, die mit einer die Rolle einnehmenden Entität zu einem Zeitpunkt innerhalb einer Assoziation verbunden sein können.
Das Intervall wird in der Schreibweise „i:j“ angegeben, wobei *i* eine beliebige natürliche Zahl oder die Null ist und *j* eine beliebige natürliche Zahl oder das Symbol *n* ist. Zusätzlich gilt: i<=j.

- 0:1.
- 3:7.
- 0:n.
- 1:n.
- 99:n.

Beispiel 13: Beispiele für Kardinalitätsintervalle

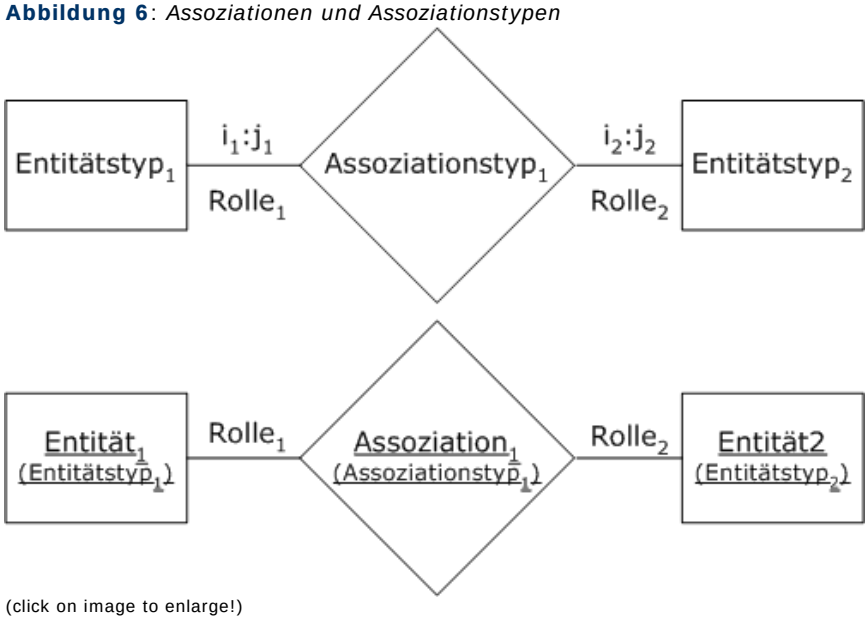
Ungültig hingegen sind:

- 7:0 (Obergrenze kleiner als Untergrenze).
- -5:7 (-5 ist keine natürliche Zahl.)

- n:8 (*n* ist nicht als Untergrenze erlaubt.)

Allgemein gilt: Für *n*-äre Assoziationstypen gilt die Einschränkung, daß die Maximalkardinalität die ein Entitästyp zu einem *n*-ären Assoziationstypen beitragen darf größer gleich *n*-1 ist.

Abbildung 6: *Assoziationen und Assoziationstypen*



(click on image to enlarge!)

Zentrales Konzept des E^{3}R-Modells ist die Idee der Rolle, welche als hauptinformationstragendes Konstrukt fungiert:

Definition 21: *Rolle*
Eine Rolle die durch einen Entitätstypen innerhalb eines Assoziationstypen eingenommen wird charakterisiert die konkrete Verwendung von Entitäten des gegebenen Typs im Kontext der Assoziationen die zum betrachteten Assoziationstyp zusammengefaßt werden.

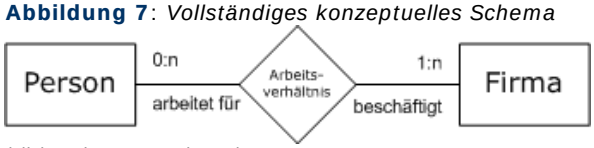
Anmerkung: Für den relationalen Datenbankentwurf ist es notwendig, daß jeder im konzeptuellen Schema modellierte Entitätstyp entweder über einen Repräsentationstyp verfügt oder über eine *Namenskonvention*, d.h. eine binäre Assoziationstyp deren Kardinalitätsintervalle ausschließlich auf 1:1 festgelegt sind, die den Entitätstyp direkt oder transitiv mit einem mit Repräsentation versehenen Entitätstypen verbindet.

Gleichzeitig wird durch die Rolle der Brückenschlag zwischen natürlicher Sprache und formaler graphischer Darstellung im E^{3}R-Modell ermöglicht.
So lassen sich die Sätze

- Jede Person arbeitet optional für mehrere Firmen.
- Jede Firma beschäftigt ein oder mehrere Personen.

in das nachfolgende konzeptuelle Schema überführen:

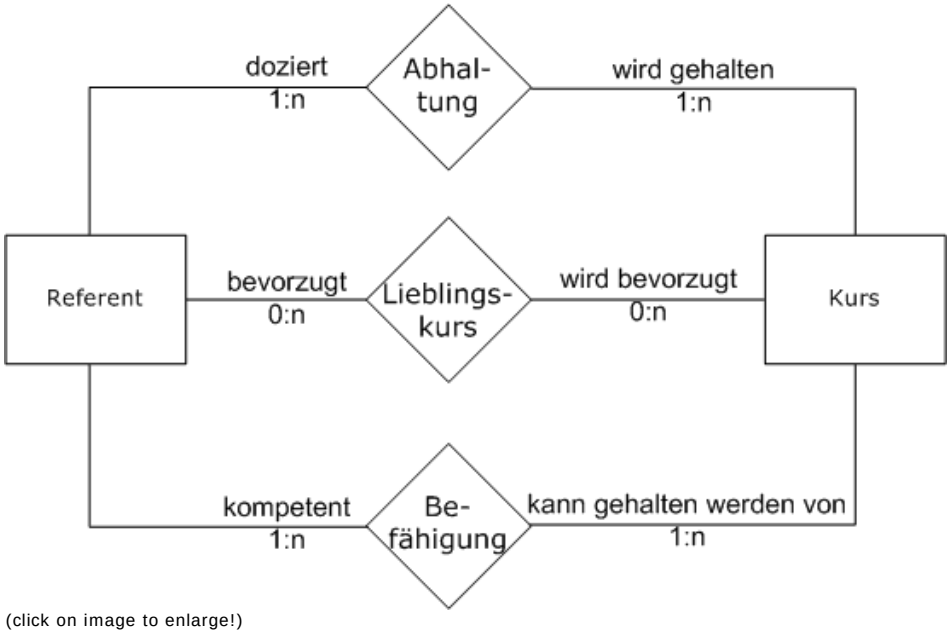
Abbildung 7: *Vollständiges konzeptuelles Schema*



(click on image to enlarge!)

Zusätzlich zeigt das konzeptuelle Schema der Abbildung 8 die Mächtigkeit des Rollenkonzepts zur Darstellung verschiedener Informationszusammenhänge.
So enthält das abgebildete konzeptuelle Schema die drei verschiedenen Assoziationstypen `Abhaltung`, `Lieblingskurs` und `Befähigung` welche ausschließlich Rollen enthalten die durch die beiden dargestellten Entitätstypen `Referent` und `Kurs` gespielt werden.

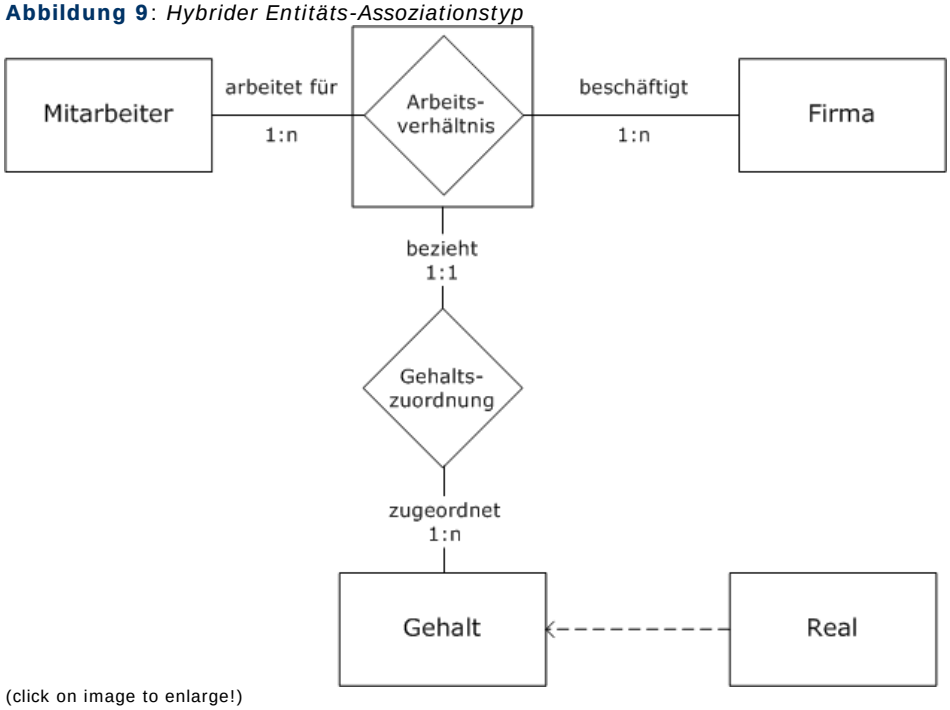
Abbildung 8: *Verschiedene Rollen*



(click on image to enlarge!)

Definition 22: *Hybrider Entitäts-Assoziationstyp*
Ein hybrider Entitäts-Assoziationstyp vereinigt das Sprachelement des Entitätstyps und des Assoziationstyps in sich und bewahrt die Semantik beider Konstrukte.

Abbildung 9: *Hybrider Entitäts-Assoziationstyp*

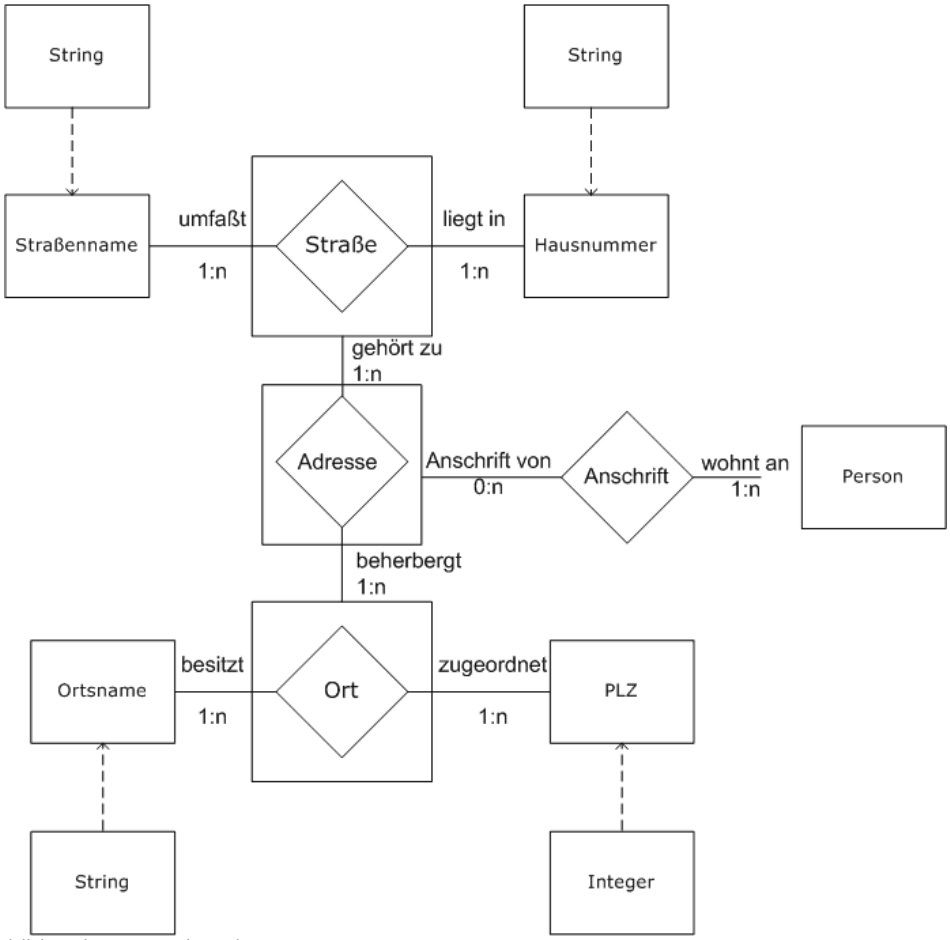


(click on image to enlarge!)

Abbildung Abbildung 10 stellt die Informationsstruktur einer Adresse dar.
Dabei zeigt das konzeptuelle Schema die Verwendung der hybriden Entitäts-Assoziationstypen.
So können jedem `Straßennamen` beliebig viele `Hausnummern` zugeordnet werden und umgekehrt.
Jeweils zwei dieser Angaben zusammen bilden die `Straße`.
Jedem `Ortsnamen` kann über mehrere `Postleitzahlen` verfügen, ebenso kann dieselbe Postleitzahl mehreren gleich benannten Orten zugeordnet werden (Beispiel: Ortsteile).
`Postleitzahl` und `Ortsname` zusammen bilden den `Ort`.
Aus der Kombination von `Straße` und `Ort` wird eine `Adresse` gebildet. Dabei kann jede `Straße` (=Kombination aus `Straßenname` und `Hausnummer`) mehreren Orten (=Kombination aus `Ortsname` und `Postleitzahl`) und umgekehrt zugeordnet sein.

Das Beispiel unterstreicht die alleinige Bildbarkeit hybrider Entitäts-Assoziationstypen beim vorliegen von Kardinalitätsintervallen, die alle über ein Maximum größer 1 verfügen.
Vgl. hierzu Aussagen der Anmerkung zur Bildung von Kardinalitätsintervallen

Abbildung 10: *Informationsstruktur Adresse*

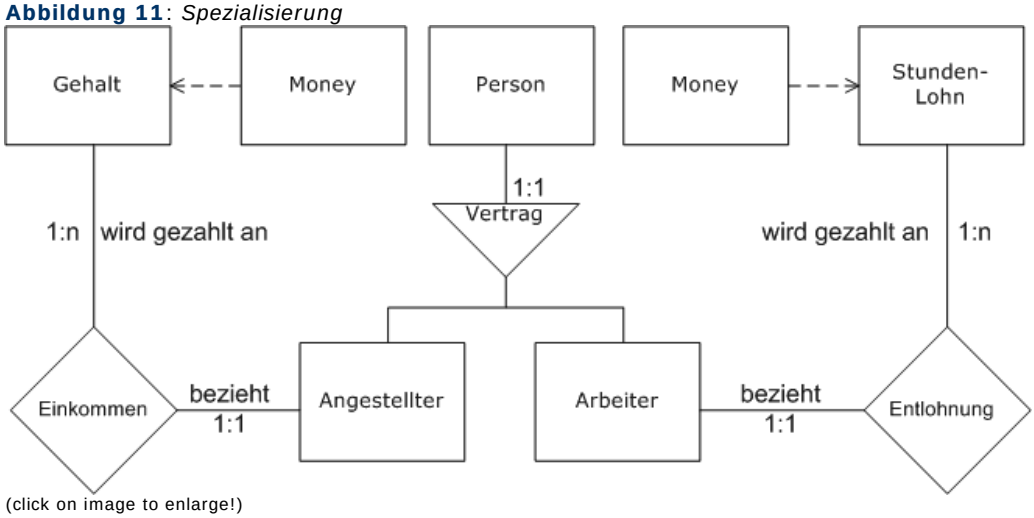


(click on image to enlarge!)

Weiterführende Konzepte

Definition 23: *Spezialisierungsassoziationstyp*
Ein Spezialisierungsassoziationstyp ist eine duplatfreie ungeordnete Sammlung von logisch als zusammengehörig betrachteten Assoziationen.Zu den in der Menge enthaltenen Assoziationen tragen der zu spezialisierende Entitätstyp (der sog. *Super-* oder *Obertyp*) und der spezialisierende (entsprechend als sog. *Sub-* oder *Untertyp* bezeichnet) Rollen bei.
Die Rolle des Supertyps ist hierbei auf *wird spezialisiert zu* fixiert, als Kardinalitätsintervalle sind ausschließlich `0:1`, `0:n`, `1:1` und `1:n` zulässig.
Die Rolle des Subtyps ist auf *ist Spezialisierung von* mit dem Kardinalitätsintervall `1:n`fixiert.
Jeder Spezialisierungsassoziationstyp wird durch ein Distinktionsmerkmal charakterisiert, das expliziert hinsichtlich welchen Merkmals die Spezialisierung gebildet wird.
Die Verknüpfung durch einen Spezialisierungsassoziationstyp bewirkt, daß alle Assoziations- und Repräsentationstypen, die für den Supertyp definiert sind auch automatisch für alle Subtpyen definiert werden.

Abbildung 11: *Spezialisierung*



(click on image to enlarge!)

Die Abbildung 11 zeigt die Spezialisierung des Entitätstypen `Person` hinsichtlich des Distinktionsmerkmals `Vertrag`. Dabei gilt: Jede Person kann nur genau einmal (`1:1`) hinsichtlich

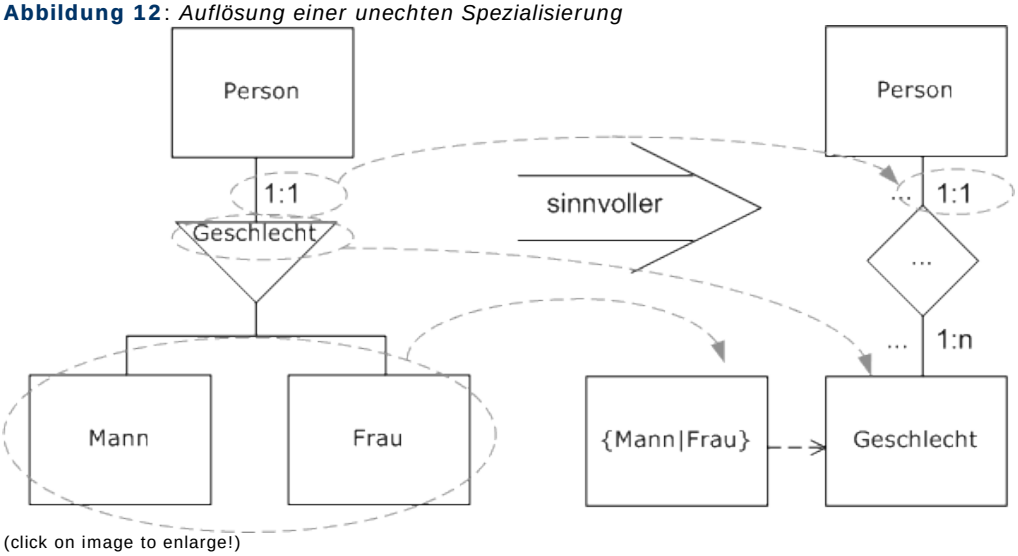
ihres (Arbeits-)Vertrages zu `Angestellter` oder `Arbeiter` spezialisiert werden.

Abbildung 11 zeigt ferner, daß die spezialisierten Entitätstypen `Angestellter` und `Arbeiter` über zusätzliche, d.h. für den Obertyp `Person` nicht definierte, Eigenschaften (`Gehalt` bzw. `Stundenlohn`) verfügen.

Zur Pragmatik des Spezialisierungsassoziationstyps:
Spezialisierungsassoziationstypen sollten ausschließlich dann eingesetzt werden, wenn es sich um „echte“ Spezialisierungen handelt. Eine *echte Spezialisierung* liegt immer dann vor, wenn jeder spezialisierte Entitätstyp über Assoziationstypen verfügt, die der allgemeinere Obertyp nicht besitzt.

Gilt dieses Kriterium nicht, d.h. können für die spezialisierten Entitätstypen keine zusätzlichen Eigenschaften gebildet werden, dann solte die Spezialisierung *nicht vorgenommen* werden.
In diesem Falle bietet sich die Überführung der unnötigen Spezialisierung eine Eigenschaft des (vermeintlichen) Obertypen an. Zusätzlich sollte ein Entitätstyp zur Aufnahme derjenigen Information gebildet werden, für welche die Darstellung als Spezialisierungsassoziationstyp intendiert war.
Dieser neue Entitätstyp kann geeignet mit einem Repräsentationstyp versehen werden, um die unterscheidende (distingierende) Information darzustellen.
Die einzelnen Schritte sind in Abbildung 12 beispielhaft zusammengestellt.

Abbildung 12: *Auflösung einer unechten Spezialisierung*



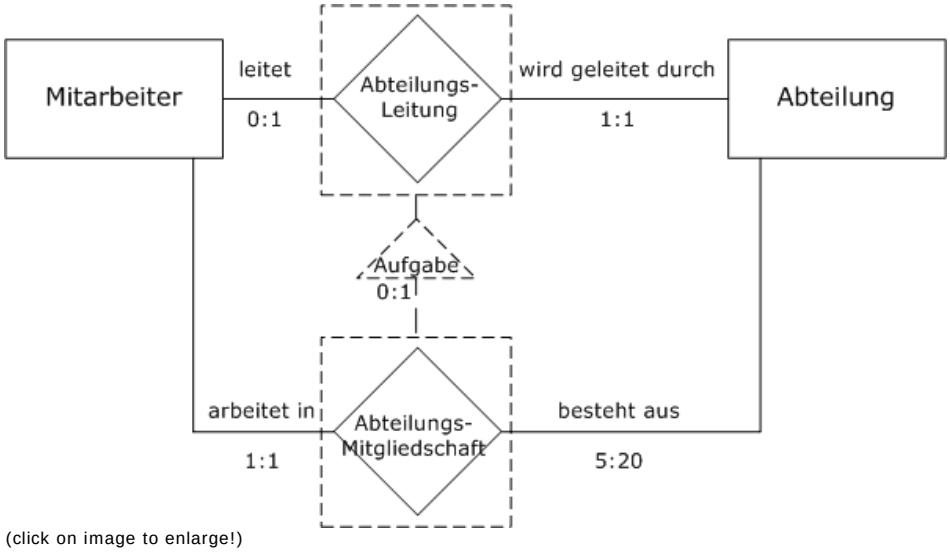
(click on image to enlarge!)

Definition 24: *Metainformation*
Informationsanteile eines Modells, die nicht direkt in das logische Schema übernommen werden, sondern in konsistenzgarantierende Regeln oder Applikationscode abgebildet werden.

Die Abbildung 13 veranschaulicht die Nutzung des Spezialisierungsassoziationstyps zur Formulierung von Metainformation. Im Beispiel wird gefordert, daß jeder `Abteilungsleiter` auch gleichzeitig als Mitarbeiter der durch ihn geleiteten Abteilung erfaßt sein muß.

Hinweis: Metainformation muß nicht zwingend semantisch irreduzibel erfaßt werden, wie die --- eigentlich illegale Bildung der beiden hybriden Entitäts-Assoziationstypen `Abteilungsleitung` und `Abteilungsmitgliedschaft` zeigt.

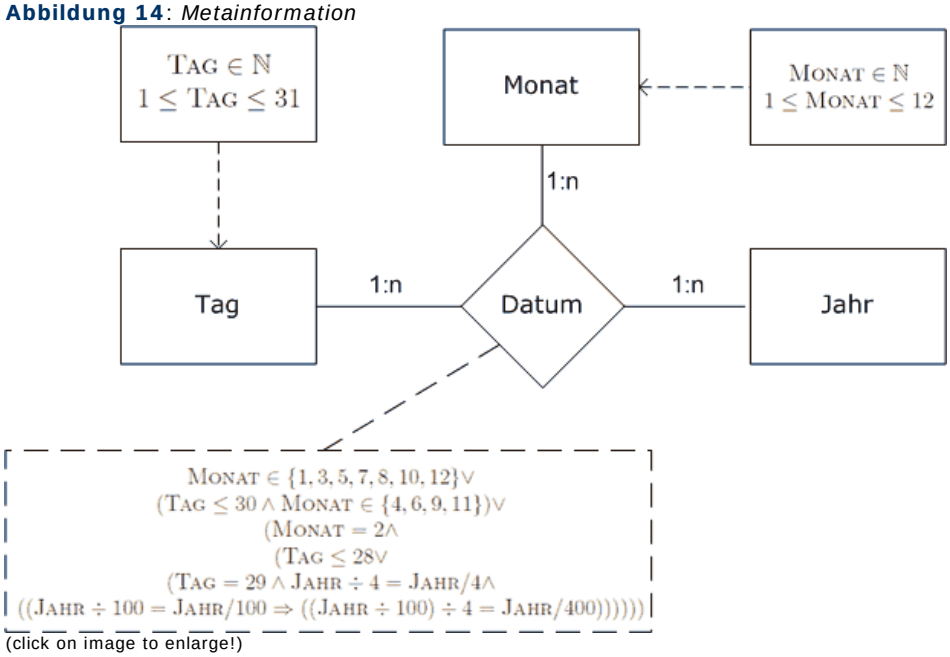
Abbildung 13: *Metainformation*



(click on image to enlarge!)

Das konzeptuelle Schema der Abbildung 14 zeigt ein Beispiel für konsistenzgarantierende Metainformation, die nicht durch E^{3}R-Syntax ausdrückbar ist und daher in textueller Form annotiert wird.

Abbildung 14: *Metainformation*



(click on image to enlarge!)

Phasenmodell der Erstellung eines konzeptuellen Schemas mit E^{3}R

E^{3}R ist eine Notation und Methode zur Entwicklung des konzeptuellen Schemas für jede beliebige Art von Kommunikationssituationen.

Phase 0 ist die Vorbereitungsphase, die von der Idee, ein E^3-Schema für einen Realitätsausschnitt zu erstellen, über die Auswahl der Beteiligten bis zu ihrer Ausbildung in den Techniken zur Darstellung von Information und in der Vorgehensweise der Analyse reicht.

Es folgt die Festlegung des Informationsbereichs (**Phase 1**). Hier werden die für den gewünschten Anwendungsbereich relevanten Entitäten und Assoziationen gesammelt und zu Entitäts- und Assoziationstypen zusammengefaßt. Eine große Hilfe dabei sind verbale Beschreibungen der in der Datenbank zu verwaltenden Information, kommentierte Listen mit Daten des betrachteten Informationsbereichs oder ähnliches. Das Ergebnis ist eine erste, grobe Struktur der relevanten Information.

Diese Struktur wird in **Phase 2** immer weiter verfeinert, wobei man für jeden Entitätstyp entweder direkt oder transitiv eine Repräsentation definiert.
Dann werden alle relevanten Eigenschaften, die eine Entität eines Typs haben kann, in Form von semantisch irreduzibel formulierten Assoziationstypen beschrieben. Dabei treten erfahrungsgemäß neue, zuvor nicht berücksichtigte Entitätstypen auf.

Deshalb wird die Phase 2 solange inkrementell iteriert, bis keine neuen Entitätstypen mehr

identifiziert werden zu denen noch Repräsentation zu definieren oder durch Assoziationstypen anzubinden sind.

Bis zu diesem Punkt standen strukturelle, formale Gesichtspunkte im Vordergrund. In **Phase 3** treten diese zurück; nun stehen semantische Gesetzmäßigkeiten im Vordergrund, soweit diese nicht bereits in den Phasen 1 und 2 erkannt und behandelt worden sind.

Ziel der Phase 3 ist es, die bis dato erstellte Informationsbeschreibung geeignet zu ergänzen um auch alle nicht durch die E^3R-Notation darstellbaren Konsistenzregeln zu erfassen.
Zusätzlich kann die E^3R-Notation zur Formulierung von Metainformation auf einer höheren Modellebene angewendet werden.

Hinweis: Es kann beim Erstellen des konzeptuellen Schemas durchaus vorkommen, daß sich das Ergebnis der vorausgegangenen Phase als unvollständig herausstellt. In diesem Fall ist es unbedingt notwendig, in diese Phase zurückzukehren und dann mit dem korrigierten Ergebnis dieser Phase weiterzuarbeiten. Dies ist kein Wegwerfen der bisher geleisteten Arbeit, denn meist genügen einige wenige Streichungen und Ergänzungen.
Bleibt diese Regel unberücksichtigt, so nimmt begibt man sich der Möglichkeit wichtige Eigenschaften der Information im konzeptuellen Schema eindeutig festzuhalten. Dabei spricht das Verhältnis zwischen der gewonnen Exaktheit und dem zusätzlichen Aufwand sehr zugunsten der exakten und sauberen Lösung.

Resultat der korrekten Anwendung des Phasenmodelles ist ein *vollständiges konzeptuelles Schema* als Voraussetzung der Umsetzbarkeit in beliebige logische Strukturen.

Definition 25: *Vollständiges konzeptuelles Schema*
Ein vollständiges konzeptuelles Schema ist ein E^3R-Schema in dem alle Entitäts- und Assoziationstypen, sowie alle Rollen benannt sind. Darüberhinaus ist jede Rolle mit einem korrekten Kardinalitätsintervall versehen, sowie jedem Entitätstypen direkt oder transitiv ein Repräsentationstyp zugeordnet.
Sofern Metainformation existiert, ist diese auch in adäquater Weise dargestellt.

Fallstudie: Fächerdatenbank

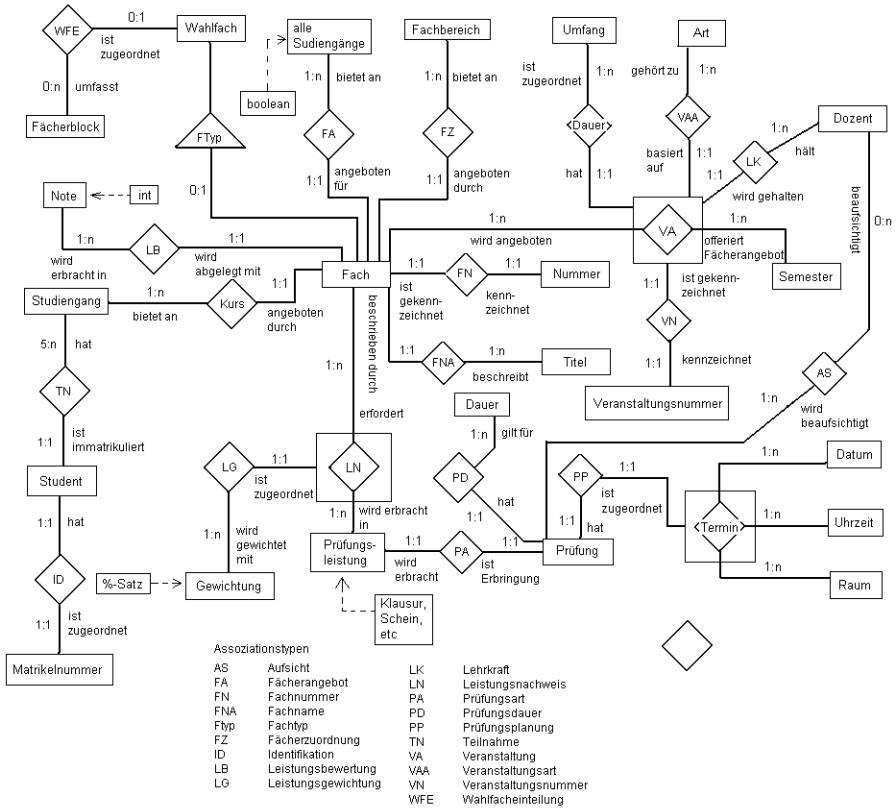
Der Fachbereich möchte die Belegung der Fächer in einer Datenbank abspeichern; hierfür gelten folgende semantische Regeln:

Die vorgesehenen Fächer haben eine feste Nummer, die sich niemals ändert sowie eine einen längeren Titel. Zusätzlich sind sie von einem Fachbereich entweder für einen speziellen Studiengang (z.B. *WIB*) oder allen Studiengängen angeboten, dann gilt die Zuordnung *FH*. Gleichzeitig kann jedes Wahlfach einem Fächerblock (z.B. *Consulting* oder *Informatik*) zugeordnet sein.

Die in einem bestimmten Semester angebotenen Fächer erhalten eine Veranstaltungsnummer, die nur für dieses Semester gilt. Dazu wird der jeweilige Dozent angegeben und die Art der Lehrveranstaltung (*Vorlesung*, *Seminar*, *Praktikum* etc.) sowie ihr Umfang in Semesterwochenstunden.
Die Notenbildung in jedem Fach kann durch eine oder mehrere Prüfungsleistungen erfolgen (z.B. *Leistungsnachweis*, *Schein*, *Prüfung*, etc.), die in unterschiedlichen Prozentsätzen gewichtet werden. Jede Prüfung findet zu einem festgelegten Datum in einem Raum zu einer Uhrzeit statt und wird durch mindestens einen Dozenten beaufsichtigt. Zusätzlich soll die Dauer der Prüfungsleistung vermerkt werden.

Ein Student ist durch eine eindeutige Matrikelnummer gekennzeichnet. Zusätzlich wird sein Studiengang gespeichert.

Abbildung 15: *Konzeptuelles Schema der Fallstudie*



(click on image to enlarge!)

2.2 Ableitung logischer Relationenstrukturen

Erweiterung der Grundbegriffe des Relationenmodells

Definition 26: *Superschlüssel*
Ein Superschlüssel *SK* ist eine nicht-leere Teilmenge von Attributen einer Relation für die gilt, daß zwei verschiedene Tupel t_1 und t_2 dieser Relation keine gleiche Wertbelegung aufweisen.

Der Superschlüssel definiert damit eine Eindeutigkeitkeitseinschränkung, nach der zwei Tupel allein über die Betrachtung der im Superschlüssel zusammengefaßten Attribute unterscheidbar sind.

Gegeben sei die Relation *Mitarbeiter*:

Vorname	Nachname	Geburtsdatum	Persausweisnummer
Xaver	Obermüller	1970-03-04	134975459
Rosi	Hinterhuber	1973-06-02	781367519
Rosi	Obermüller	1963-11-03	783148384
Hans	Hinterhuber	1970-03-04	977554422

Mögliche Superschlüssel dieser Relation sind:

- (Vorname, Nachname, Personalausweisnummer)
- (Nachname, Geburtsdatum, Personalausweisnummer)
- (Geburtsdatum, Personalausweisnummer)
- ...

Beispiel 14: Beispiele für Superschlüssel

Definition 27: *Schüssel*
Ein Schlüssel *K* ist ein Superschlüssel, der sofern man ein Attribut aus ihm entfernt nicht mehr eindeutigkeitseinschränkend wirkt.

Der einzige Schlüssel der Relation Mitarbeiter ist `Personalausweisnummer`.

Beispiel 15: Beispiele für Schlüssel

Anmerkungen:

- Die Menge aller Attribute einer Relation ist immer Superschlüssel.
- Jeder Schlüssel ist auch ein Superschlüssel.
  Der Umkehrschluß gilt nicht, da Schlüssel eine schärfere Forderung darstellt.
- Jeder Schlüssel ist zwingend eine minimal identifizierende Attributkombination.

Häufig tritt es in der Praxis auf, daß sich in einer Relation mehr als ein Schlüssel finden läßt. Jeder dieser möglichen gleichwertigen Schlüssel wird daher als *Schlüsselkandidat* bezeichnet.

Gegeben sei die Relation *Lagerverwaltung*:

Lagerplatz	Produktnummer	Produktname	Menge
7952	7946	Wusch Superfein	3
7412	9854	Blitzbank Extra	5
7894	6542	Maiengrün natur	7
9461	8954	Gelber Gigant	5

Die Relation enthält folgende Schlüsselkandidaten.

- `Lagerplatz`
- `Produktnummer`
- `Produktname`

Beispiel 16: Beispiele für Superschlüssel

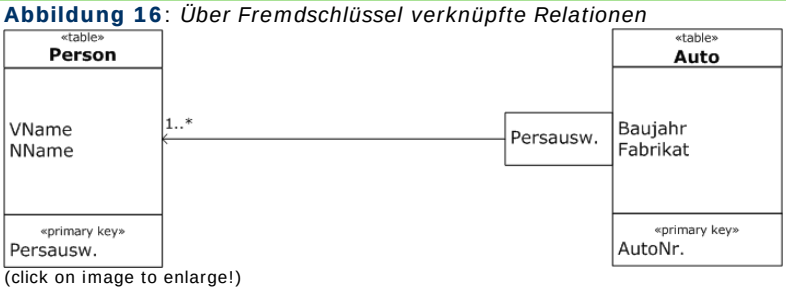
Definition 28: *Primärschüssel*
Ein Primärschlüssel *P* ist ein Schlüssel, der als identifizierendes Merkmal ausgewählt wurde.

In der graphischen Darstellung der Demo-DB sind die Primärschlüssel durch Unterstreichung der beitragenden Attribute hervorgehoben.

Zur Wahrung der Konsistenz innerhalb einer relationalen Datenbank wird üblicherweise u.a. das Mittel der *referentiellen Integrität* eingesetzt, um gleicher Wertinhalte in Attributen (derselben oder verschiedener Relationen) aufeinander abzustimmen.

Definition 29: *Referentielle Integrität*
Attributwerte einer durch referentielle Integrität verknüpften Relation müssen auch in der verknüpften Relation existieren.

Abbildung 16: *Über Fremdschlüssel verknüpfte Relationen*



(click on image to enlarge!)

Das Attribut `Persausw.` der Relation `Auto` verweist auf den Primärschlüssel gleichen Namens der Relation `Person`.
Als Konsequenz dürfen für `Persausw.` in `Auto` nur Werte definiert werden, die sich bereits im Attribut `Persausw.` von `Person` finden.

Beispiel 17: Beispiel für referentielle Integrität

Die Prüfung von Primärschlüsselwerten erfordert u.U. eine Reihe zusätzlicher Datenbankzugriffe. Zu ihrer Beschleunigung können zusätzliche Speicherbereiche, sog. *Indexe* angelegt werden.

Definition 30: *Index*
Ein Index ist ein zusätzlicher Speicherbereich der in der Datenbank verwaltet wird um den lesenden Zugriff auf einzelne Tupel zu beschleunigen.

In der Konsequenz der Beschleunigung der lesenden Zugriffe durch zusätzlichen Speicherplatz verringert sich die Geschwindigkeit der schreibenden Zugriffe (Erzeugung, Aktualisierung und Löschung) etwas.

Die Tabelle `tab` besteht aus zwei Attributen `UUID1` und `UUID2`, wobei ersteres duplikatfrei indexiert wird.

Aktion	Dauer ohne Index [sec]	Dauer mit Index [sec]
Einfügen von 10.000.000 Tupeln `INSERT INTO tab VALUES (...)`	1812	2025
Auswahl aller Tupel `SELECT COUNT(*) FROM tab WHERE UUID1<> "X"` (`UUID1` enthält niemals den Wert X daher werden alle Tupel selektiert)	2100	1800
Auswahl genau eines Tupels `SELECT UUID2 FROM tab WHERE UUID1="..."`	0,422	0,028
Aktualisierung genau eines Tupels `UPDATE tab SET UUID2="Z" WHERE UUID1="..."`	0,415	0,033
Aktualisierung keines Tupels, jedoch vollständige Durchsuchung eines Attributs. `UPDATE tab SET UUID2="Z" WHERE UUID1<>"X"`	0,395	0,014
Löschung eines Tupels `DELETE FROM tab WHERE UUID1="..."`	0,431	0,043
Löschung keines Tupels, jedoch vollständige Durchsuchung eines Attributs. `DELETE FROM tab WHERE UUID1="x"`	0,037	0,008

Beispiel 18: Geschwindigkeitsverhalten mit/ohne Index

Die Erstellung des Index nimmt, bei in der Tabelle gehaltenen 10.000.000 Tupeln 363,063 Sekunden in Anspruch.

Definition 31: *NULL-Wert*
Fehlende Attributwerte in einer Relation werden durch den gesonderten Datenbankeintrag `NULL` dargestellt.

Für die Wertbelegung `NULL` stellt das DBMS sicher, daß sie nicht mit der Ziffer 0 oder dem leeren String kollidiert.

Der Algorithmus

Zentrale Zielsetzung der Erstellung des konzeptuellen Schemas ist die Möglichkeit von ihm ausgehend unterschiedliche logische Modelle, die später in die physische Implementierungssicht abgebildet werden, ableiten zu können.
Dieser Abschnitt stellt einen Algorithmus vor, der es erlaubt aus dem mit E^3R formulierten konzeptuellen Schema logische Strukturen gemäß dem Relationenmodell abzuleiten.

Dabei operiert der Algorithmus ausschließlich auf der graphischen Repräsentation des konzeptuellen Schemas und kann daher auch von entsprechend ausgebildeten Fachexperten manuell durchgeführt werden.
Der durch den Algorithmus abgeleitete logische Datenbankentwurf orientiert sich an festgelegten Gütekriterien um einen redundanzfreien und somit anomalienfreinen Entwurf zu gewährleisten.

Schritt 1

Markiere alle Verbindungslinien (d.h. Rollen), an denen das Kardinalitätsintervall 1:1 steht.

Anmerkung: Eine ausschließliche 1:1-Markierung stellt einen logisch korrekten DB-Entwurf sicher.
Verbindungslinien, die zu Spezialisierungsassoziationstypen führen müssen nicht markiert werden.

Das zusätzliche Markieren aller 0:1-Verbindungen führt zu Performanceverbesserungen.
Bei relationalen DBMS, die fehlende Werte (NULL) zulassen führt das Markieren von 0:1-Verbindungen zu einem optimalen relationalen DB-Entwurf. Bei Implementiereungen die auch optionale Schlüsselkandidaten zulassen führt das fortgesetzte Markieren über 0:1 Verbindungen hinweg zu effizienten DB-Strukturen.

Schritt 2

1. Bilde die Zusammenhangskomponenten (bestehend aus Entitäts- und Assoziationsstypen) bezüglich der markierten Verbindungslinien.
2. Übrigbleibende (d.h. noch außerhalb von Zusammenhangskomponenten plaziert Entitäts- und Assoziationstypen bilden jeweils eine eigene Zusammenhangskomponenten, wenn gilt:
    - Falls ausschließlich alle Mitgliedschaftsintervalle an der Verbindung zwischen einem solchen Entitätstyp und irgendeinem Assoziationstyp den minimalen Wert 0 haben, ist aus diesem Entitätstyp eine eigenständige Zusammenhangskomponente zu bilden, sofern der Entitätstyp kein Repräsentationstyp oder die entsprechenden Entitäten nicht schon in einem anderen Assoziationstyp definiert sind (Kardinalitätsintervall 1:z; z>=1).
    - Repräsentationstypen werden nicht berücksichtigt.
    - Assoziationstypen bilden jeweils (als einziger Inhalt) eine eigene Zusammenhangskomponente.

Schritt 3

Treten innerhalb einer Zusammenhangskomponente Zyklen auf, so sind diese wie folgt zu behandeln:

Anmerkung: Ein Zyklus in einer Zusammenhangskomponente ist eine Folge $\{ET_1, AT_{1,2}, ET_2, AT_{2,3}, ..., ET_n, AT_{n,1}\}$ mit den Eigenschaften:

- In allen Assoziationstypen$_{i,k}$ ($1<=i, k<=n$) wird die eine Rolle von ET_i und die andere Rolle von ET_k gespielt.
  Anmerkung:Zu Untertypen führende Kanten werden behandelt wie gewöhnliche Beziehungen zu Assoziationstypen.
- Alle Verbindungslinien einer Folge sind markiert.

Zur Ableitung von Relationen müssen Zyklen aufgelöst werden:

- Falls innerhalb eines Zyklus 0:1-Markierungen vorhanden sind, werden diese gelöscht;
- falls nur 1:1-Markierungen vorhanden sind, wird eine beliebig festzusetzende Verbindungslinie gelöscht.

Schritt 4

Aus jeder Zusammenhangskomponente wird eine Relation nach folgenden Regeln:

1. Namensgebend für eine Relation ist genau einer der innen liegenden Entitätstypen. Enthält eine Zusammenhangskomponente nur einen Assoziationstyp, so bekommt die abgeleitete Relation dessen Namen.
2. Die Relation enthält je ein Attribut für jeden direkt mit einer Repräsentation versehenen Entitätstypen im Inneren der Zusammenhangskomponente. Ein Entitätstyp wird zusammen mit seinen sämtlichen Untertypen als ein Entitätstyp betrachtet, sofern die Untertypen über keine eigene Repräsentation verfügen.
3. Zusätzlich enthält die Relation für jeden Assoziationstyp im Inneren einer Zusammenhangskomponente noch je ein Attribut für jede Rolle die zu einem innenliegenden Assoziationstyp beiträgt, welche ein Entitätstyp spielt, der außerhalb der Zusammenhangskomponente liegt.
   Für einen im Inneren der Zusammenhangskomponente liegenden Assoziationstyp sind alle

Entitätstypen, zu denen eine nicht markierte Verbindungslinie führt, außerhalb.

4. Zusammenhangskomponenten, die ausschließlich aus genau einem Assoziationstyp bestehen werden in eine eigenständige Relation überführt, die für jede zum Assoziationstyp beitragende Rolle ein Attribut enthält.
   Dieses Attribut wird mit der Repräsentation des rollenspielenden Entitätstypen typisiert.
5. Jedes aus einem Entitätstyp im Inneren einer Zusammenhangskomponente abgeleitete Attribut ist Schlüsselkandidat.
   Außerdem sind alle diejenigen Attribute Schlüsselkandidaten, deren entsprechende Kardinalitätsintervalle das Maximum 1 besitzen.
6. In den restlichen Fällen sind alle Attribute zusammen Schlüsselkandidat.
7. Existiert in einer Relation mehr als genau ein Schlüsselkandidat, so ist einer unter diesen als Primärschlüssel auszuzeichnen.
8. Jeder Untertyp erbt alle Rollen, die sein Obertyp innerhalb von Assoziationstypen spielt. Zusätzlich erbt er auch die vorhandene Repräsentation des Obertyps.
9. Relationen, deren Attribute sich aus jeweils gleichen Rollen ableiten, werden durch eine einzige Relation dargestellt.
   Treten in einer Zusammenhangskomponente gleiche Rollen eines Entitätstyps mehrfach auf, so werden sie in einer entsprechenden Relational als genau ein Attribut übernommen.
10. Manuelles Eingreifen:
    bei `0:1`-Markierung ist u.U. eine Entscheidung, orientiert an der modellierten Semantik, zu treffen:
    Folgende Konstellationen können das Rückgängigmachen von Markierungen innerhalb einer Zusammenhangskomponente notwendig werden lassen:
    mehrere Schlüsselkandidaten und:
    - alle Schlüsselkandidaten sind optional
    - nicht alle verpflichtend und die Notwendigkeit vorhanden, einen bestimmten als Primärschlüssel festzulegen.

Schritt 5

1. Leiten sich aus einem Entitätstyp mehrere sich entsprechende Attribute ab, so sind die folgenden Abhängigkeiten (Fremdschlüsselbeziehungen) zu berücksichtigen:
   - Ist eine Attributkombination Schlüsselkandidat, so sind zu den entsprechenden Attributkombinationen, die als Primärschlüssel ausgewählt wurden, Fremdschlüsselbeziehungen vorzusehen.
     Hinweis: Fremdschlüsselbeziehungen bedeuten zusätzliche Zugriffe und sollten daher in der Datenbank entsprechend durch Indexstrukturen unterstützt werden.
   - Beim Auftreten identischer Schlüsselkandidaten sind ebenfalls Fremdschlüsselbeziehungen vorzusehen.
2. Metainformationen, die nicht die DB-Strukturebene betreffen, sondern Ausprägungen einschränken, werden den entsprechenden DB-Strukturelementen zugeordnet (z.B. Domäneneinschränkungen bei Attributen).

Schritt 6

Ist ein Entitätstyp Spezialisierung (d.h. Untertyp) eines anderen, so wird in die Relation die aus der Zusammenhangskomponente gebildet wurde, welche den Untertypen beinhaltet die Primärschlüsselattribute derjenigen Relation übernommen, die den Obertypen beinhaltet.

Anmerkung: Schlüsselkandidaten dieser Relation werden identisch zu den anderen Relationen ermittelt.

Schritt 7

1. Systemunabhängig
   - Alle Attribute bekommen als Datentyp den Entitätstyp, durch den sie repräsentiert werden.
   - Bei `1:1`-Markierung wird für alle Attribute der Relation ein `NOT NULL` vergeben.
   - Ein Primärschlüssel muß stets mit `NOT NULL` vereinbart werden.
   - Bei markierten `0:1`-Beziehungen: Alle aus über `0:1`-Beziehungen angebundenen Entitätstypen entstehenden

Attribute werden auf NULL gesetzt.
- Schlüsselkandidaten und Zugriffspfade werden als Indexe angelegt.
- Soweit für Metainformation formale Umsetzungsmöglichkeiten existieren, werden die entsprechenden konsistenzgarantierenden Einschränkungen formuliert.

2. Systemabhängig
   Die Syntax für die physische Realisierung der Relationen (Tabellen) und der Indexstrukturen sowie der Datentypen und Einschränkungen (soweit unterstützt) müssen dem jeweiligen DBMS angepaßt werden.
   Evtl. durch das DBMS automatisch angelegte Indexstrukturen müssen nicht mehr explizit formuliert werden.

2.3 Algebraischer Entwurf mit der Normalformentheorie

Neben dem graphischen Entwurf logischer DB-Strukturen genießt der algebraische Entwurf auf Basis der sog. *Normalformentheorie* in Theorie und Praxis große Bedeutung.
Historisch gesehen stellt die Betrachtung von relationalen Strukturen mit Hilfe mathematischer Methoden die älteste Disziplin dar und findet sich heute in allen bedeutenden Lehrbüchern.
Dieser Abschnitt führt in die sechs verschiedenen Normalformen hinsichtlich ihrer Definition sowie ihrer Implikationen auf die Struktur des logischen Modells ein und zieht Parallelen zur Vorgehensweise des eingeführten Algorithmus zur Umsetzung konzeptueller Strukturen des E^3R-Modells.

Ebenso wie der Algorithmus zur Transformation eines E^3R-Schemas führt auch die Normalisierung zu einem anomalienfreien relationalen Datenbankentwurf. Voraussetzung der Anomaliefreiheit ist die konsequente Ermittlung und Eliminierung von Redundanz, d.h. keine Information darf in der Datenbank mehrfach vorhanden sein.

Insgesamt verfolgt der Normalisierungsprozeß folgende Ziele:

- Redundanzvermeidung als Basis der Anomaliefreiheit
- Vermeidung unnötiger Abhängigkeiten, die Performanceeinbußen bei Einfüge-, Lösch- und Änderungsoperationen nach sich ziehen
- Senkung der Anzahl beteiligter Tabellen bei der Modifikation der Datenbank
- Erhöhung des Dokumentationsgrades des entstehenden Datenmodells (Ziel: Verständlichkeit)

Die Anomalienfreiheit ist die zentrale Basisforderung und Zielsetzung des Normalisierungsprozesses. Im Detail werden drei Ausprägungen unterschiedlicher Anomalien unterschieden:

- **Einfügeanomalie: Durch das Hinzufügen eines korrekten neuen Tupels werden konsistent vorliegende Daten in einen inkonsistenten Zustand überführt.**
- **Löschanomalie: Durch die Entfernung eines Tupels entsteht ein inkonsistenter Datenbestand.**
- **Aktualisierungsanomalie: Durch die Änderung eines vorhandenen Tupels entsteht ein inkonsistenter Datenbestand.**

Alle Arten von Anomalien gehen auf das Vorhandensein von Redundanz, mithin einem Verstoß gegen die Grundregel jede im konzeptuellen Schema modellierte Information an nur genau einer Stelle abzuspeichern, zurück.

Ausgangssituation des Normalisierungsvorganges ist die *Urrelation* die alle Attribute in genau einer Relation zusammenfaßt.

Erste Normalform

Definition 32: *Erste Normalform (1NF)*
Eine Relation ist dann in erster Normalform, wenn ihre Domänen (=Wertausprägungen der Attribute) nur einfache (atomare) Werte besitzen.

Atomarer Wert bedeutet hierbei, daß kein Attributinhalt strukturiert sein darf, d.h. durch mögliche Zerlegungsoperationen in kleine eigenständige Informationseinheiten zerlegt werden kann.

Beispiel einer Relation die **nicht in 1NF** ist:

FNAME	LNAME	ADDRESS	BDATE
John	Smith	731 Fondren, Houston, TX	1965-01-09
Franklin	Wong	638 Voss, Houston, TX	1955-12-08
Joyce			1972-07-31
Polly Esther	Wallace	291 Berry, Bellaire, TX	1941-06-20, 1952-09-04

Beispiel 19: Relation, die nicht in 1NF ist

Ziel der Überführung in 1NF ist es, Relationen zu erhalten, die in gängigen RDBMS abspeicherbar sind. Diese bieten zwar heute technische Mechanismen (wie Array- und Referenztypen) an, die Strukturen ähnlich den dargestellten verwaltbar werden lassen. Voraussetzung ihrer konzeptionellen Beherrschung ist jedoch die vorherige Normalisierung.

Im Beispiel befinden sich die Zeilen von `Franklin` und `Joyce Wong` nicht in 1NF, da sie nicht für jedes Attribut einen Wert besitzen, sondern sich eine Wertausprägung (`Wong` und `638 Voss, Houston, TX`) teilen.
Ebenso befindet sich der Eintrag von `Polly Esther Wallace` nicht in 1NF, da hier für das Geburtsdatum unerlaubterweise zwei Einträge auftreten.
Die beiden Vornamen sind im Rahmen der Semantik des Attributes `FNAME` zugelassen und daher im Normalisierungsprozeß nicht zu beanstanden.

Die Relation aus Beispiel 19 in erster Normalform:

FNAME	LNAME	ADDRESS	BDATE
John	Smith	731 Fondren, Houston, TX	1965-01-09
Franklin	Wong	638 Voss, Houston, TX	1955-12-08
Joyce	Wong	638 Voss, Houston, TX	1972-07-31
Polly Esther	Wallace	291 Berry, Bellaire, TX	1941-06-20
Polly Esther	Wallace	291 Berry, Bellaire, TX	1952-09-04

Beispiel 20: Relation, die in 1NF ist

Zur Umformung der Relation in eine Relation in erster Normalform wurden die „gemeinsamen“ Attribute aufgelöst, so das nunmehr jedes Attribut genau einem Tabelleintrag (Tupel) zugeordnet ist.
Zusätzlich wurde für jedes Attribut die atomare Belegung sichergestellt.

Die Einführung der ersten Normalform verhindert damit die Bildung *geschachtelter Relationen*, die entstünden, jedes Attribut eine Menge anderer Attribute, mithin wiederum eine vollständige Relation, enthalten könnte.
Anmerkung: Diese Forderung wird durch die in SQL:1999 definierten `ARRAY`-Typen aufgeweicht und für postrelationale und objektorientierte Datenbanken vollständig aufgegeben, weshalb diese Strukturen auch als *Non-First-Normal-Form* (kurz: NFNF, NF2 oder NF^2) bezeichnet werden.

Test auf Einhaltung der ersten Normalform:
Die Relation sollte keine nicht-atomaren Attribute oder verschachtelte Relationen enthalten.

Der algorithmische Ableitungsprozeß aus dem konzeptuellen Schema stellt durch die Organisation der Repräsentationstypen sicher, daß Attribute ausschließlich durch atomare Werte repräsentiert werden.

Zweite Normalform

Grundlegende Voraussetzung zum Verständnis der zweiten Normalform ist das Konzept der *vollen funktionalen Abhängigkeit*.

Definition 33: *Volle funktionale Abhängigkeit*
Ein Attribut *y* einer Relation ist vollfunktional abhängig von einem Attribut *x* wenn gilt, daß jede Ausprägung von *x* genau eine Ausprägung von *y* bestimmt und *y* nicht abhängig von Teilattributen von *x* ist.
Im Zeichen: *x->y*.
Die vollfunktionale Abhängigkeit wird häufig als *FD* (*functional dependency*) abgekürzt.

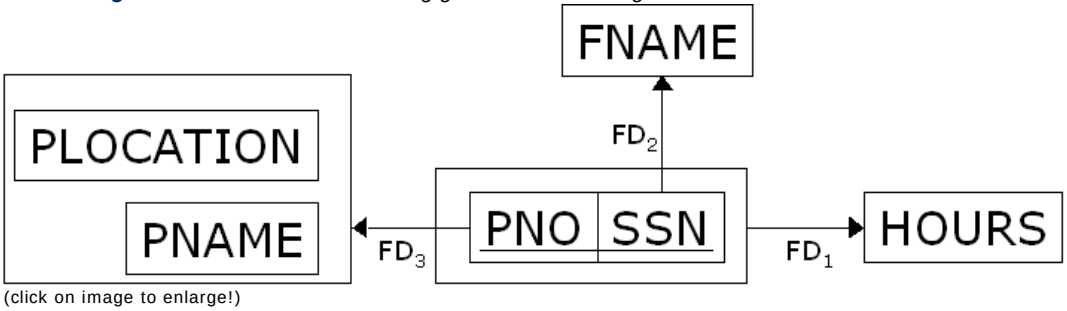
Bemerkung: Man beachte, daß der Begriff *Attribut* in Definition 33 eine nichtleere Menge von Attributen bezeichnet.

Beispiel aus der Demodatenbank:

SSN	PNO	HOURS	FNAME	DNAME	PNAME	PLOCATION
123456789	1	32.5	John	Research	ProductX	Bellaire
123456789	2	7.5	John	Research	ProductY	Sugarland
333445555	2	10.0	Franklin	Research	ProductY	Sugarland
333445555	3	10.0	Franklin	Research	ProductZ	Houston
333445555	10	10.0	Franklin	Research	Computerization	Stafford
333445555	20	10.0	Franklin	Research	Reorganization	Houston
453453453	1	20.0	Joyce	Research	ProductX	Bellaire
453453453	2	20.0	Joyce	Research	ProductY	Sugarland
666884444	3	40.0	Ramesh	Research	ProductZ	Houston
888665555	20	NULL	James	Headquarters	Reorganization	Houston
987654321	20	15.0	Jennifer	Administration	Reorganization	Houston
987654321	30	20.0	Jennifer	Administration	Newbenefits	Stafford
987987987	10	35.0	Ahmad	Administration	Computerization	Stafford
987987987	30	5.0	Ahmad	Administration	Newbenefits	Stafford
999887777	10	10.0	Alicia	Administration	Computerization	Stafford
999887777	30	30.0	Alicia	Administration	Newbenefits	Stafford

In der Relation existieren folgende voll funktionale Abhängigkeiten:

Abbildung 17: *Voll funktionale Abhängigkeiten in der dargestellten Relation*



(click on image to enlarge!)

Es ist offensichtlich, daß zwar HOURS vom vollständigen Primärschlüssel (gebildet aus PNO gemeinsam mit SSN) abhängen (FD_1), aber der Name (FNAME) und die Kombination aus PLOCATION und PNAME nur von SSN bzw. PNO und damit Teilen des Primärschlüssels abhängen (FD_2 bzw. FD_3).

Inhaltlich manifestieren sich diese Probleme im Zwang verschiedene Daten (etwa SSN und FNAME) wiederholt (redundant) abspeichern zu müssen. Ändert sich eine dieser Angaben, so muß potentiell eine große Anzahl Tupel in der Datenbank aktualisiert werden. Werden hierbei nicht

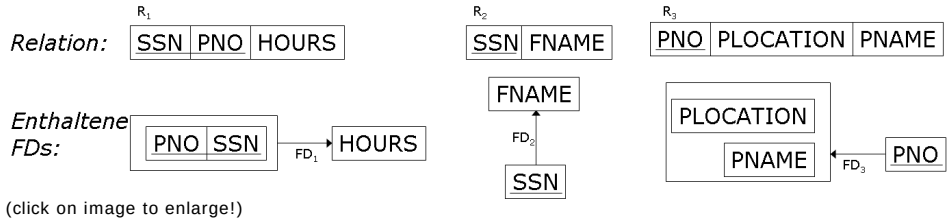
alle Datensätze aktualisiert so entsteht ein inkonsistenter Datenbestand.
Zusätzlich ist es nicht möglich bestimmte Informationszusammenhänge abzubilden. Hierunter fällt beispielsweise der Wunsch Projekte (etwa: PLOCATION und PNAME) zur verwalten, denen noch keinen Mitarbeiter zugeordnet ist.

Um eine Relation in 2NF zu überführen muß sie so zerlegt werden, daß alle Attribute der neu entstehenden Relation vom selben Schlüsselkandidaten abhängen.

Definition 34: *Zweite Normalform (2NF)*
Eine Relation ist in 2NF genau dann, wenn sie in 1NF ist und jedes Nichtschlüsselattribut voll funktional abhängig von einem Schlüsselkandidaten ist.

Entstehende Relationen:

Abbildung 18: *Relationen in 2NF*

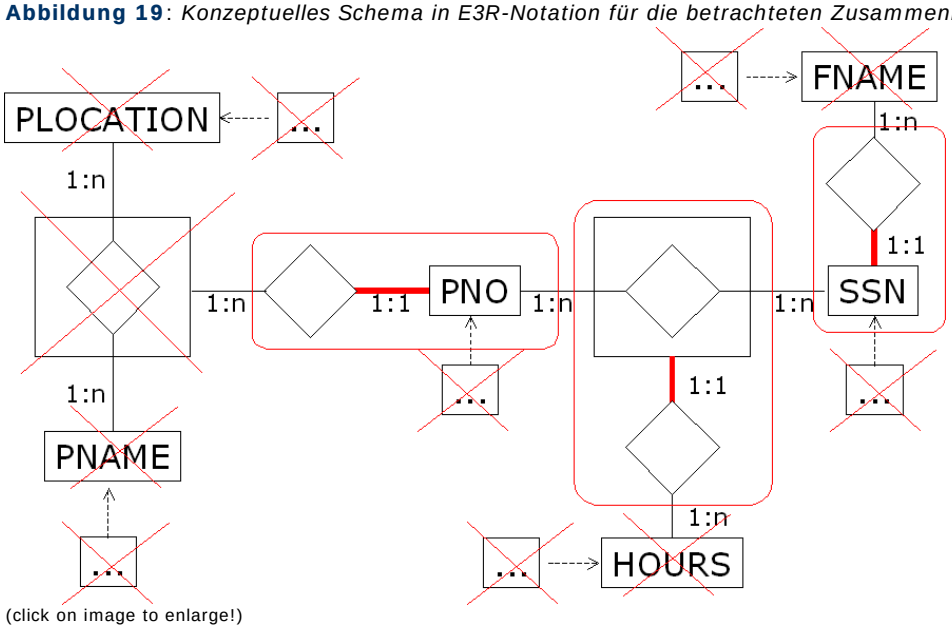


(click on image to enlarge!)

Test auf Einhaltung der zweiten Normalform:
In Relationen deren Primärschlüssel mehrere Attribute enthalten, sollte kein Nichtschlüsselattribut voll funktional von einem Teil des Primärschlüssels abhängen.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E^{3}R-Notation gewährleistet automatisch die Erzeugung von Relationen in 2NF:

Abbildung 19: *Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Dritte Normalform (3NF)

Die dritte Normalform erweitert die für die Zweite getroffenen Aussagen dahingehend, daß zusätzlich zur voll funktionalen Abhängigkeit die *transitive Abhängigkeit* eingeführt und betrachtet wird.

Definition 35: *Transitive Abhängigkeit*
In einer Relation *R* ist ein Attribut *z* transitiv von einem Attribut *x* abhängig dann und nur dann, wenn *z* voll funktional von *y* und *y* voll funktional von *x* abhängig ist.
Im Zeichen: x->->z

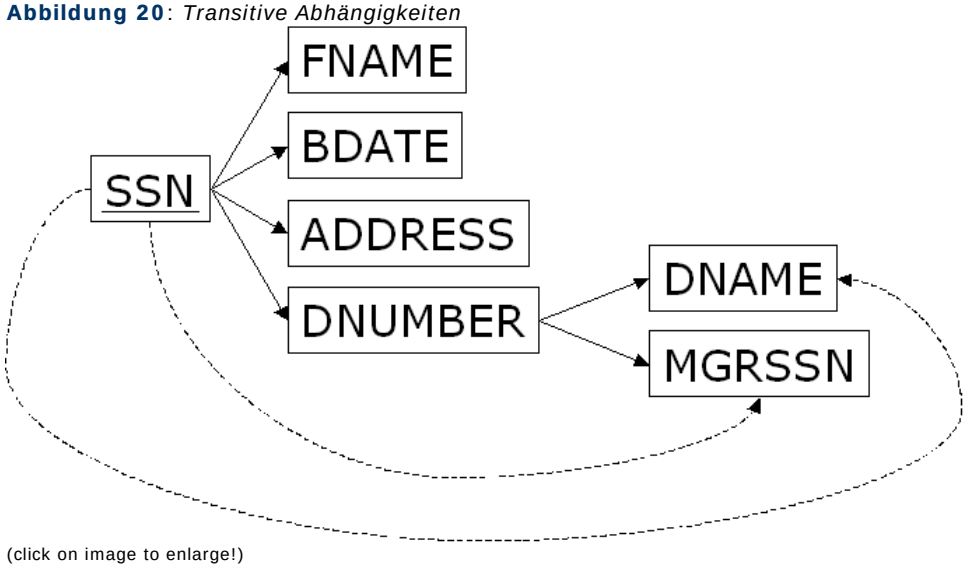
Bemerkung: Man beachte, daß der Begriff *Attribut* in Definition 35 eine nichtleere Menge von Attributen bezeichnet.

Im Beispiel der Demodatenbank:

```
+----------+-----------+------------+--------------------------+---------
+---------------+-----------+
| FNAME    | SSN       | BDATE      | ADDRESS                  | DNUMBER |
DNAME          | MGRSSN    |
+----------+-----------+------------+--------------------------+---------
+---------------+-----------+
| James    | 888665555 | 1937-11-10 | 450 Stone, Houston, TX   |       1 |
Headquarters   | 888665555 |
| Jennifer | 987654321 | 1941-06-20 | 291 Berry, Bellaire, TX  |       4 |
Administation | 987654321 |
| Ahmad    | 987987987 | 1969-03-29 | 980 Dallas, Houston, TX  |       4 |
Administation | 987654321 |
| Alicia   | 999887777 | 1968-07-19 | 3321 Castle, Spring, TX  |       4 |
Administation | 987654321 |
| John     | 123456789 | 1965-01-09 | 731 Fondren, Houston, TX |       5 |
Research       | 333445555 |
| Franklin | 333445555 | 1955-12-08 | 638 Voss, Houston, TX    |       5 |
Research       | 333445555 |
| Joyce    | 453453453 | 1972-07-31 | 5631 Rice, Houston, TX   |       5 |
Research       | 333445555 |
| Ramesh   | 666884444 | 1962-09-15 | 975 Fire Oak, Humble, TX |       5 |
Research       | 333445555 |
+----------+-----------+------------+--------------------------+---------
+---------------+-----------+
```

In der Relation existieren folgende direkten und transitive Abhängigkeiten (die direkten funktionalen Abhängigkeiten sind durch gerichtete Kanten mit durchgezogener Linienführung, die Transitiven durch unterbroche Linienführung dargestellt):

Abbildung 20: *Transitive Abhängigkeiten*



(click on image to enlarge!)

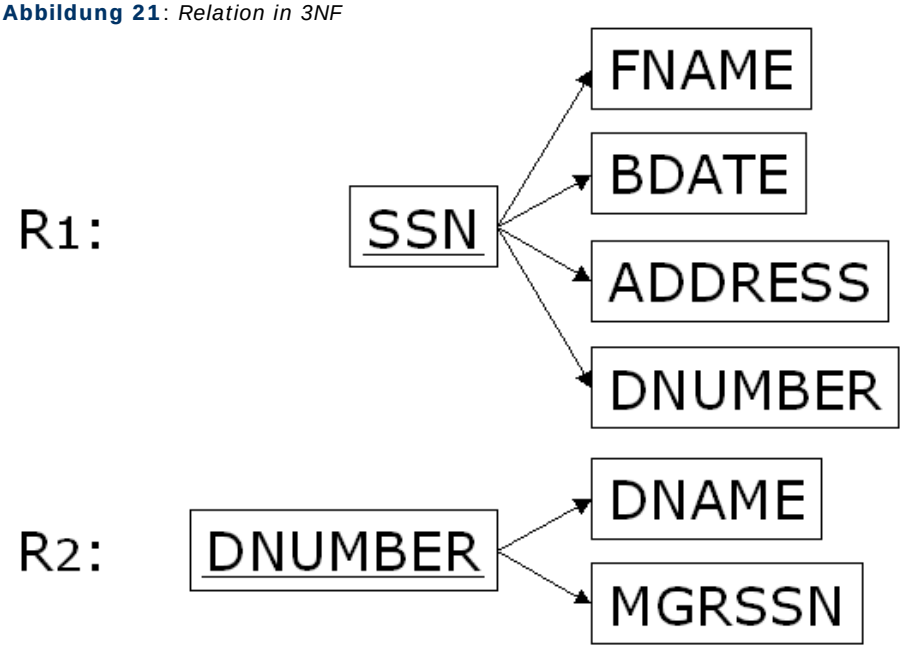
In dieser Organisationsform tritt eine Einfügeanomalie auf, wenn ein Mitarbeiter neu eingefügt wird und dabei „falsche“ (d.h. inkonsistente) Werte für die Abteilung angelegt werden. Zusätzlich fällt das Einfügen neuer Abteilungen, zu denen (noch) kein Mitarbeiter abgespeichert wird, schwer, da SSN zwingend anzugeben ist (NULL-Wert ist wegen der Definition als Primärschlüssel nicht zugelassen!).
Daneben existiert eine Löschanomalie dahingehend, daß wenn der letzte Mitarbeiter einer Abteilung aus der Datenbank entfernt wird auch alle Informationen über diese Abteilung verloren gehen.
Werden Abteilungsdaten verändert, so kann es zu einer Modifikationsanomalie kommen, da nicht in jedem Falle eindeutig klärbar ist ob nur die Abteilungsdaten dieses Mitarbeiters verändert werden sollen (beispielsweise im Falle eines Abteilungswechsels) oder die Daten aller abgespeicherten Abteilungen (beispielsweise im Falle der Umbenennung einer Abteilung).

Die dritte Normalform setzt sich zum Ziel die Ursachen dieser Anomalien zu beseitigen:

Definition 36: *Dritte Normalform (3NF)*
Eine Relation ist dann und nur dann in 3NF, wenn sie in 2NF ist und jedes Nicht-Schlüsselattribut nicht-transitiv abhängig ist von einem Schlüsselkandidaten; sie ist auch in 3NF, wenn sich eine transitive Abhängigkeit ausschließlich über Bestandteile des Schlüsselkandidaten herleiten läßt.

Gemäß dieser Definition ist die Beispielrelation zu zerlegen in:
R_1(SSN, FNAME, BDATE, ADDRESS, DNUMBER) und
R_2(DNUMBER, DNAME, MGRSSN).

Abbildung 21: *Relation in 3NF*

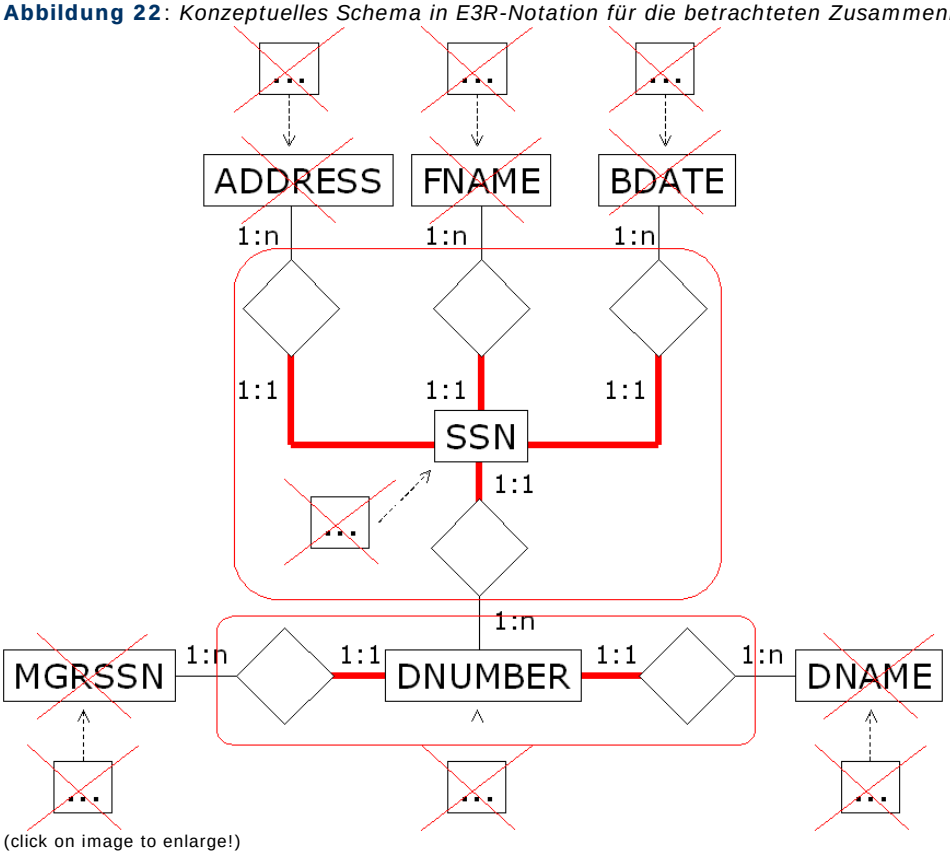


(click on image to enlarge!)

Test auf Einhaltung der dritten Normalform:
Eine Relation sollte kein Nicht-Schlüsselattribut enthalten, das voll funktional von einem anderen Nicht-Schlüsselattribut (oder von einer Menge von Nichtschlüsselattributen) abhängig ist. Das heißt, es sollte keine transitive Abhängigkeit eines Nichtschlüsselattributs vom Primärschlüssel bestehen.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E^{3}R-Notation gewährleistet automatisch die Erzeugung von Relationen in 3NF:

Abbildung 22: *Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Das Rissanen-Theorem klärt die sich prinzipiell ergebene Fragestellung warum die in Abbildung 21 dargestellte Zerlegung gewählt wurde, da sich prinzipiell unter Ausnutzung der transitiven Abhängigkeiten auch eine (Alternativ-)Zerlegung in R_1(SSN, FNAME, BDATE, ADDRESS, DNUMBER) und R_2(SSN, DNAME, MGRSSN) angeboten hätte.

Nach dem Theorem von Heath und Rissanen ist jedoch die in Abbildung 21 gewählte Zerlegung die einzig korrekt mögliche, da sie die tatsächlich vorhandenen funktionalen Abhängigkeiten erhält, was bei obiger Alternative nicht der Fall wäre.

Angewendet auf unser Beispiel bedeutet dies, daß für obige (nach Heath/Rissenan fehlerhafte) Zerlegung beispielsweise der Anwendungsfall nach Anlage einer Abteilung (DNUMER gemeinsam mit ihrem Namen (DNAME) und der SSN ihres Abteilungsleiters (MGRSSN) nicht möglich wäre, da DNAME und MGRSSN nur verwaltet werden können, wenn gleichzeitig die als Primärschlüssel zwingend anzugebende SSN eines Mitarbeiters dieser Abteilung existiert. Mithin wäre die Speicherung einer Abteilung die nur über ihren Leider aber (noch) nicht über Mitarbeiter verfügt nicht möglich.
Die gewählte Zerlegung vermeidet jedoch diese Einschränkung und liefert, ebenso wie der Abteilungsalgorithmus aus dem konzeptuellen Schema, das korrekte Resultat.

Boyce/Codd-Normalform (BCNF)

Ursprünglich wurde die *Boyce/Codd Normalform* (BCNF) als Vereinfachung der dritten Normalform vorgeschlagen. Jedoch faßt sie diese schärfer und führt so zu einem neuen Typ von Normalform, der nach ihren Schöpfern Boyce und Codd benannt wurde.
Inhaltlich räumt sie mögliche Anomalien aus, die in Relationen, welche sich bereits in 3NF befinden, noch auftreten können.

Definition 37: *Boyce/Codd-Normalform*
In einer Relation ist dann und nur dann in BCNF, wenn sie in 3NF ist und gleichzeitig jede Determinante Schlüsselkandidat ist.

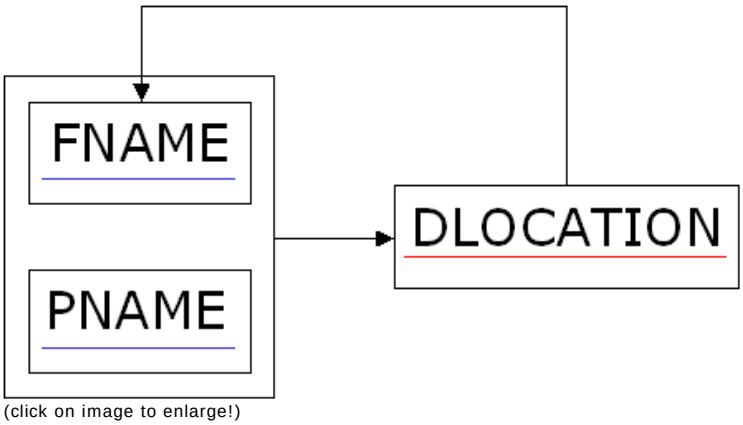
Hierbei definiert die BCNF den Begriff der *Determinante* als den Ausgangspunkt einer funktionalen Abhängigkeit.

Im Beispiel der Demodatenbank:

FNAME	PNAME	DLOCATION
Franklin	ProductY	Sugarland
Franklin	ProductZ	Houston
Jennifer	Newbenefits	Stafford
...	...	...

In der Relation existieren folgende funktionale Abhängigkeiten:

Abbildung 23: *Funktionale Abhängigkeiten*



(click on image to enlarge!)

Die Schlüsselkandidaten sind durch farbliche Unterstreichung hervorgehoben. Hierbei bestimmt FNAME gemeinsam mit PNAME eindeutig einen Wert für DLOCATION (blau) und DLOCATION (rot) bestimmt eindeutig einen Wert für FNAME.

Da sich die transitive Abhängigkeit PNAME->DLOCATION->FNAME ausschließlich über Schlüsselkandidaten herleitet ist die Relation in 3NF.
Dennoch ist sie nicht anomalienfrei, da sie beispielsweise die Ablage von Abteilungsstandorten (DLOCATION) und den Namen der Abteilungsleiter (FNAME) verbietet, wenn kein Projektname (PNAME) zusätzlich abgespeichert wird.

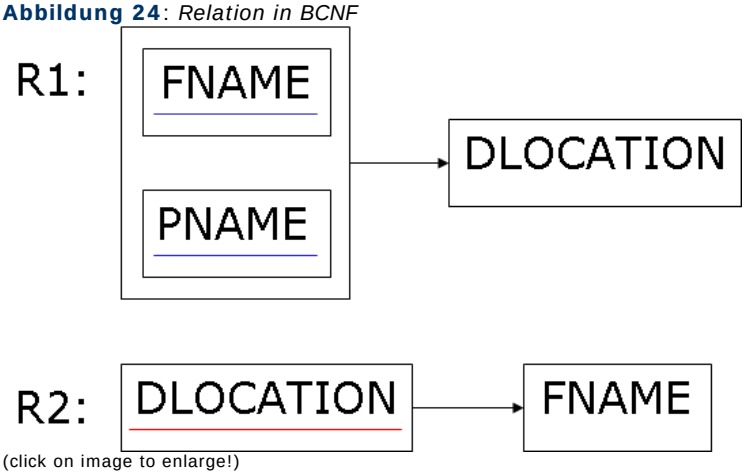
Ebenso ist die Ablage von Projekten und den zuständigen Abteilungen erst dann möglich, wenn zusätzlich der Name des Abteilungsleiters bekannt ist.

Dieses Manko wird durch die Forderung der BCNF behoben. Sie erzwingt die Zerlegung der abgebildeten Ausgangsrelation in zwei Eigenständige. Hierbei wird jede Determinante der Ausgangsrelation zum Schlüsselkandidaten in den neu gebildeten Relationen.

Gemäß Definition 37 ist die Beispielrelation zu zerlegen in:

R_1(FNAME, PNAME, DLOCATION)
R_2(DLOCATION, FNAME).

Abbildung 24: *Relation in BCNF*

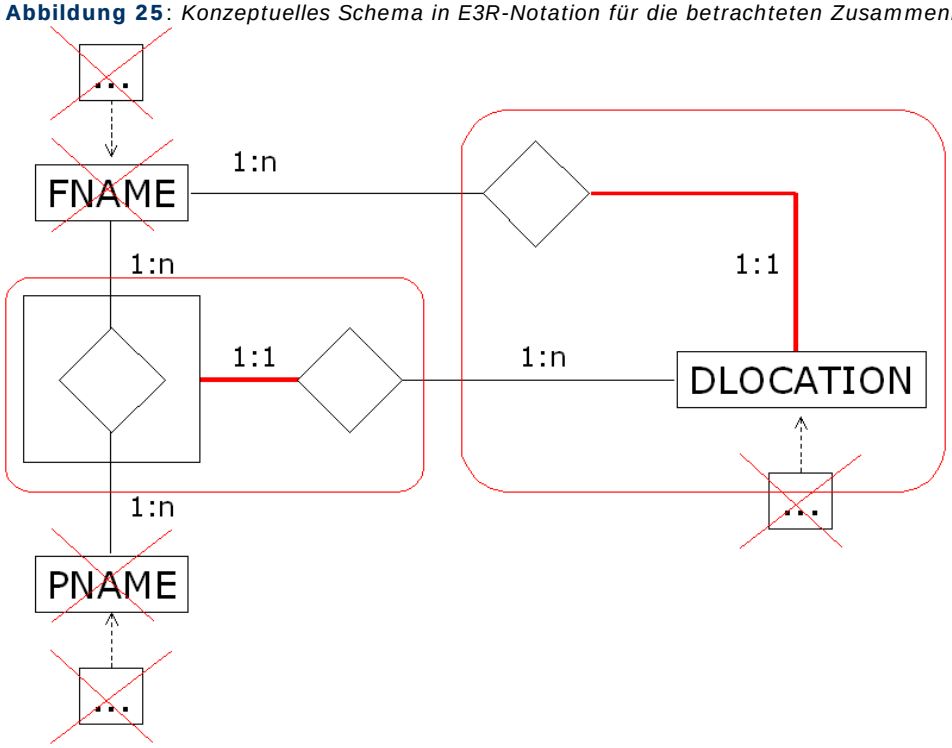


(click on image to enlarge!)

Test auf Einhaltung der Boyce/Codd-Normalform:
Eine Relation sollte lediglich direkt vom Schlüssel abhängende Attribute enthalten.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E^{3}R-Notation gewährleistet automatisch die Erzeugung von Relationen in BCNF:

Abbildung 25: *Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Vierte Normalform (4NF)

Die bisher betrachteten Normalformen nutzen alle das Vorhandensein funktionaler Abhängigkeiten aus. Jedoch existieren neben diesem Beziehungstyp auch noch andere, im vorhergehenden nicht berücksichtige, Abhängigkeit.

Definition 38: *Mehrwertige Abhängigkeit*
Eine mehrwertige Abhängigkeit (*multivalue dependency*, MVD) ist eine Abhängigkeit, die einem

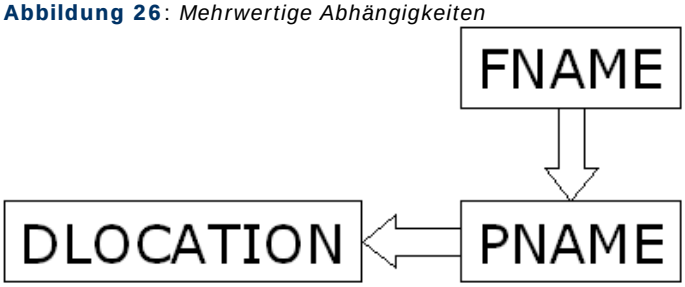
einem Attribut eine Menge verschiedener Werte zuordnet.

Das Vorhandensein mehrwertiger Abhängigkeiten in einer Relation kann zu Aktualisierungsanomalien führen, wie die Betrachtung des bekannten Beispiels aus der Demodatenbank:

FNAME	PNAME	DLOCATION
Franklin	ProductY	Sugarland
Franklin	ProductZ	Houston
Jennifer	Newbenefits	Stafford
...	...	...

In der Relation existieren folgende mehrwertige Abhängigkeiten:

Abbildung 26: *Mehrwertige Abhängigkeiten*



(click on image to enlarge!)

Die Existenz der dargestellten mehrwertigen Abhängigkeiten in der Datenbank führt dazu, daß dieselbe Information mehrfach abgespeichert werden muß. So enthält die Beispieldatenbank mehrfach denselben FNAME sowie wiederholt denselben Produktnamen (PNAME).
Als Konsequenz dieser Wiederholung müssen bei jedem Aktualisierungsvorgang, der die Zuordnung des Produktes zum FNAME betrifft alle FNAME-Einträge geändert werden, jedoch bei der Zuständigkeitsänderung eines Produktverantwortlichen nur der betroffene FNAME.

Ziel der vierten Normalform ist die Elimination von Redundanzen, die aus mehrwertigen Abhängigkeiten herrühren. Allerdings können mehrwertige Abhängigkeiten nicht vollständig entfernt werden, daher ist für die Normalisierung gemäß 4NF die Eliminierung aller *nicht trivialen* mehrwertigen Abhängigkeiten als Zielsetzung definiert.

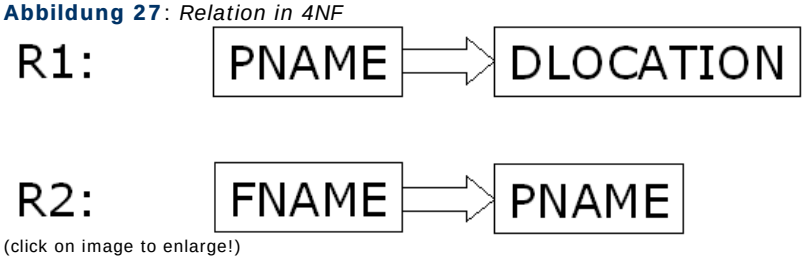
Eine *triviale mehrwertige Abhängigkeit* ist hierbei festgelegt als:

Definition 39: *Triviale mehrwertige Abhängigkeit*
Eine triviale mehrwertige Abhängigkeit ist eine mehrwertige Abhängigkeit zwischen Attributmengen X und Y der Relation R für die gilt: Y ist eine Teilmenge von X oder die Vereinigung von X und Y bildet R.

Definition 40: *Vierte Normalform*
Eine Relation ist dann in vierter Normalform, wenn sie nur noch triviale mehrwertige Abhängigkeiten enthält.

Gemäß dieser Definition ist die Beispielrelation zu zerlegen in:
R_1(DLOCATION, PNAME) und
R_2(FNAME, PNAME).

Abbildung 27: *Relation in 4NF*

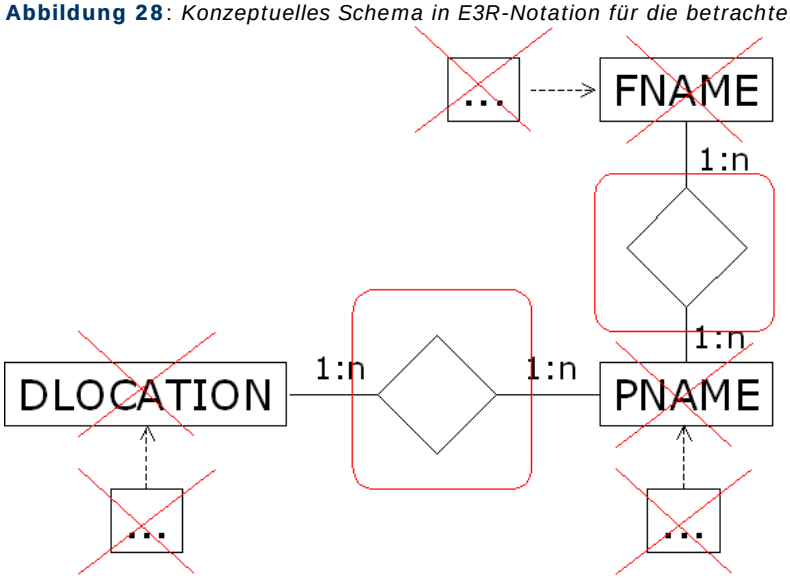


(click on image to enlarge!)

Test auf Einhaltung der vierten Normalform:
Eine Relation sollte keine nicht trivialen mehrwertigen Abhängigkeiten enthalten.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E^{3}R-Notation gewährleistet automatisch die Erzeugung von Relationen in 4NF:

Abbildung 28: *Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Fünfte Normalform (5NF)

Die fünfte Normalform greift anders als alle vorhergehenden nicht auf die Zusammenhänge der Typebene zurück, sondern benötigt zu ihrer Untersuchung die Betrachtung von tatsächlichen Datenbankinhalten.
Ausgehend von diesen definiert sie die fünfte Normalform als:

Definition 41: *Fünfte Normalform*
Eine Relation ist dann in fünfter Normalform (5NF), bei ihrer Zerlegung durch Projektionen und deren anschließender Kombination durch Verbundoperationen keine Tupel gebildet werden, die nicht Bestandteil der Ausgangsrelation waren.

Aufgrund ihrer Abstützung auf die Projektions- und Verbundoperation (engl. *join*) wird die 5NF auch als *Project Join Normalform* (PJNF) bezeichnet.

Im Beispiel der Demodatenbank:

Relation `Ur`:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

Durch Projektion ergeben sich die drei möglichen Relationen:

```
SELECT Ur.FNAME, Ur.DNUMBER INTO R11 FROM Ur
```

R_{11}:

FNAME	DNUMBER
John	5
John	5
James	1

```
SELECT Ur.FNAME, Ur.DLOCATION INTO R12 FROM Ur
```

R_{12}:

FNAME	DLOCATION
John	Bellaire
John	Houston

James	Houston

```
SELECT Ur.DNUMBER, Ur.DLOCATION INTO R13 FROM Ur
```

R_{13}:

DNUMBER	DLOCATION
5	Bellaire
5	Houston
1	Houston

Durch Verbundoperationen entstehen die Relationen:

```
SELECT R11.FNAME, R11.DNUMBER, R12.DLOCATION INTO R21 FROM R11 INNER JOIN R12 ON
R11.FNAME=R12.FNAME
```

R_{21}:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

```
SELECT R11.FNAME, R11.DNUMBER, R13.DLOCATION INTO R22 FROM R11 INNER JOIN R13 ON
R11.DNUMBER = R13.DNUMBER
```

R_{22}:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

```
SELECT R12.FNAME, R12.DLOCATION, R13.DNUMBER INTO R23 FROM R12 INNER JOIN R13 ON
R12.DLOCATION=R13.DLOCATION
```

R_{23}:

FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston
James	5	Houston
John	1	Houston

Trotz der ausschließlich durch Rekombination der zuvor aus der Urrelation gebildeten Projektionen (d.h. ohne Hinzunahme neuer) Daten „entstehten“ in Relation R_{23} (rot hervorgehobene) Tupel (sog. *spurious tupel*), die in dieser Form nicht in der Ausgangsrelation präsent waren.

Erst das Zusammenfügen aller aus Verbundoperationen erzeugten Relationen durch einen erneuten Verbund eliminiert diesen *unechten Tupel*:

```
SELECT R21.FNAME, R22.DNUMBER, R23.DLOCATION INTO RESULT FROM (R21 INNER JOIN R22
ON (R21.FNAME = R22.FNAME) AND (R21.DNUMBER = R22.DNUMBER) AND (R21.DLOCATION =
R22.DLOCATION)) INNER JOIN R23 ON (R22.FNAME = R23.FNAME) AND (R22.DNUMBER = R23.
DNUMBER) AND (R22.DLOCATION = R23.DLOCATION) AND (R21.FNAME = R23.FNAME) AND (R21.
DNUMBER = R23.DNUMBER) AND (R21.DLOCATION = R23.DLOCATION)
```

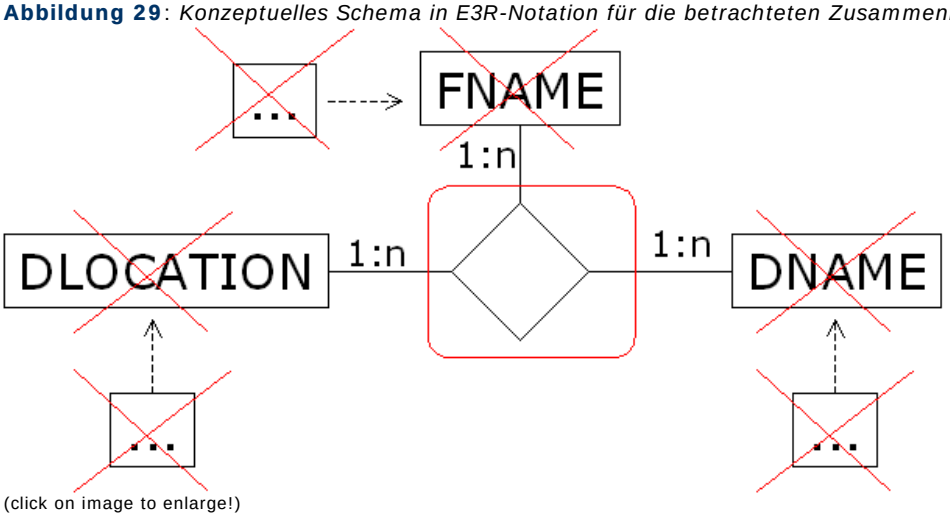
FNAME	DNUMBER	DLOCATION
John	5	Bellaire
John	5	Houston
James	1	Houston

Das Auftreten unechter Tupel ist ein Indiz für eine Relation in der Verbundabhängigkeiten existieren. Eine solche Relation darf nicht weiter zerlegt werden, da durch die Entfernung von Attributen Information verloren ginge.

Test auf Einhaltung der fünften Normalform:
Wenn durch Projektions- und Verbundoperationen keine unechten Tupel entstehen, dann ist die Relation in 5NF.

Der Ableitungsprozeß aus dem konzeptuellen Schema in E^3R-Notation gewährleistet automatisch die Erzeugung von Relationen in 5NF:

Abbildung 29: *Konzeptuelles Schema in E3R-Notation für die betrachteten Zusammenhänge*



(click on image to enlarge!)

Die 5NF ist die höchstmögliche über den Normalisierungsprozeß erreichbare Normalform.
Dies bedeutet jedoch nicht, daß jede Relation bis in 5NF gebracht werden kann. Der Normalisierungsprozeß endet generell mit der höchstmöglichen Normalform, dies muß jedoch nicht immer 5NF sein, sondern orientiert sich an den modellierten Informationszusammenhängen.

Weitere Normalisierungsaspekte

Über die im vorhergehedenden betrachteten formalisierten Normalformen hinaus existieren noch weitere Abhängigkeiten und Normalformen-ähnliche Güteaussagen für Datenbanken.

Inklusionsabhängigkeit:
Die Inklusionsabhängigkeit (engl. *inclusion dependence*, ID) beschreibt die Beziehungen zwischen Super- und Subtypen, insbesondere die *Attributentsprechung*, d.h. die Tatsache, daß der Subtype mindestens alle Attribute des Supertypen aufweist, sowie die *Kompatibilitätsrelation* worunter die Austauschbarkeit von Ausprägungen der beiden Typen verstanden wird für Anwendungsbereiche die lediglich auf die gemeinsam vorhandenen Attribute zugreifen.
aus der Inklusionsabhängigkeit folgen u.a. die drei Inferenzregeln:
$IDIR_1$: (Reflexivität) Die Inklusionsabhängigkeit ist reflexiv.
$IDIR_2$: (Attributentsprechung) Verfügen zwei Typen über dieselben Attribute, dann entsprechen sie sich.
$IDIR_3$: (Transitivität) Ist *B* Untertyp von *A* und *C* Untertyp von *B*, dann ist auch *C* Untertyp von *A*.
Trotz dieser formalisierbaren Aussagen und der breiten Verwendung von Modellierungskonstrukten zur Darstellung von Spezialisierungsbeziehungen wurden auf Basis der Inklusionsabhängigkeit bisher noch keine Normalformen vorgeschlagen.

Template-Abhängigkeiten:
Die Idee der Template-Abhängigkeit fußt auf der Definition einer Reihe von *Hypothesetupeln* und daraus abgeleiteten *Konklusionstupeln*, welche gültige Ausprägungen der Datenbank abstrahiert beispielhaft aufzeigen.
Template-Abhängigkeiten gestatten die einfach Formulierung von Intrarelationsabhängigkeiten die sich auf konkrete Wertausprägungen einzelner Attribute beziehen.
Das Beispiel zeigt die Abhängigkeit, daß kein Angestellter mit mehr Einkommen ausgestattet sein darf als sein Vorgesetzter:

```
EMPLOYEE(FNAME, SSN ..., SALARY, SUPERSSN)
             a     b ...    c     NULL
Hypothese    e     f ...    g     b
-------------------------------------
Konklusion       c < g
```

Domain-Key-Normalform (DKNF):
Grundannahme der Domain-Key-Normalform (DKNF) ist es, daß wenn für jedes Attribut einer Relation eine Domäne (d.h. die Menge der zugelassenen Wertbelegungen) angegeben wird, alle Änderungsanomalien verschwinden.
Gleichzeitig fordert die DKNF die eineindeutige Identifikation jedes Attributs einer Relation durch einen Schlüssel.
In der Praxis kann jedoch die Angabe einer allgemein formulierten eindeutigen Domäne mit unter zu Schwierigkeiten führen, weshalb die Prüfung auf Einhaltung dieser Normalform mit erheblichen technsichen Umsetzungsschwierigkeiten verbunden ist.

3 Arbeiten mit einer Datenbank

3.1 Codd'sche Regeln und Eigenschaften relationaler Systeme

Trotz des in dieser Hinsicht sehr eindeutigen grundlegenden Papiers von E. F. Codd über die Relationenstruktur (*A Relational Model of Data for Large Shared Data Banks*) existierte lange Zeit keine Übereinkunft darüber welche Eigenschaften ein relationales DBMS mindestens aufweisen muß um dieser Systemklasse zugerechnet werden zu können.
Daher definiert Codd 1986 in einem zweiteiligen Artikeln für die Zeitschrift *Computer World* 12 strenge Regeln die ein RDBMS aus seiner Sicht zwingend erfüllen muß um als solches eingestuft werden zu können.
Diese hierin erhobenen Forderungen sind jedoch so streng, daß sie bis heute kein System vollständig erfüllt.
Die Regeln sind nachfolgend mit ihren englischsprachigen Originalbezeichnungen wiedergegeben, da sich für sie bisher keine eindeutige und allgemeinverständliche deutsche Übersetzung etablieren konnte.

Regel 1: **The Information Rule**:
Alle Daten, die in einer Datenbank gespeichert werden sind auf dieselbe Art dargestellt, nämlich durch Werte in Tabellen.
Anmerkung: In dieser Definition wurde bewußt der Begriff der Tabelle gegenüber dem der Relation bevorzugt.

Regel 2: **Guaranteed Access Rule**:
Jeder gespeicherte Wert muß über Tabellenname, Spaltenname und Wert des Primärschlüssels zugreifbar sein, wenn der zugreifende Anwender über hinreichende Zugriffsrechte verfügt.

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

Regel 4: **Dynamic On-line Catalog Based on the Relational Model**: Forderung nach einem Online-Datenkatalog (*data dictionary*) in Form von Tabellen.
Dieser Katalog beschreibt die in der Datenbank abgelegten Tabellen hinsichtlich ihrer Struktur und zugelassenen Inhaltsbelegungen.

Regel 5: **Comprehensive Data Sublanguage Rule**:
Für das DBMS muß mindestens eine Sprache existieren durch die sich die verschiedenen Inhaltstypen (Tabelleninhalte, Sichten, Integritätsstrukturen (Schlüsselbeziehungen, Wertebereichseinschränkungen, Aufzählungstypen) sowie Zugriffsrechte) definieren lassen.

Regel 6: **View Updating Rule**:
Sofern theoretisch möglich, müssen Inhalte von Basistabellen auch über deren Sichten änderbar sein.

Regel 7: **High-level Insert, Update, and Delete**:
Innerhalb einer Operation können beliebig viele Tupel bearbeitet werden, d.h. die Operationen werden grundsätzlich mengenorientiert ausgeführt. Hierfür ist eine so abstrahierte Sicht dieser Operationen notwendig, daß keinerlei Information über die syteminterne Darstellung der Tupel notwendig ist.

Regel 8: **Physical Data Independence**:
Änderungen an der internen Ebene dürfen keine Auswirkungen auf die auf den abgespeicherten Daten operierenden Anwendungsprogramme besitzen.
Werden Daten demnach reorganisiert oder beispielsweise durch Indexe zugriffsbeschleunigt, so darf eine solche Änderung die auf die Datenbank zugreifenden Anwendungsprogramme nicht beeinträchtigen.

Regel 9: **Logical Data Independence**:
Änderungen des konzeptuellen Schemas dürfen keine Auswirkung auf die Anwedungsprogramme besitzen, solange diese nicht direkt von der Änderung betroffen sind.

Regel 10: **Integrity Independence**:
In Verfeinerung der fünften Regel wird gefordert, daß alle Integritätsbedingungen ausschließlich durch die Sprache des DBMS definieren lassen können müssen. Definierte Integritätsbedingungen müssen in Tabellen abgespeichert werden und durch das DBMS zur Laufzeit abgeprüft werden.

Im Mindesten werden folgende Forderungen durch verfügbare Systeme unterstützt:

- Kein Attribut welches Teil eines Primärschlüssels ist darf NULL sein.
- Ein Fremdschlüsselattribut muß als Wert des zugehörigen Primärschlüssels existieren.

Regel 11: **Distribution Independence**:
Die Anfragesprache muß so ausgelegt sein, daß Zugriffe auf lokal gehaltene Daten identisch denen auf verteilt gespeicherte Daten formuliert werden können.
Hieraus läßt sich auch die Ausdehnung der Forderungen nach logischer und physischer Datenunabhängigkeit für verteilte Datenbanken ableiten.

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

Zusätzlich faßt Codd in **Regel 0** nochmals die Anforderungen dahingehend zusammen, daß er postuliert, alle Operationen für Zugriff, Verwaltung und Wartung der Daten ausschließlich mittels relationaler Fähigkeiten abzuwickeln.

Derzeit existiert kein am Markt verfügbares kommerzielles System welches alle zwölf Regeln vollständig umsetzt. Insbesondere sind die Regeln 6, 9, 10, 11 und 12 in der Praxis schwer umzusetzen.

Darüber hinaus greifen die Coddschen Regeln nicht alle Gesichtspunkte des praktischen Datenbankeinsatzes auf. So bleiben Fragestellungen des Betriebs (wie Sicherungs-, Wiederherstellungs- und Sicherheitsaspekte) eines DBMS völlig ausgeklammert.

3.2 Implementierung des logischen Modells mit SQL-DDL

Die SQL-DDL dient allgemein der Definition und Verwaltung von Tabellen- und Indexdefinitionen innerhalb einer relationalen Datenbank entlang ihres gesamten Lebenszyklus, d.h. von ihrer Erstellung über alle Wartungsstadien bis hin zur Entfernung.

Nicht normiert durch den SQL-Standard sind die notwendigen Schritte zur Erzeugung einer Datenbank innerhalb eines Datenbankmanagementsystems. Überdies variieren die hierfür abzusetzenden Kommandos von Hersteller zu Hersteller und müssen der spezifischen Dokumentation entnommen werden.

Hinweis: Zwar läßt SQL inzwischen die beliebige Schreibung der Schlüsselworte (groß, klein oder gemischt) zu, zur bessern Hervorhebung und Kompatibilität mit existierender Literatur werden sie jedoch in den nachfolgenden Syntaxübersichten und Beispielen durchgehend in Großschreibung wiedergegeben.

Innerhalb der Syntaxbeschreibungen gelten folgende Konventionen:

- Schlüsselworte, die direkt wie abgedruckt eingegeben werden müssen sind großgeschrieben.
- Optionale Bestandteile, die weggelassen werden können sind in eckigen Klammern („[]“) dargestellt.

Die Klammern selbst sind nicht Bestandteil der Syntax und müssen nicht eingegeben werden.

- Dargestellte runde Klammern („()“) sind Syntaxbestandteil und müssen unverändert eingegeben werden.
- Senkrechte Striche („|“) trennen Alternativen von denen jeweils eine ausgewählt werden kann, nicht jedoch mehrere.
- Kommentare, die auf die Ausführung keinen Einfluß haben werden durch zwei Minuszeichen („--“) eingeleitet und enden mit dem Zeilenende.
- Alle SQL-Befehle werden generell durch ein Semikolon abgeschlossen. Dieses ist aus Übersichtlichkeitsgründen in den Syntaxübersichten weggelassen.

Erzeugen von Tabellen

Die SQL-Anweisung `CREATE TABLE` dient der Erzeugung neuer Tabellen innerhalb einer bestehenden Datenbank. Sie legt die Struktur und die zugelassenen Typausprägungen, sowie Einschränkungen hinsichtlich der erlaubten Werte fest.

Die vereinfachte Syntax der `CREATE TABLE`-Anweisung lautet:

```
CREATE [TEMPORARY] TABLE tbl_name [(create_definition,...)]
[table_options] [select_statement]

create_definition:
  col_name type [NOT NULL | NULL] [DEFAULT default_value] [AUTO_INCREMENT]
            [PRIMARY KEY] [reference_definition]
  or    PRIMARY KEY (index_col_name,...)
  or    KEY [index_name] (index_col_name,...)
  or    INDEX [index_name] (index_col_name,...)
  or    UNIQUE [INDEX] [index_name] (index_col_name,...)
  or    [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_col_name,...)
            [reference_definition]

type:
        TINYINT[(length)] [UNSIGNED] [ZEROFILL]
  or    SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
  or    MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
  or    INT[(length)] [UNSIGNED] [ZEROFILL]
  or    INTEGER[(length)] [UNSIGNED] [ZEROFILL]
  or    BIGINT[(length)] [UNSIGNED] [ZEROFILL]
  or    REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
  or    DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
  or    FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
  or    DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
  or    NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
  or    CHAR(length) [BINARY]
  or    VARCHAR(length) [BINARY]
  or    DATE
  or    TIME
  or    TIMESTAMP
  or    DATETIME
  or    TINYBLOB
  or    BLOB
  or    MEDIUMBLOB
  or    LONGBLOB
  or    TINYTEXT
  or    TEXT
  or    MEDIUMTEXT
  or    LONGTEXT
  or    ENUM(value1,value2,value3,...)
  or    SET(value1,value2,value3,...)

index_col_name:
        col_name [(length)]

reference_definition:
        REFERENCES tbl_name [(index_col_name,...)]
                   [MATCH FULL | MATCH PARTIAL]
                   [ON DELETE reference_option]
                   [ON UPDATE reference_option]
```

```
reference_option:
        RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

```
mysql> CREATE TABLE Person(
        Name VARCHAR(25)
);
```

Beispiel 21: Erzeugung einer Tabelle

Die Anweisung aus Beispiel 21 stellt den einfachsten Fall ein Tabellenerzeugungsanweisung dar. Es wird die Tabelle `Person`, die mit `Name` nur über eine Spalte verfügt erzeugt.
Eine „kleinere“ Fassung ist nicht möglich, da spaltenlose Tabellen nicht erstellt werden können.

Informationen über eine angelegte Tabelle können durch den Befehl `DESCRIBE` gefolgt vom Namen der abzufragenden Tabelle erlangt werden:

```
mysql> DESCRIBE Person;
+-------+-------------+-------------------+------+-----+---------+-------+
| Field | Type        | Collation         | Null | Key | Default | Extra |
+-------+-------------+-------------------+------+-----+---------+-------+
| Name  | varchar(25) | latin1_swedish_ci | YES  |     | NULL    |       |
+-------+-------------+-------------------+------+-----+---------+-------+
```

Beispiel 22: Ermittlung von Tabelleninformation

Im Beispiel werden die zuvor festgelegten Daten wie Spaltenname (`Name`) und Datentyp (`VARCHAR (25)`) ermittelt, sowie die Belegung einiger Vorgabewerte (u.a. `NULL` und `Default`).

Durch Angabe des Schlüsselwortes `TEMPORARY` können Tabellen erstellt werden, die während der Arbeit mit der Datenbank den herkömmlichen gleichgestellt behandelt werden, jedoch dem Ende der Datenbankverbindung automatisch inclusive aller darin abgelegten Daten aus der Datenbank entfernt werden.

```
mysql> CREATE TEMPORARY TABLE Person2(
        Name VARCHAR(25)
);

--Verbindungsende
--Aufbau einer neuen Verbindung

mysql> DESCRIBE Person2;
ERROR 1146: Table 'SQLTest.Person2' doesn't exist
```

Beispiel 23: Erzeugung einer temporären Tabelle

Wird die Datenbankverbindung getrennt und neu aufgebaut, so ist die zuvor temporär erstellte Tabelle Peson2 nicht mehr vorhanden und zugreifbar. Das `DESCRIBE`-Kommando liefert daher einen Fehler.

Wie bereits in Beispiel 21 gezeigt muß jede Spalte einer Tabelle einen Datentyp besitzen. Dieser definiert die zugelassenen Wertbelegungen und wird durch das DBMS bei jeder schreibenden Operation (d.h. Einfügen, Ändern und Leeren) geprüft.
Wird versucht ein ungültiger Wert zu setzen, so erfolgt eine Fehlermeldung und die Ablehnung des Eintragungs- oder Änderungswunsches. Im Falle eines zeichenkettenwertigen Datentyps (z. B. `VARCHAR`) erfolgt keine Fehlermeldung, sondern die Werte werden nur abgeschnitten eingefügt.

```
mysql> CREATE TABLE Person(
        Name VARCHAR(25)
);

mysql> INSERT INTO Person values("Max Mustermann");
--ok

mysql> INSERT INTO Person values("Franz Obermüller-Hinterhuber-Niedermayer");
--keine Fehlermeldung
--Wert wird jedoch nur abgeschnitten eingefügt:

mysql> SELECT * FROM Person;
+---------------------------+
| Name                      |
+---------------------------+
| Max Mustermann            |
| Franz Obermüller-Hinterhu |
+---------------------------+
```

Beispiel 24: Auswirkung von Datentypen I

Beispiel 24 zeigt die Auswirkung eines gesetzten Datentypen VARCHAR. Werden, wie im Beispiel Werte eingefügt, die die zulässige maximale Zeichenkettenlänge überschreiten, so werden die überzähligen Zeichen ohne Fehlermeldung abgeschnitten.

```
mysql> CREATE TABLE Person(GebDat date);

mysql> INSERT INTO Person values ('1970-12-12');
--ok

mysql> insert into Person values ('1970-19-42');
--offensichtlich falsch
--Übernommener Wert: 0000-00-00
```

Beispiel 25: Auswirkung von Datentypen II

Im Beispiel 25 wird ein offensichtlich ungültiges Datum eingefügt. Die Datenbank übernimmt jedoch nicht diesen falschen Wert sondern setzt den Vorgabewert von `0000-00-00`.
Dasselbe geschieht auch bei numerischen Datentypen (etwa: `INTEGER`) wenn versucht wird sie mit einer Zeichenkette zu belegen.

MySQL unterstützt drei Datentypklassen:

- Numerischedatentypen zur Darstellung von Zahlen.
- Zeichenkettendatentypen zur Darstellung von Texten und Binärdaten.
- Datumsdatentypen zur Darstellung von Uhrzeit- und Datumsformaten.

Die derzeit durch MySQL angebotenen Datentypen sind in der nachfolgenden Tabelle zusammengestellt:

Typname	Beispiel	Bemerkung
Numerische Datentypen		
`TINYINT [UNSIGNED]`	(Vorzeichenbehaftete) Ganzzahl der Breite acht Bit.	(2^7, ..., -1) 0, 1, ..., 2^7-1), (2^7, ... 2^8-1) (-128, ..., -1,) 0, 1, ..., 127, (128, ..., 255)
`BOOL`	Boole'scher Wahrheitswert.	Zugelassene Belegungen `0` und `1`. Synonym für `TINYINT(1)`
`SMALLINT [UNSIGNED]`	(Vorzeichenbehaftete) Ganzzahl der Breite 16 Bit.	(2^{15}, ..., -1) 0, 1, ..., 2^{15}-1), (2^{15}, ... 2^{16}-1) (-32.768, ..., -1,) 0, 1, ..., 32.767, (32.768, ..., 65.535)
`MEDIUMINT [UNSIGNED]`	(Vorzeichenbehaftete) Ganzzahl der Breite 24 Bit.	(2^{23}, ..., -1) 0, 1, ..., 2^{23}-1), (2^{23}, ... 2^{24}-1) (-8.388.608, ..., -1) 0, 1, ..., 8.388.607, (8.388.608, ..., 16.777.215)
`INT [UNSIGNED]`	(Vorzeichenbehaftete) Ganzzahl der Breite 32 Bit.	(2^{31}, ..., -1) 0, 1, ..., 2^{31}-1), (2^{31}, ... 2^{32}-1) (-2.147.483.648, ..., -1) 0, 1, ..., 2.147.483.647, (2.147.483.648, ..., 4.294.967.295)

INTEGER [UNSIGNED]		Synonym für INT
BIGINT [UNSIGNED]	(Vorzeichenbehaftete) Ganzzahl der Breite 64 Bit.	(2^{63}, ..., -1) 0, 1, ..., 2^{63}-1), (2^{63}, ... 2^{64}-1) (-9.223.372.036.854.774.808, ..., -1) 0, 1, ..., 9.223.372.036.854.774.807, (9.223.372.036.854.774.808, ..., 18.446.744.073.709.551.615)
FLOAT [UNSIGNED]	(Vorzeichenbehaftete) Fließkommazahl der Breite 32 Bit, gemäß dem Standard IEEE-754	$(-(2-2^{-23})^{127}$, ..., $(2-2^{-23})^{127})$ $-3{,}402...*10^{38}$, ..., $-1{,}175...*10^{-38}$ und $1{,}175...*10^{-38}$ ... $3{,}402...*10^{38}$
DOUBLE [UNSIGNED]	(Vorzeichenbehaftete) Fließkommazahl der Breite 64 Bit, gemäß dem Standard IEEE-754	$(-(2-2^{-52})^{1023}$, ... $(2-2^{-52})^{1023}$ $-1{,}797...*10^{-308}$, ..., $-2{,}225...*10^{308}$ und $1{,}797...^{308}$, ..., $2{,}225...^{-308}$
REAL [UNSIGNED]		Synonym für DOUBLE
DECIMAL [(Genauigkeit, [Nachkommastellen])]	Festkommazahl beliebiger Genauigkeit	DECIMAL(9,2) liefert eine Dezimalzahl mit neun Gesamtziffern, davon zwei Nachkommastellen.
DEC		Synonym für DECIMAL
NUMERIC		Synonym für DECIMAL
Zeichenkettendatentypen		
CHAR [Länge]	Textfeld konstanter Länge bis zu 255 Zeichen, das immer die angegebene Anzahl Speicherstellen benötigt.	Dies ist ein Test0x20;0x20;0x20;
CHARACTER		Synonym zu CHAR
NCHAR		Synonym zu CHAR
NATIONAL CHARACTER		Synonym zu CHAR
CHARACTER VARYING		Synonym zu VARCHAR
NATIONAL VARCHAR		Synonym zu VARCHAR
VARCHAR [Länge]	Textfeld variabler Länge. Überzählige Leerzeichen am Ende einer Zeichenkette werden vor dem Abspeichern entfernt.	Dies ist ein Test
TINYTEXT	Textfeld variabler Länge, bis zu 2^8-1 (=255) Zeichen.	... etwas mehr Text
TEXT	Textfeld variabler Länge, bis zu 2^{16}-1 (=65.535) Zeichen.	Sehr viel ... Text hier ...
MEDIUMTEXT	Textfeld variabler Länge, bis zu 2^{24}-1 (=16.777.215) Zeichen.	... etwas mehr Text hier ...
LONGTEXT	Textfeld variabler Länge, bis zu 2^{32}-1 (=2.294.967.295) Zeichen.	... noch mehr Text hier ...
TINYBLOB	Binäre Form von TINYTEXT.	
MEDIUMBLOB	Binäre Form von MEDIUMTEXT	
	BLOB	Binäre Form von TEXT
Datumstypen		
DATE	Datum in ISO-8601-Schreibweise (JJJJ-MM-TT)	2004-06-08
TIME	Uhrzeit gemäß ISO 8601 (hh:mm:ss)	07:45:00
DATETIME	Datum und Uhrzeit gemäß ISO 8601 (JJJJ-MM-TT hh:mm:ss)	2005-05-27 07:45:00
TIMESTAMP	Sekundengenauer Zeitpunkt zwischen 1970-01-01 und 2037-12-31	Anwenderdarstellung: 2004-06-08 12:52:03
YEAR	Vierstellige Jahreszahl	2004
Komplexe Datentypen		
ENUM	Aufzählungstyp mit bis zu 65.535 Elementen	
SET	Menge mit bis zu 64 Elementen	

Jede Spalte einer Relation muß mit genau einem Datentyp der oben dargestellten Liste versehen werden. Nachfolgend sind einige Definitionen und Besonderheiten zusammengestellt:

```
mysql> CREATE TABLE test(wenigText CHAR(300));
--zulässige Grenze für CHAR überschritten ...

mysql> DESCRIBE test;
+-----------+------+-------------------+------+-----+---------+-------+
| Field     | Type | Collation         | Null | Key | Default | Extra |
+-----------+------+-------------------+------+-----+---------+-------+
| wenigText | text | latin1_swedish_ci | YES  |     | NULL    |       |
+-----------+------+-------------------+------+-----+---------+-------+
```

Beispiel 26: Auswirkung von Datentypen III

Beispiel 26 zeigt die automatische Konversion des Datentypen CHAR in der Datenbank in TEXT sobald bereits bei der Anlage die zulässige Größenbegrenzung für CHAR überschritten wird. Werden zur Laufzeit Texte größer als die im CREATE TABLE-Ausdruck angegebene Maximalkapazität abgespeichert, so werden alle überzähligen Zeichen abgeschnitten.

```
mysql> CREATE TABLE test(ts timestamp, x VARCHAR(10));
Query OK, 0 rows affected (0.35 sec)

mysql> INSERT INTO test values(null, "abc");
Query OK, 1 row affected (0.06 sec)

mysql> INSERT INTO test values(null, "def");
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO test values(null, "abc");
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM test;
+---------------------+------+
| ts                  | x    |
+---------------------+------+
| 2003-05-26 23:25:51 | abc  |
| 2003-05-26 23:26:13 | def  |
| 2003-05-26 23:26:28 | abc  |
+---------------------+------+
3 rows in set (0.00 sec)

mysql> UPDATE test SET x="xyz" WHERE x="abc";
Query OK, 2 rows affected (0.05 sec)
Rows matched: 2  Changed: 2  Warnings: 0

mysql> SELECT * FROM test;
+---------------------+------+
| ts                  | x    |
+---------------------+------+
| 2003-05-26 23:27:10 | xyz  |
| 2003-05-26 23:26:13 | def  |
| 2003-05-26 23:27:10 | xyz  |
+---------------------+------+
3 rows in set (0.00 sec)
```

Beispiel 27: Auswirkung von Datentypen IV

Das Beispiel zeigt die Nutzung des Typs TIMESTAMP. Spalten dieses Typs werden automatisch durch das DMBS mit Werten versorgt werden. Daher ist die Belegung mit NULL bei Einfügung der drei Zeichenketten wirkungslos.
Überdies wird der Wert jeder TIMESTAMP-typisierten Spalte (im Beispiel: ts) bei jedem Schreibvorgang aktualisiert. Dies zeigt die nochmalige Ausgabe der Tabelleninhalte nach Aktualisierung der beiden Tupel, die für das Attribut x den Wert abc aufweisen.

```
mysql> create table Ampel(farbe enum('rot','gelb','gruen'));
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Ampel VALUES('rot');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT into Ampel VALUES('blau');
Query OK, 1 row affected (0.04 sec)

mysql> SELECT * FROM Ampel;
+-------+
| farbe |
+-------+
| rot   |
|       |
+-------+

mysql> INSERT INTO Ampel Values(2);
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM Ampel;
+-------+
| farbe |
+-------+
| rot   |
|       |
| gelb  |
+-------+
3 rows in set (0.00 sec)
```

Beispiel 28: Auswirkung von Datentypen V

Das Beispiel 28 zeigt die Nutzung eines Aufzählungstypen.
Er erlaubt ausschließlich das Einfügen der vordefinierten Werte und legt für alle ungültigen Belegungen (im Beispiel: `blau`) die leeren Zeichenkette ab.
Die Werte können dabei wie in der Aufzählung definiert oder durch ihre Indexposition (beginnend ab 1) gespeichert werden.

```
mysql> CREATE TABLE test(x SET("a","b","c","d"));
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO test VALUES("a");
Query OK, 1 row affected (0.07 sec)

mysql> INSERT INTO test values("a,b");
Query OK, 1 row affected (0.04 sec)

mysql> INSERT INTO test values("a,c");
Query OK, 1 row affected (0.07 sec)

mysql> INSERT INTO test values("b,c");
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM test;
+------+
| x    |
+------+
| a    |
| a,b  |
| a,c  |
| b,c  |
+------+
4 rows in set (0.00 sec)

mysql> select * FROM test WHERE x & 1;
+------+
| x    |
+------+
| a    |
```

```
| a,b  |
| a,c  |
+------+
3 rows in set (0.00 sec)
--liefert alle Tupel, die das zweite Mengenelement (="a") enthalten
```

Beispiel 29: Auswirkung von Datentypen VI

Das Beispiel zeigt die Definition eines mengenwertigen Datentyps, d.h. einer Tabellenspalte, die mehr als einen Wert aufnehmen kann sowie die Abfragemöglichkeiten dafür.
Hinweis: Dieser Datentyp führt bereits in die NF^2-Datenstrukturen über und wird daher nicht im Rahmen des Entwurfsprozesses im konzeptuellen Schema verwendet.

Ergänzend zur Datentypangabe können für jede Spalte weitere einschränkende Angaben zur Spezifikation der erlaubten Werte getroffen werden.

`NOT NULL` legt hierbei fest, daß ein Tupel für eine Spalte zwingend einen von `NULL` verschiedenen Wert besitzen muß.

```
mysql> CREATE TABLE Person(Name VARCHAR(20), PersAuswNr INT NOT NULL);
Query OK, 0 rows affected (0.08 sec)

mysql> INSERT INTO Person VALUES('Max Obermüller', '123456789');
Query OK, 1 row affected (0.11 sec)

mysql> INSERT INTO Person VALUES('Xaver Hinterhuber', NULL);
ERROR 1048: Column 'PersAuswNr' cannot be null

mysql> select * from Person;
+----------------+------------+
| Name           | PersAuswNr |
+----------------+------------+
| Max Obermüller |  123456789 |
+----------------+------------+
1 row in set (0.00 sec)
```

Beispiel 30: Definition einer Spalte als NOT NULL

Das Beispiel zeigt eine Tabelle, bei der das Attribut `PersAuswNr` als `NOT NULL` definiert wurde. Einfüge- oder Aktualisierungsversuche, die zu Nullwerten dieses Attributs führen würden, werden durch das DMBS unterbunden.

Alternativ dazu gestattet die Angabe von `NULL` die Existenz von Nullwerten in der Tabelle. Diese Definition ist optional und wird bei fehlender Angabe automatisch als Vorgabe gesetzt.
Das Beispiel zeigt die Definition der Spalte `Autofahrer` als `NULL`, die neben den beiden vorgegebenen Werten auch keinen Wert enthalten darf.

```
mysql> CREATE TABLE Person(
    -> Name VARCHAR(20),
    -> PersAuswNr INT NOT NULL,
    -> Autofahrer ENUM('J','N') NULL);
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES('Max Obermüller', '123456789', NULL);
Query OK, 1 row affected (0.12 sec)

mysql> INSERT INTO Person VALUES('Xaver Hinterhuber', '234567891', 'J');
Query OK, 1 row affected (0.04 sec)

mysql> select * from Person;
+-------------------+------------+------------+
| Name              | PersAuswNr | Autofahrer |
+-------------------+------------+------------+
| Max Obermüller    |  123456789 | NULL       |
| Xaver Hinterhuber |  234567891 | J          |
+-------------------+------------+------------+
3 rows in set (0.01 sec)
```

Beispiel 31: Definition einer Spalte als NULL

Die Angabe der Klausel `DEFAULT VALUE` gestattet es einen Vorgabewert zu definieren, der gesetzt wird wenn kein Wert für eine Spalte angegeben wird.
Dies ersetzt jedoch nicht automatisch die Möglichkeit des Auftretens von Nullwerten innerhalb einer Spalte. Diese können auch weiterhin auftreten, sofern sie explizit eingefügt werden.
Die unterschiedlichen Wirkungsweisen zeigt Beispiel 32. Dort findet sich die Spalte `Autofahrer` (vorgabegemäß, da keine andere Angabe erfolgte) als nullwertfähig mit Vorgabewert `J`, sowie die Spalte `Hundebesitzer`, die mit Vorgabe `N` und als nicht nullwertfähig deklariert wurde.

```
mysql> CREATE TABLE Person(
    -> Name VARCHAR(20) NOT NULL,
    -> Autofahrer ENUM('J','N') DEFAULT 'J',
    -> Hundebesitzer ENUM('J','N') NOT NULL DEFAULT 'N'
    -> );
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES('Xaver Obermüller', 'J', 'J');
Query OK, 1 row affected (0.10 sec)

mysql> INSERT INTO Person VALUES('Max Hinterhuber', NULL, 'N');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES('Schorsch Huber');
Query OK, 1 row affected (0.05 sec)

mysql> SELECT * FROM Person;
+------------------+------------+---------------+
| Name             | Autofahrer | Hundebesitzer |
+------------------+------------+---------------+
| Xaver Obermüller | J          | J             |
| Max Hinterhuber  | NULL       | N             |
| Schorsch Huber   | J          | N             |
+------------------+------------+---------------+
3 rows in set (0.00 sec)
```

Beispiel 32: Definition einer Spalte mit Vorgabewerten

Ist die Auszeichnung einer einzelnen Spalte als Primärschlüssel gewünscht, so kann der Definition `(PRIMARY) KEY` nachgestellt werden um dies zu erreichen.
Die Definition eines Primärschlüssels impliziert den Zwang in jedem Tupel einen von `NULL` verschiedenen eineindeutigen Wert dafür abspeichern zu müssen. Primärschlüsselspalten sind damit immer auch `NOT NULL`.
Zusätzlich kann für jede Tabelle höchstens ein Primärschlüsselattribut angegeben werden.
Hinweis: Aus mehr als einer Spalte zusammengesetzte Primärschlüssel können nicht durch diese Syntax gebildet werden.

```
mysql> CREATE TABLE Person(Name VARCHAR(20) PRIMARY KEY, Adresse VARCHAR(50));
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO Person VALUES('Xaver Obermüller', 'Dorfstr. 12');
Query OK, 1 row affected (0.10 sec)

mysql> INSERT INTO Person VALUES('Hans Hintermeier', 'Dorfstr. 13');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES ('Schorsch Huber');
Query OK, 1 row affected (0.06 sec)

mysql> select * from Person;
+------------------+--------------+
| Name             | Adresse      |
+------------------+--------------+
| Hans Hintermeier | Dorfstr. 13  |
| Schorsch Huber   | NULL         |
| Xaver Obermüller | Dorfstr. 12  |
+------------------+--------------+
```

```
4 rows in set (0.00 sec)

mysql> INSERT INTO Person VALUES(NULL, 'Hauptstr. 11');
ERROR 1048: Column 'Name' cannot be null
```

Beispiel 33: Definition eines Primärschlüssels

Zur Erstellung eines zusammengesetzten Primärschlüssels kann nicht das nachgestellte Schlüsselwort `PRIMARY KEY` verwendet werden, da seine wiederholte Angabe mehrdeutig wäre. Für diesen Fall muß eine gesondert `PRIMARY KEY`-Definition in den `CREATE TABLE`-Ausdruck aufgenommen werden:

```
CREATE TABLE DEPT_LOCATIONS(
   DNUMBER INTEGER(1) NOT NULL,
   DLOCATION VARCHAR(20) NOT NULL,
   PRIMARY KEY (DNUMBER, DLOCATION));
```

Beispiel 34: Definition eines zusammengesetzten Primärschlüssels

Das Beispiel zeigt verschiedene Einfügeoperationen. Bemerkenswert ist die letzte, die das Primärschlüsselattribut leer läßt. Die hierbei erfolgende Belegung mit der leeren Zeichenkette (nicht `NULL`!) ist ein gültiger Eintrag im Sinne des angegebenen Datentypen `VARCHAR` und der Restriktion keine Belegung mit `NULL` vorzunehmen.

Soll ein nicht-sprechender Schlüssel (z.B. eine einfache Zählnummer) zur Identifikation genutzt werden, so kann diese durch das DBMS automatisiert bereitgestellt werden.

```
mysql> CREATE TABLE Person(
    -> LfdNr Int AUTO_INCREMENT PRIMARY KEY,
    -> Name VARCHAR(20) NOT NULL);
Query OK, 0 rows affected (0.07 sec)

mysql> INSERT INTO Person VALUES(1,'Max Obermüller');
Query OK, 1 row affected (0.09 sec)

mysql> INSERT INTO Person VALUES(3,'Schorsch Hinterhuber');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person VALUES(3,'Xaver Mayer');
ERROR 1062: Duplicate entry '3' for key 1

mysql> SELECT * FROM Person;
+-------+----------------------+
| LfdNr | Name                 |
+-------+----------------------+
|     1 | Max Obermüller       |
|     3 | Schorsch Hinterhuber |
+-------+----------------------+
2 rows in set (0.00 sec)

mysql> INSERT INTO Person (Name) VALUES('Xaver Mayer');
Query OK, 1 row affected (0.05 sec)

mysql> INSERT INTO Person (Name) VALUES('Hans Huber');
Query OK, 1 row affected (0.04 sec)

mysql> select * from Person;
+-------+----------------------+
| LfdNr | Name                 |
+-------+----------------------+
|     1 | Max Obermüller       |
|     2 | Xaver Mayer          |
|     3 | Schorsch Hinterhuber |
|     4 | Hans Huber           |
+-------+----------------------+
4 rows in set (0.00 sec)
```

Beispiel 35: Definition eines automatisch befüllten Primärschlüssels

Im Beispiel 35 wird der Wert der Spalte LfdNr, sofern nicht durch den Anwender explizit angegeben, automatisch ermittelt und eingefügt.

Zur Zugriffsbeschleunigung dienende Indexstrukturen können bereits zum Tabellenerstellungszeitpunkt durch den Anwender angegeben werden. Diese werden jedoch nicht der Spaltendefinition als nachgestellt, sondern bilden einen eigenen Eintrag innerhalb der Tabellendefinition.
Ein Index kann gleichzeitig eine oder mehrere Spalten umfassen. Beispiel Beispiel 36 zeigt dies:

```
mysql> CREATE TABLE Person(
    -> Name VARCHAR(20) PRIMARY KEY,
    -> GebDat DATE, Str_HsNr VARCHAR(20),
    -> PLZ CHAR(5),
    -> Ort VARCHAR(50),
    -> INDEX GebDatIdx (GebDat),
    -> INDEX AdresseIndex (Str_HsNr, PLZ, Ort));
Query OK, 0 rows affected (0.07 sec)
```

Beispiel 36: Definition von Indexen

Als beschränkende Verschärfung, die sich auch positiv auf die Zugriffsgeschwindigkeit auswirkt, kann ein Index als UNIQUE INDEX definiert werden. Er darf dann ausschließlich eindeutige Werte oder Wertkombinationen aufnehmen.

Erzeugung von Fremdschlüsselbeziehungen

Die Erzeugung von Fremdschlüsselbeziehungen ist das integrale Element zur Wahrung der referentiellen Integrität. Fremdschlüsselbeziehungen können bereits zum Erstellungszeitpunkt einer Tabelle angegeben werden wie Beispiel 37 zeigt oder nachträglich durch einen ALTER TABLE-Ausdruck hinzugefügt werden wie durch Beispiel 38 gezeigt.

```
CREATE TABLE DEPARTMENT(
   DNAME VARCHAR(20) NOT NULL,
   DNUMBER INTEGER(1) PRIMARY KEY,
   MGRSSN INTEGER(9),
   MGRSTARTDATE DATE);

CREATE TABLE EMPLOYEE(
   FNAME VARCHAR(10) NOT NULL,
   MINIT VARCHAR(1),
   LNAME VARCHAR(10) NOT NULL,
   SSN INTEGER(9) PRIMARY KEY,
   BDATE DATE,
   ADDRESS VARCHAR(30),
   SEX ENUM('M','F'),
   SALARY REAL(7,2) UNSIGNED,
   SUPERSSN INTEGER(9),
   DNO INTEGER(1) REFERENCES DEPARTMENT(DNUMBER),
   INDEX DNO_IDX (DNO));
```

Beispiel 37: Erzeugung von Fremdschlüsselbeziehungen zum Tabellenersellungszeitpunkt

Das Beispiel erzeugt neben der angestrebten Fremdschlüsselbeziehungen zwischen dem Attribut DNO der Tabelle EMPLOYEE und dem Primärschlüssel DNUMBER in DEPARTMENT einen Index auf den Fremdschlüssel innerhalb der Tabelle EMPLOYEE.
Dies ist für einige Datenbankmanagementsysteme (darunter MySQL) notwendig, um die Zugriffe auf den Fremdschlüssel zu beschleunigen.

Das nachfolgende Beispiel liefert dasselbe Ergebnis, jedoch unter nachträglicher (d.h. nach dem Erstellungszeitpunkt der Tabellen) Fremdschlüsselerzeugung:

```
CREATE TABLE DEPARTMENT(
   DNAME VARCHAR(20) NOT NULL,
   DNUMBER INTEGER(1) PRIMARY KEY,
   MGRSSN INTEGER(9),
   MGRSTARTDATE DATE);

CREATE TABLE EMPLOYEE(
   FNAME VARCHAR(10) NOT NULL,
   MINIT VARCHAR(1),
   LNAME VARCHAR(10) NOT NULL,
   SSN INTEGER(9) PRIMARY KEY,
   BDATE DATE,
   ADDRESS VARCHAR(30),
   SEX ENUM('M','F'),
   SALARY REAL(7,2) UNSIGNED,
   SUPERSSN INTEGER(9),
   DNO INTEGER(1));

ALTER TABLE EMPLOYEE ADD INDEX DNO_IDX (DNO);
ALTER TABLE EMPLOYEE ADD CONSTRAINT DNO_FK FOREIGN KEY (DNO) REFERENCES DEPARTMENT
(DNUMBER);
```

Beispiel 38: Nachträgliche Erzeugung von Fremdschlüsselbeziehungen

3.3 Der Anfrageteil von SQL

Anfragen zur Ermittlung von Datenbankinhalten stellen den eigentlichen Sprachkern von SQL und zweifellos den in der Praxis bedeutsamsten Anteil der Sprache dar.
Die gesamte Mächtigkeit des Anfrageteils von SQL erschließt sich durch das Schlüsselwort SELECT. Es gestattet Anfragen theoretisch unbegrenzter Komplexität in einer uniformen und leicht zu behaltenden Syntax zu formulieren, deren Mächtigkeit von einfachsten Anfragen bis zu aufwendigen Auswertungen reicht.

Anfragen von Datenbankinhalten

Die SQL-Anweisung SELECT dient der Abfrage von in einer Datenbank abgelegten Inhalten. Sie benötigt Wissen über die angelegten Tabellen sowie deren Struktur hinsichtlich Spalten und deren Typen.
Alle nachfolgenden Beispielanfragen beziehen sich, sofern nicht anders angegeben auf die Demodatenbank.

Die vereinfachte Syntax der SELECT-Anweisung lautet:

```
SELECT [ALL|DISTINCT] select_item,...
FROM table_specification,...
[WHERE search_condition]
[GROUP BY grouping_column,...]
[HAVING search_condition]
[ORDER BY sort_specification,...]
```

```
SELECT FNAME
FROM EMPLOYEE;
```

Beispiel 39: Einfache Anfrage

Die Anfrage liefert die Inhalte der Spalte FNAME aller in der Tabelle EMPLOYEE abgelegten Tupel. Die Werte werden in keiner vorgegebenen Reihenfolge ausgegeben, d.h. eine etwaige Sortierung ist zufallsbedingt und kann durch DBMS interne Reorganisationsprozesse zerstört werden.

Durch diese Anfrage wird als Resultat eine nicht in der Datenbank abgelegte Tabelle erzeugt, welche nur die im SELECT-Ausdruck angegebenen Spalten enthält. Die Ergebnistabelle stimmt zwar in Tupelanzahl mit der Ursprungstabelle überein, blendet jedoch einzelne Attribute aus. Diese Vorgang wird als *Projektion* bezeichnet.

Definition 42: *Projektion*
Die Projektion blendet einzelne Spalten aus.

```
SELECT FNAME, MINIT, LNAME, SSN, BDATE, ADDRESS, SEX, SALARY, SUPERSSN, DNO
FROM EMPLOYEE;
```

Beispiel 40: Anfrage aller Spalten einer Tabelle

Die Abfrage aus Beispiel 40 liefert die Wertinhalte aller Spalten (sie sind explizit nach dem Schlüsselwort SELECT angegeben) der Relation EMPLOYEE.
Als Besonderheit wird für das Attribut SUPERSSN des Mitarbeiters James E. Borg der Wert NULL ausgegeben. Diese Zeichenkette gibt an, daß für dieses Attribut der Relation kein Wert abgespeichert wurde.

Die schreibaufwendige und damit fehlerträchtige Explizierung aller Spalten einer Relation ist kaum praktikabel und überdies äußerst änderungssensitiv im Falle der Aufnahme neuer Spalten oder der Lösung Existierender.
Aus diesem Grunde kann statt des Spaltennamens ein Stern „*“ als Jokerzeichen stellvertretend für alle Spalten einer Relation angegeben werden.
Beispiel 41 zeigt dies als Umschreibung der Anfrage aus Beispiel 40:

```
SELECT *
FROM EMPLOYEE;
```

Beispiel 41: Anfrage aller Spalten einer Tabelle mit Jokerzeichen

Enthält die Ausgabe nicht den Primärschlüssel einer Tabelle, so kann es vorkommen, das mehrfach dieselben Werte ausgegeben werden. Dies kann durch Angabe des Schlüsselwortes DISTINCT in der SELECT-Klausel vermieden werden.
DISTINCT überschreibt das vorgegebene Verhalten (ALL) alle Einträge auszugeben.

```
SELECT DISTINCT SALARY
FROM EMPLOYEE;
```

Beispiel 42: Duplikatfreie Ausgabe aller verschiedenen Werte

Beispiel 42 liefert alle verschiedenen Werteinträge der Spalte SALARY duplikatfrei.

Durch Angabe mehrerer Einträge in der FROM-Klausel können Inhalte aus verschiedenen Tabellen innerhalb einer Anfrage extrahiert werden:

```
SELECT DNAME, PNUMBER
FROM DEPARTMENT, PROJECT;
```

Beispiel 43: Anfrage auf zwei Tabellen

Die Anfrage bildet das kartesische Produkt der beiden angefragten Tabellen.

Aliasbildung

Bei Anfragen über mehrere Tabellen kann es zu Problemen hinsichtlich der Eindeutigkeit der Spaltenbezeichner kommen. So würde die Anfrage aus Beispiel 44 nicht die für ESSN abgelegten Werte liefern, sondern den Fehler ERROR 1052: Column: 'ESSN' in field list is ambiguous, da in jeder der beiden in der Anfrage berücksichtigten Tabellen eine Spalte mit ESSN benannt ist.

```
SELECT ESSN, ESSN
FROM WORKS_ON, DEPENDENT;
```

Beispiel 44: Fehlerhafte Anfrage auf zwei Tabellen

Als Lösung bietet SQL die Möglichkeit den Spaltennamen zusätzlich durch Voranstellung des Namens der die Spalte beherbergenden Tabelle zu qualifizieren um die erforderliche Eindeutigkeit herzustellen:

```
SELECT WORKS_ON.ESSN, DEPENDENT.ESSN
FROM WORKS_ON, DEPENDENT;
```

Beispiel 45: Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen

Unter Nutzung der Möglichkeit Alternativnamen für Tabellen, sog. *Aliasnamen*, anzugeben ergibt sich eine in der Schreibung kompaktere Umsetzung:

```
SELECT w.ESSN, d.ESSN
FROM WORKS_ON AS w, DEPENDENT AS d;
```

Beispiel 46: Lösung des Mehrdeutigkeitsproblems bei Anfrage auf zwei Tabellen

Gleichzeitig kann die Aliasbildung eingesetzt werden, um die Benennung der Spalten bei der Ausgabe zu modifizieren. Auf diesem Wege können wenig sprechende Namen oder Doppelbenennungen umgangen werden.

```
SELECT w.ESSN AS "Mitarbeiter Sozialversicherungsnummer",
d.ESSN AS "Sozialversicherungsnummer des Verwandten"
FROM WORKS_ON AS w, DEPENDENT AS d;
```

Beispiel 47: Umbenennung von Ausgabespalten

Berechnete Ausgaben

Durch die Angabe einfacher arithmetischer Formeln in der SELECT-Klausel können vor ihrer Ausgabe Berechnungen auf den Werten aus der Datenbank angestellt werden.

```
SELECT FNAME, SALARY*12 as Jahreseinkommen
FROM EMPLOYEE;
```

Beispiel 48: Berechnungen I

Beschränkung der Ergebnismenge

Die bisher betrachteten Anfrageformen lieferten immer die gesamten Inhalte der betrachteten Tabellen. Durch Angabe einer einschränkenden Bedingung innerhalb der WHERE-Klausel einer SELECT-Anweisung können die Tabelleninhalte hinsichtlich einer Bedingung gefiltert werden.

Beispiel 49 liefert nur die abgespeicherten Werte für Geburtsdatum (BDATE) und Adresse (ADDRESS) derjenigen Mitarbeiter, deren Vornamen (FNAME) *John* ist.

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME="John";
```

Beispiel 49: Einschränkung der Anfrage

Durch die Filterung der Ergebnistupel werden alle diejenigen Datenbankeinträge, welche die getroffene Bedingung nicht erfüllen ausgeblendet. Das Ergebnis kann (sofern SELECT * ... gewählt wurde) zwar in der Anzahl Spalten mit der Ursprungsrelation übereinstimmen, wird dies jedoch typischerweise (d.h. außer im Falle, daß alle Tupel der Relation die formulierte Bedingung erfüllen) nicht tun.
Dieser Vorgang wird als *Selektion* bezeichnet.

Definition 43: *Selektion*
Die Selektion blendet einzelne Werteinträge aus..

Als relationale Operatoren für Vergleichstests zwischen zwei (möglicherweise einelementigen) Mengen stehen zur Verfügung:

Operator	Funktion	Bemerkung
=	Gleichheitstest	
<>	Test auf Ungleichheit	Teilweise (auch in MySQL!) ist der Standardoperator durch != ersetzt
<	Test auf kleiner	Liefert bei Zeichenkettendatentypen alle lexikalisch „kleineren“, d.h. diejenigen die in der alphabetischen Sortierung früher auftreten. Ebenso liefert der Operator bei der Anwendung auf Datumsdatentypen die kalendarisch früheren.
<=	Kleiner oder gleich	
>	Größer	
>=	Größer oder gleich	
IS NULL	Testet ob eine Spalte NULL enthält	
IS NOT NULL	Testet ob eine Spalte nicht NULL enthält	
BETWEEN	Testet ob ein Wert in vorgegebenen Grenzen liegt	
IN	Testet ob ein Wert innerhalb einer vorgegebenen Menge liegt	

Neben den einfachen Vergleichsoperationen können durch den LIKE-Operator unscharfe musterbasierte Suchen ausgedrückt werden.
Die Musterausdrücke werden dabei aus den tatsächlich in der Ergebnisemenge erwarteten Zeichen ergänzt um Metazeichen mit besonderer Bedeutung zusammengesetzt. Hierbei stehen „%“ zur Stellvertretung einer (möglicherweise leeren) Menge beliebiger Zeichen und „_“ zur Stellvertretung genau eines Zeichens zur Verfügung.

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME LIKE "J%";
```

Beispiel 50: Musterbasierte Anfrage I

Die Anfrage aus Beispiel 50 liefert die Werte der Spalten BDATE und ADDRESS aller Mitarbeiter deren Name (FNAME) mit einem „J“ beginnt.
Die Anfrage aus Beispiel 51 beschränkt die Suche zusätzlich auf diejenigen Namen, deren vorletztes Zeichen ein „e“ ist, d.h. diejenigen Einträge für die nach dem „e“ nur noch genau ein beliebiges Zeichen auftritt.

```
SELECT BDATE, ADDRESS
FROM EMPLOYEE
WHERE FNAME LIKE "J%e_";
```

Beispiel 51: Musterbasierte Anfrage II

Die Variante aus Beispiel 52 extrahiert alle Spalteninhalte der Mitarbeitertabelle deren Name aus genau fünf Zeichen besteht.

```
SELECT *
FROM EMPLOYEE
WHERE FNAME LIKE "____";
```

Beispiel 52: Musterbasierte Anfrage III

Kombination von Einzelbedingungen

Zur Selektion nach mehreren Bedingungen können diese mit den logischen Operationen AND, OR und NOT kombiniert werden.

Die Anfrage aus Beispiel 53 liefert die Namen aller Mitarbeiter, die in „Houston“ wohnen und weniger als 50000 verdienen.

```
SELECT FNAME
FROM EMPLOYEE
WHERE ADDRESS LIKE "%Houston%" AND SALARY < 50000;
```

Beispiel 53: Kombination von Bedingungen

Mittels der Verknüpfungsoperatoren können auch einige der zuvor gezeigten Vergleichsoperatoren ausgedrückt werden.

Vergleichsoperator	Alternative Schreibweise mit Bedingungsverknüpfung
a BETWEEN b and c	(a >= b) AND (a <= c)
x IN (a, b, c)	(x = a) OR (x = b) OR (x = c)

Für die Kombinationsoperatoren gilt, aufgrund der Möglichkeit des Auftretens von NULL-Werten, die dreiwertige Logik:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Kombination von Anfrageergebnissen

In manchen Fällen ist es gewünscht das Ergebnis eigenständiger Anfragen zu einem Ergebnis zu kombinieren. Hierzu kann das UNION-Schlüsselwort zur Verbindung der Einzelanfragen.

```
CREATE TABLE Artikel(
        ArtNo           VARCHAR(4) PRIMARY KEY,
        Bezeichnung VARCHAR(10) NOT NULL,
        Preis                   DECIMAL(5,2) NOT NULL);
CREATE TABLE Sonderpreise(
        ArtikelNo       VARCHAR(4) PRIMARY KEY,
        Bezeichnung     VARCHAR(10) NOT NULL,
        Sonderverkaufsgrund VARCHAR(20));

INSERT INTO Artikel VALUES("2222", "Blitz Superrein", 19.99);
INSERT INTO Artikel VALUES("1111", "Wusch Superfein", 99.95);

INSERT INTO Sonderpreise VALUES("4444", "Kratzweich", "Wasserschaden");
INSERT INTO Sonderpreise VALUES("3333", "Glanz Extraweiss", NULL);

SELECT ArtNo
FROM Artikel
UNION
SELECT ArtikelNo
FROM Sonderpreise;
```

Beispiel 54: Kombination mittels UNION

Die Anfrage aus Beispiel 54 erstellt zunächst zwei Tabellen und fügt einige Daten ein. Anschließend werden die Artikelnummern (Spalte ArtNo bzw. ArtikelNo) beider Tabellen angefragt und das Ergebnis mittels UNION zu einer Resultattabelle kombiniert.

Voraussetzung der Kombinierbarkeit ist die Typgleichheit der selektierten und zu vereinigenden

Attribute (im Beispiel beide vom Typ VARCHAR(4)).

Implizit führt die Verwendung von UNION sowohl die Sortierung der Ergebnissmenge als auch die Duplikatentfernung daraus herbei.

Verbünde

Häufig besteht der Wunsch Werte aus verschiedenen Tabellen nicht nur gemeinsam abzufragen und anzuzeigen, sondern auch inhaltlich in Beziehung zu setzen.

Die Anfrage aus Beispiel 55 versucht durch Abfrage der Tabellen EMPLOYEE und DEPARTMENT die Namen der Mitarbeiter und die (in der anderen Tabelle abgelegte) Bezeichnung Abteilung zu ermitteln die sie beschäftigen.
Aufgrund der Bildung des kartesischen Produkts werden jedoch alle theoretisch möglichen Kombinationen geliefert und nicht die Untermenge der tatsächlich existierenden Paarungen.

```
SELECT FNAME, DNAME
FROM DEPARTMENT, EMPLOYEE;
```

Beispiel 55: Fehlerhafte Verbundbildung

Durch (geschickte) Nutzung der WHERE-Bedingung, die Werte aus beiden Tabellen miteinander in Beziehung setzt, gelingt jedoch die gewünschte Ermittlung:

```
SELECT FNAME, DNAME
FROM DEPARTMENT AS d, EMPLOYEE as e
WHERE d.DNUMBER = e.DNO;
```

Beispiel 56: Innerer Verbund

Das Beispiel 56 liefert lediglich diejenigen Tupel, für die das in EMPLOYEE abgespeicherte DNO-Attribut einen Wert enthält, der auch in der Spalte DNUMBER der Tabelle DEPARTMENT auftritt.

Definition 44: *Innerer Verbund*
Ein Innerer Verbund enthält die selektierten Daten aller beteiligten Tabellen, welche die formulierte Einschränkungsbedingung erfüllen.

Der SQL-Standard gibt für diese besondere Anfrageform eine eigene Syntax vor:

```
SELECT FNAME, DNAME
FROM DEPARTMENT AS d INNER JOIN EMPLOYEE AS e
ON d.DNUMBER = e.DNO;
```

Beispiel 57: Innerer Verbund in Standardnotation

Die Bildung von Verbünden ist nicht auf die Angabe verschiedener Tabellen beschränkt, sondern kann auch durch mehrfache Bezugnahme auf dieselbe Tabelle geschehen:

```
SELECT e1.FNAME as Chef, e2.FNAME as Mitarbeiter
FROM EMPLOYEE AS e1, EMPLOYEE AS e2
WHERE e1.SSN = e2.SUPERSSN;
```

Beispiel 58: Innerer Verbund unter mehrfacher Nutzung derselben Tabelle

Beispiel 59 zeigt ein Beispiel der Bildung eines inneren Verbundes unter Zugriff auf drei Tabellen. Die Anfrage liefert die Familiennamen (Tabelle EMPLOYEE) sowie die Abteilungen denen der Mitarbeiter zugeordnet ist (aus Tabelle DEPARTMENT) sowie die durch den Mitarbeiter bearbeiteten Projekte (Tabelle PROJECT).
Die Tabelle PROJECT kann jedoch nicht direkt in den Verbund einbezogen werden, da sie über keine geeigneten Attribute (d.h. Attribute die mit derselben Semantik in einer der beiden anderen Tabellen auftreten) verfügt. Daher wird zusätzlich die Tabelle WORKS_ON in die Anfrage miteinbezogen, weil sie mit dem Attribut ESSN ein Attribut bietet, welches die in EMPLOYEE enthaltene Attribut SSN als Fremdschlüssel beinhaltet. Ausgehend hiervon kann eine Bedingung

unter Einbezug von PROJECT formuliert werden.

```
SELECT FNAME, DNAME, PNAME
FROM EMPLOYEE AS e, DEPARTMENT AS d, PROJECT AS p, WORKS_ON AS w
WHERE d.DNUMBER = e.DNO AND e.SSN = w.ESSN AND w.PNO = p.PNUMBER;
```

Beispiel 59: Innerer Verbund dreier Tabellen

Für Verbünde ist die Bildung durch ausschließliche Nutzung des Gleichheitsoperators innerhalb der WHERE-Klausel keineswegs zwingend, wenngleich diese sog. *Equi Joins* eine häufige Anwendungsform darstellen.
Beispiel 1 zeigt einen durch Nutzung des kleiner-Operators gebildeten Verbund, der alle Abteilungen enthält, in denen ein Mitarbeiter (noch) nicht arbeitet und deren Abteilungsnummer größer ist als die Nummer der Abteilung welcher der Mitarbeiter gegenwärtig zugeordnet ist. (Mögliche semantische Deutung: Liste möglicher Beförderungen, sofern größere Abteilungsnummern einen Aufstieg codieren.

```
SELECT FNAME, DNAME
FROM EMPLOYEE JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER < EMPLOYEE.DNO;
```

Beispiel 60: Non-Equi-Join

Äußere Verbunde

Neben der Möglichkeit durch innere Verbünde Tupel die über Attribute mit übereinstimmenden Wertbelegungen zu selektieren besteht durch *äußere Verbünde* die Möglichkeit neben den Tupeln mit übereinstimmenden Werten alle Tupel einer am Verbund beteiligten Tabelle vollständig zu selektieren.

Definition 45: *Äußerer Verbund*
Ein Äußerer Verbund enthält die selektierten Daten aller beteiligten Tabellen, welche die formulierte Einschränkungsbedingung erfüllen, sowie alle Daten der „äußeren“ Tabelle.
Die nicht mit Werten belegbaren Felder werden durch NULL aufgefüllt.

Konzeptionell wird zwischen *linken* und *rechten Äußeren Verbünden* unterschieden. Die „Seite“ des Verbundes gibt diejenige beteiligte Tabelle an, die im Rahmen der Verbundbildung vollständig ausgegeben wird.
Beispiel 61 zeigt ein Beispiel eines linken Äußeren Verbundes, Beispiel 62 illustriert einen rechten äußeren Verbund.

```
INSERT INTO EMPLOYEE VALUES("John", "X", "Doe", "999999999", "1965-03-04", "42 XYZ
Street", "M", 50000, NULL, NULL);

SELECT FNAME, DNAME
FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 61: Linker Äußerer Verbund

Das Beispiel fügt zunächst einen Tupel zur Tabelle EMPLOYEE hinzu, der keiner Abteilung zugeordnet ist. In einem Inneren Verbund erscheint dieser Tupel daher nicht. Der linke Äußere Verbund des Beispiels hingegen umfaßt alle Tupel aus *EMPLOYEE* sowie die Werte der hinsichtlich der Bedingung DEPARTMENT.DNUMBER = EMPLOYEE.DNO ermittelten Übereinstimmungen in DEPARTMENT.
Für die nicht ermittelbaren Übereinstimmungen werden NULL-Werte erzeugt.

```
INSERT INTO DEPARTMENT VALUES("New Dept.", 0, 888665555, NULL);

SELECT FNAME, DNAME
FROM EMPLOYEE RIGHT OUTER JOIN DEPARTMENT
ON DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 62: Rechter Äußerer Verbund

Auch das Beispiel 62 zum rechten Äußeren Verbund fügt zunächst einen Datensatz ein; diesmal in die Tabelle DEPARTMENT, der zu keinem Tupel in EMPLOYEE in Beziehung steht. Analog dem linken Äußeren Verbund liefert der rechte Äußere Verbund alle Tupel der rechtsstehenden Tabelle (DEPARTMENT) sowie die mit *EMPLOYEE* übereinstimmenden.

Kreuzverbund

Der Kreuzverbund liefert alle gemäß den Gesetzen des kartesischen Produkts bildbaren Kombinationen aus Tupeln der beitragenden Relationen:

```
SELECT DNAME, PNUMBER
FROM DEPARTMENT CROSS JOIN PROJECT;
```

Beispiel 63: Kreuzverbund

Das Beispiel entspricht damit im Ergebnis der Anfrage aus Beispiel 43.

Wird beim Kreuzverbund eine Bedingung angegeben, so entspricht er dem Inneren Verbund. Das Ergebnis der Anfrage aus Beispiel 64 ist daher identisch zum inneren Verbund aus Beispiel 56.

```
SELECT FNAME, DNAME
FROM DEPARTMENT CROSS JOIN EMPLOYEE
WHERE DEPARTMENT.DNUMBER = EMPLOYEE.DNO;
```

Beispiel 64: Kreuzverbund mit Bedingung

Artikel von Satya Komatineni: The Effective Use of Joins in Select Statements

Sortierungen

Zur Sortierung hinsichtlich einer oder mehrerer Spalten der als Anfrageergebnis ermittelten Tabelle steht die ORDER BY-Klausel zur Verfügung.
Beispiel 65 zeigt die Anwendung zur lexikalischen Sortierung:

```
CREATE TABLE Person(
        Vorname VARCHAR(10),
        Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");

SELECT *
FROM Person
ORDER BY Nachname;
```

Beispiel 65: Sortierung

Ist die Sortierung bezüglich mehrerer Attribute, d.h. Sortierung innerhalb eines gleicher Attributwerte hinsichtlich eines anderen Attributs, gewünscht, so können auch mehrere Sortierattribute in der ORDER BY-Klausel versammelt werden.
Zusätzlich zeigt das Beispiel die Kurzschreibweise, welche die zu sortierenden Attribute nicht namentlich benennt, sondern nur hinsichtlich ihrer Position innerhalb der SELECT-Klausel referenziert.

```
CREATE TABLE Person(
        Vorname VARCHAR(10),
        Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");

SELECT *
FROM Person
ORDER BY 2,1;
```

Beispiel 66: Sortierung bezüglich mehrerer Attribute

Vorgabegemäß erfolgt die Sortierung aufsteigend (*ascending*). Die Umkehrung der Sortierreihenfolge kann durch nachstellen der Zeichenfolge DESC (für *descending*) nach dem Namen des Sortierattributes erreicht werden.
Die aufsteigende Vorgabesortierung (ASC) wird üblicherweise nicht ausgeschrieben, ist aber im Beispiel 67 zur besseren Verdeutlichung expliziert.

```
CREATE TABLE Person(
        Vorname VARCHAR(10),
        Nachname VARCHAR(10));

INSERT INTO Person VALUES("Adam", "C-Mann");
INSERT INTO Person VALUES("Cesar", "C-Mann");
INSERT INTO Person VALUES("Berta", "C-Mann");
INSERT INTO Person VALUES("Adam", "A-Mann");
INSERT INTO Person VALUES("Cesar", "A-Mann");
INSERT INTO Person VALUES("Berta", "A-Mann");
INSERT INTO Person VALUES("Adam", "B-Mann");
INSERT INTO Person VALUES("Cesar", "B-Mann");
INSERT INTO Person VALUES("Berta", "B-Mann");

SELECT *
FROM Person
ORDER BY 2 ASC,1 DESC;
```

Beispiel 67: Auf- und Absteigende Sortierung bezüglich mehrerer Attribute

Unteranfragen

Bisher wurden Anfragen lediglich auf Tabellen in ihrer Rolle als in der Datenbank abgelegte Eingabemengen betrachtet. Die relationale Sichtweise erfordert jedoch keineswegs, daß die Eingangswerte einer Anfrage direkt aus der Datenbank gelesen werden müssen. Sie können auch Ergebnis einer weiteren Anfrage sein.

Anfragen die vor einer anderen Anfrage ausgeführt werden müssen um für diese Eingangswerte zu liefern werden daher als *Unterabfragen* (*subqueries* oder *nested queries*) bezeichnet.

Das Beispiel 68 zeigt eine solche Unteranfrage die alle Projektnummern liefert welche Projekten zugeordnet sind die in der durch *Smith* gleiteten Abteilung bearbeitet werden. Eine zweite Unterabfrage des Beispiels liefert alle Nummern von Projekten an denen dieser Mitarbeiter selbst arbeitet.
Die durch diese Abfrage gelieferten Daten (Projektnummern) sind Eingangsdaten in die Ermittlung der Projektnamen.

```
SELECT DISTINCT PNAME
FROM PROJECT
WHERE PNUMBER IN (
   SELECT PNUMBER
   FROM PROJECT AS p, DEPARTMENT AS d, EMPLOYEE AS e
   WHERE e.SSN = d.MGRSSN AND
   d.DNUMBER = p.DNUM AND
   e.LNAME="Smith")
      OR
         PNUMBER IN (SELECT PNO
            FROM WORKS_ON AS w, EMPLOYEE AS e
            WHERE w.ESSN = e.SSN AND
            e.LNAME="Smith");
```

Beispiel 68: Unterabfrage I

Beispiel 69 zeigt den Vergleich eines Einzelwertes (SALARY) mit einer Menge gelieferter Werte. Die Anfrage ermittelt diejenigen Mitarbeiter, deren Einkommen höher liegt als das Einkommen aller Mitarbeit in Abteilung Nummer 5. (Hinweis es wird nicht ermittelt ob das Einkommen größer ist als die Summe aller Einkommen der Mitarbeiter aus Abteilung 5, sondern nur ob das Einkommen größer ist als jedes Einzeleinkommen eines Mitarbeiters aus Abteilung 5.)

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE SALARY > ALL (SELECT SALARY FROM EMPLOYEE WHERE DNO=5);
```

Beispiel 69: Unterabfrage II

Korrelierte Unteranfragen

Eine besondere Form der Unteranfragen stellen solche dar, die sich in ihrer WHERE-Klausel auf die äußere Anfrage beziehen.
Diese Form der Anfrageschachtelung wird auch als *korrelierte Unteranfrage* bezeichnet.

Das Beispiel 70 zeigt eine solche Anfrage, die alle Verwandten (DEPENDENT) ermittelt, die das selbe Geschlecht haben wie der in der Tabelle EMPLOYEE erfaßte Mitarbeiter.

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e
WHERE e.SSN IN (SELECT ESSN
                                        FROM DEPENDENT
                                        WHERE e.SEX = SEX);
```

Beispiel 70: Korrelierte Unterabfrage

Jede korrelierte Unterabfrage kann durch Umschreibung in eine nicht-korrelierte Fassung überführt werden. So lautet die Formulierung des aus Beispiel 70 ohne geschachtelte Unterabfrage:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e, DEPENDENT AS d
WHERE e.SSN = d.ESSN AND
e.SEX = d.SEX;
```

Beispiel 71: Auflösung der korrelierten Unterabfrage

Die Formulierung als geschachtelte Unterabfrage ist damit nicht zwingend notwendig, kann jedoch aus Gründen der Übersichtlichkeit gewünscht sein.

Die nähere Betrachtung der Anfragen aus Beispiel 70 und Beispiel 71 zeigen, daß die aus der Tabelle DEPENDENT angefragten Daten lediglich zur Formulierung der Bedingung, nicht jedoch zur Ausgabe herangezogen werden. Daher läßt sich die Bedingung unter Verwendung des EXISTS-Operators umschreiben zu:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e
WHERE EXISTS (  SELECT *
                                                FROM DEPENDENT
                                                WHERE e.SSN = ESSN AND
                                                e.SEX = SEX);
```

Beispiel 72: Korrelierte Unterabfrage mit EXISTS

EXISTS liefert den Boole'schen Wahrheitswert immer dann, wenn die (Unter-)Abfrage eine nichtleere Menge ist, d.h. Daten enthält.

Anfragen die EXISTS oder IN beinhalten können auch durch linke Äußere Verbünde ausgedrückt werden, wie Beispiel 73 zeigt:

```
SELECT e.FNAME, e.LNAME
FROM EMPLOYEE AS e LEFT JOIN DEPENDENT AS d
ON e.SSN = d.ESSN AND e.SEX = d.SEX
WHERE d.SEX IS NOT NULL;
```

Beispiel 73: Korrelierte Unterabfrage ausgedrückt als linker äußerer Verbund

Eine ähnliche Funktion wie die EXISTS-Operation stellt ANY bereit, jedoch liefert diese die durch die Unterabfrage angefragten Tupel zurück um sie an eine Bedingung zu knüpfen.
Beispiel 74 zeigt die Ermittlung der Namen derjenigen Mitarbeiter, die mehr als irgendein beliebiger Manager verdienen.

```
SELECT FNAME
FROM EMPLOYEE
WHERE SALARY > ANY (SELECT SALARY
   FROM EMPLOYEE
   WHERE SSN IN (SELECT SUPERSSN
      FROM EMPLOYEE));
```

Beispiel 74: Unterabfrage unter Verwendung von ANY

Aggregatfunktionen und Gruppierung

Über die Sortierung hinausgehend ist oftmals ein bestimmte Anordnung der durch eine Anfrage ermittelten Ergebnistupel gewünscht, etwa als inhaltliche Gruppierung.
Gleichzeitig sind oft quantiative Aussagen über Eigenschaften der Resultatmenge --- wie größter oder kleinster Wert sowie Summen- oder Durchschnittsbildung --- gewünscht.

Beispiel 1 zeigt die Ermittlung der Summe aller Gehälter (SQL-Funktion SUM) sowie des Maximal- (MAX), Minimal- (MIN) und Durchschnittsgehalts (AVG) für die Mitarbeiter der *Research*-Abteilung. Die genannten SQL-Funktionen werden als *Aggregierungsfunktionen* bezeichnet, da sie die durch die Abfrage ermittelten Einzelwerte (d.h. die Einträge der Spalte SALARY) jeweils zu genau einer Aussage verdichten.

```
SELECT SUM(SALARY), MAX(SALARY), MIN(SALARY), AVG(SALARY)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME="Research";
```

Beispiel 75: Aggregierungsfunktionen

Mit der Funktion COUNT steht eine Möglichkeit zur Ermittlung der Mächtigkeit einer Tupelmenge zur Verfügung. Beispiel Beispiel 76 zeigt ihre Verwendung zur Ermittlung der Anzahl der Mitarbeiter der mit *Research* bezeichneten Abteilung.

```
SELECT COUNT(*)
FROM EMPLOYEE, DEPARTMENT
WHERE DNO = DNUMBER AND DNAME = "Research";
```

Beispiel 76: Zählfunktion I

Als Argument der COUNT-Funktion kann mit DISTINCT ein Schlüsselwort angegeben werden, welches die ausschließliche Zählung verschiedener Werte erwirkt.
Die Anfrage aus Beispiel Beispiel 77 ermittelt durch Nutzung dieses Schlüsselwortes die Anzahl der verschiedenen Werte in der Spalte SALARY.

```
SELECT COUNT(DISTINCT SALARY)
FROM EMPLOYEE;
```

Beispiel 77: Zählfunktion II

Häufig wird, wie in Beispiel 78 gezeigt, eine Anfrage zur Ermittlung der Anzahl als Unterabfrage formuliert und in der umgebenden Hauptabfrage mit einer Bedingung versehen.

```
SELECT LNAME, FNAME
FROM EMPLOYEE
WHERE (SELECT COUNT(*)
                FROM DEPENDENT
                WHERE SSN=ESSN) >= 2;
```

Beispiel 78: Eingebettete Zählfunktion

Neben den bisher gezeigten aggregierten Aussagen über eine Gesamtmenge besteht oftmals der Wunsch nach von Ermittlung Aussagen dieses Stils über bestimmte Werteklassen innerhalb der betrachteten Gesamtmenge. Hierzu dienen Gruppierungen der Ausgangsmenge, auf welche dann die verschiedenen Aggregierungsfunktionen separat angewandt werden können.
Beispiel Beispiel 79 zeigt dies für die Ermittlung der Mitarbeiteranzahl pro Abteilung sowie der Berechnung des abteilungsinternen Durchschnittsgehalts.

```
SELECT d.DNAME AS "Abteilung", COUNT(*) AS "Anzahl Mitarbeiter", AVG(SALARY) AS
"Durchschnittsgehalt"
FROM EMPLOYEE AS e, DEPARTMENT AS d
WHERE e.DNO = d.DNUMBER
GROUP BY DNO;
```

Beispiel 79: Gruppierung

Zur Realisierung wird die GROUP BY-Klausel verwendet, welche die Angabe eines oder mehrerer Attribute zuläßt anhand der die selektierte Menge partitioniert werden soll.

Die Anfrage des Beispiels Beispiel 80 zeigt die Nutzung einer Verbundbedingung innerhalb einer Gruppierungsanfrage, die Projektnummer und -name sowie vermöge der COUNT-Funktion die Anzahl der das Projekt bearbeitenden Mitarbeiter ermittelt.

```
SELECT PNUMBER, PNAME, COUNT(*) AS "Anzahl Mitarbeiter"
FROM PROJECT, WORKS_ON
WHERE PNUMBER = PNO
GROUP BY PNUMBER, PNAME;
```

Beispiel 80: Gruppierung mit Verbundbedingung

Durch zusätzliche Angabe der HAVING-Klausel kann die Menge der Gruppierungsresultate mittels einer Bedingung beschränkt werden.
So ermittelt die Anfrage aus Beispiel 81 dieselben Resultat wie die in Beispiel 80 gezeigte, jedoch nur für Projekte deren Mitarbeiteranzahl größer 2 ist.

```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON
WHERE PNUMBER = PNO
GROUP BY PNUMBER, PNAME
HAVING COUNT(*) > 2;
```

Beispiel 81: Bedingte Gruppierung

Die formulierte Beschränkung wirkt sich nicht auf die zur Berechnung herangezogene Grundgesamtheit, sondern lediglich auf die Ausgabe der Gruppierungsergebnisse aus, die vor der Auswertung der in der HAVING-Klausel formulierten Bedingung berechnet werden müssen.
Zur Beschränkung der zur Berechnung heranzuziehenden Grundgesamtheit steht auch unter Nutzung der GROUP BY-Klausel der durch WHERE formulierte Bedingungsteil der SELECT-Anfrage zur Verfügung.

```
SELECT PNUMBER, PNAME, COUNT(*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE PNUMBER = PNO AND SSN = ESSN AND DNO=5
GROUP BY PNUMBER, PNAME;
```

Beispiel 82: Beschränkung der Gruppierungseingangsdaten

Gruppierungsschritte können auch in Unterabfragen auftreten, wie das Beispiel 83 zur Ermittlung des Abteilungsnamens und der Anteil der darin arbeitenden Personen mit einem Gehalt über 40000 für alle Abteilungen mit mindestens 2 Mitgliedern zeigt:

```
SELECT DNAME, COUNT(*)
FROM DEPARTMENT, EMPLOYEE
WHERE DNUMBER = DNO AND SALARY > 4000 AND DNO IN (
                SELECT DNO
                FROM EMPLOYEE
                GROUP BY DNO
                HAVING COUNT(*) > 2)
GROUP BY DNUMBER;
```

Beispiel 83: Gruppierung in Unterabfrage

Der Datenmanipulationsteil von SQL

Neben den bisher betrachteten Eigenschaften der Sprache SQL zur Definition von Datenbankstrukturen und zur Abfrage von Datenbankinhalten stehen auch Befehle zur Manipulation in Form von Einfüge-, Aktualisierung- und Löschoperationen zur Verfügung.

Der Einfügebefehl INSERT

Zum Hinzufügen neuer Tupel in eine bestehende Tabelle durch Angabe von Werten für einen oder mehrere Spalten dieser Tabelle wird der Befehl INSERT anboten.
Typischerweise wird dieser Befehlstyp zum Einfügen neuer Datensätze in bestehende Tabellen laufender Applikationen, ebenso wie zur Übernahme kompletter Datenbestände aus existierenden Datenquellen oder zur Neuladung einer Datenbank im Rahmen der Wiederherstellungsprozesses nach einem Systemausfall mit Datenverlust verwendet.

Die allgemeine Syntax des INSERT-Ausdruckes lautet:

```
INSERT INTO tbl_name (col_name,...)? VALUES(constant|NULL ...)
```

```
INSERT INTO EMPLOYEE VALUES(
        'John',
        'B',
        'Smith',
        123456789,
        '1965-01-09',
        '731 Fondren, Houston, TX',
        'M',
        30000,
        333445555,
        5);
```

Beispiel 84: Einfügen eines vollständigen Tupels

Beispiel 84 zeigt den Befehl zur Erzeugung eines neuen Eintrages in der Tabelle EMPLYOEE der Demodatenbank. Die Aufzählung der einzufügenden Werte ist vollständig, d.h. für jede Spalte der Tabelle wird explizit ein konstanter Wert angegeben. Per Konvention müssen alle nichtnumerischen Werte in einfache oder doppelte Hochkommata eingeschlossen werden. Hierunter fallen neben den Zeichenkettentypen auch alle Datumstypen.

Eine Sonderstellung innerhalb der angebbaren Konstanten zur Eintragung stellt die Zeichenkette NULL dar. Sie repräsentiert explizit fehlende Werte, deren Tabelleneinträge entsprechend gekennzeichnet werden. Zur Abgrenzung von der Zeichenkette *NULL* wird diese Angabe nicht in Anführungzeichen eingeschlossen, selbst wenn es sich um eine Spalte eines Zeichenkettentypen handelt.
Beispiel 85 zeigt eine exemplarische Befehlskonstruktion:

```
INSERT INTO EMPLOYEE VALUES(
        'James',
        'E',
        'Borg',
        888665555,
        '1937-11-10',
        '450 Stone, Houston, TX',
        'M',
        55000,
        NULL,
        1);
```

Beispiel 85: Einfügen eines vollständigen Tupels mit NULL-Wert

Neben der Möglichkeit vollständige Tupel einzufügen, kann durch explizite Angabe der einzufügenden Spalten auch eine partielle Befüllung des neu erzeugten Tupels vorgenommen werden.
Für die im INSERT-Befehl nicht angegebenen Spalten wird der spezifizierte Vorgabewert oder NULL eingefügt.
Beispiel 86 zeigt dies exemplarisch anhand des Einfügens der drei Attribute FNAME, LNAME und SSN. Gleichzeitig stellt das Beispiel auch heraus, daß bei expliziter Angabe der einzufügenden Spalten die gewählte Reihenfolge von der in der Tabelle realisierten abweichen kann.

```
INSERT INTO EMPLOYEE (LNAME, FNAME, SSN) VALUES(
        'Doe',
        'John',
        912873465);
```

Beispiel 86: Einfügen eines unvollständigen Tupels

Prinzipiell kann jedes Element der Potenzmenge der Attribute einer Relation eingefügt werden. Es muß jedoch zwingend einen Wert für das Primärschlüsselattribut enthalten, da hierfür der Wert NULL nicht gesetzt werden darf.

Der Aktualisierungsbefehl UPDATE

Zur Aktualisierung von Werten innerhalb bestehender Datenbankeinträge bietet der SQL-Sprachumfang den Befehl UPDATE an, der es gestattet frei wählbare Mengen von Tupeln einer Tabelle zu modifizieren.

Die allgemeine Syntax des Befehls lautet:

```
UPDATE tbl_name SET col_name=expression, ... [WHERE search_condition]
```

Beispiel Beispiel 87 zeigt den Befehl zur null-Setzung aller in der Tabelle EMPLOYEE verwalteten Geburtsdaten (BDATA):

```
UPDATE EMPLOYEE SET BDATE=NULL;
```

Beispiel 87: Modifikation aller Tupel durch Setzen eines konstanten Wertes

Neben der Eintragung von Konstanten können auch neue Inhalte aus den Bisherigen errechnet werden. So zeigt Beispiel 88 eine Aktualisierung, die das Gehalt (SALARY) aller Mitarbeiter um zehn Prozent erhöht:

```
UPDATE EMPLOYEE SET Salary=Salary*1.1;
```

Beispiel 88: Modifikation aller Tupel durch Setzen eines berechneten Wertes

Durch Nutzung der, identisch zum SELECT-Ausdruck aufgebauten, WHERE-Klausel kann die Menge der von der Änderung betroffenen Datensätze eingeschänkt werden.
Das Beispiel Beispiel 89 ändert in allen Einträgen, deren LNAME auf Zelaya lautet den Wert zu Jones.
Die Anzahl der betroffenen Tupel ist durch den UPDATE-Ausdruck nicht festlegbar, sondern richtet sich ausschließlich nach der durch die WHERE-Klausel selektierten Eintragsmenge.

```
UPDATE EMPLOYEE SET LNAME='Jones'
WHERE LNAME='Zelaya';
```

Beispiel 89: Modifikation von Tupeln

Durch die Nutzbarkeit der vollständigen Möglichkeiten der aus dem SELECT-Befehl bekannten Mächtigkeit der WHERE-Klausel lassen sich selbst komplexe Aktualisierungen realiseren.
Beispiel 90 zeigt führt die Erhöhung der Gehälter derjenigen Mitarbeiter durch, die Abteilungen zugewiesen sind, die mehr als zwei Projekte bearbeiten.

```
UPDATE EMPLOYEE SET SALARY=SALARY*1.1 WHERE DNO IN
        (SELECT DNUMBER
        FROM PROJECT AS p, DEPARTMENT AS d
        WHERE d.DNUMBER=p.DNUM
        GROUP BY 1
        HAVING COUNT(*) > 2);
```

Beispiel 90: Modifikation von Tupeln (Ermittlung der betroffenen Tupel durch Subanfrage)

***Der Löschbefehl** DELETE*

Zur Löschung von verwalteten Tupeln aus einer Tabelle existiert der DELETE-Befehl, der die betroffenen Datensätze ohne weite Nachfrage entfernt.

Seine allgemeine Syntax lautet:
`DELTE FROM tbl_name [WHERE search_condition]`

Die einfachste Ausprägung der DELETE-Anweisung löscht alle Tupel einer Tabelle:

```
DELETE FROM EMPLOYEE;
```

Beispiel 91: Löschen aller Tupel einer Tabelle

Durch Angabe der WHERE-Klausel können, wie bereits bei UPDATE für die zu aktualisierenden Tupel gezeigt, die zu löschenden Tupel eingegrenzt werden.
So entfernt der Ausdruck aus Beispiel 92 alle Mitarbeiter die in Houston wohnen.

```
DELETE FROM EMPLOYEE
WHERE ADDRESS LIKE "%Houston%";
```

Beispiel 92: Löschen aller Mitarbeiter, die in Houston wohnhaft sind

Durch die Nutzung der expliziten Mengenangabe innerhalb der WHERE-Klausel läßt sich die Menge der zu entfernenden Datensätze statische eingrenzen wie Beispiel 93 zeigt.

```
DELETE EMPLOYEE
WHERE SSN IN (333445555, 888665555, 987987987);
```

Beispiel 93: Löschen bestimmter Datenstätze

Beispiel 94 zeigt die Nutzung einer Unterabfrage zur Ermittlung aller Abteilungen, die nur genau ein Projekt durchführen und anschließenden Löschung dieser Abteilungen aus der Tabelle DEPARTMENT.

```
DELETE FROM DEPARTMENT
WHERE DNUMER IN
        (SELECT DNUM
        FROM PROJECT
        GROUP BY 1
        HAVING COUNT(*) = 1);
```

Beispiel 94: Löschen aller Abteilungen, die nur genau ein Projekt durchführen

Definitionsverzeichnis

Assoziation
Assoziationstyp
Äußerer Verbund
Boyce/Codd-Normalform
Daten
Datenbank
Datenbankmanagementsystem (DBMS)
Datenbanksprache
Datenunabhängigkeit
Dritte Normalform (3NF)
Entität
Entitätstyp
Erste Normalform (1NF)
Fünfte Normalform
Hybrider Entitäts-Assoziationstyp
Index
Information
Innerer Verbund
Kardinalitätsintervall
Konzeptuelles Schema
Logisches Schema
Mehrwertige Abhängigkeit
Metainformation
Modell
NULL-Wert
Physisches Schema
Primärschüssel
Projektion
Referentielle Integrität
Relation
Relationales DBMS
Repräsentation
Repräsentationstyp
Rolle
Schüssel
Selektion
Spezialisierungsassoziationstyp
Superschlüssel
Tabelle
Transitive Abhängigkeit
Triviale mehrwertige Abhängigkeit
Vierte Normalform
Volle funktionale Abhängigkeit
Vollständiges konzeptuelles Schema
Zweite Normalform (2NF)

Schlagwortverzeichnis

Abbildungsverzeichnis

Verzeichnis der Beispiele

Service provided by Mario Jeckle

Generated: 2004-06-08T12:52:04+01:00

Feedback SiteMap

This page's original location: http://www.jeckle.de/vorlesung/datenbanken/script.html

RDF description for this page