



Modul

Datenbanken und Kommunikationsnetze 1

Skript der Vorlesung

Datenbanken 1

[IB 331.a]

Dienstag, 7. März 2017

© Dieses Skript unterliegt dem **Copyright** von Prof. Dr. Ulrich Bröckl und vielen anderen Autoren bzw. deren Verlagen (s. u. A. Literaturliste und Hinweise im Text). Es darf nur von Studierenden der Hochschule Karlsruhe, Fakultät für Informatik und Wirtschaftsinformatik für die Veranstaltung Datenbanken 1 für Studienzwecke verwendet werden. Eine **Weitergabe an Dritte ist nicht zulässig**. Insbesondere das Veröffentlichen im **Internet** ist **verboten**. Wenn Dritte Zugang zu dem Skript brauchen, muss dies schriftlich genehmigt werden (Ulrich.Broeckl@hs-karlsruhe.de).

Inhalt

1 Einführung.....	5
1.1 Allgemeines.....	5
1.2 Motivation.....	6
1.3 Grundbegriffe.....	8
1.4 Überblick über die Architekturen eines DBMS.....	13
2 SQL.....	16
2.1 Übersicht.....	16
2.2 Datentypen.....	17
2.3 Einfache Abfragen.....	18
2.4 Unterabfragen.....	23
2.5 Funktionen.....	27
2.6 Gruppierungen.....	29
2.7 Verbunde.....	31
2.8 Mengenoperationen.....	39
3 SQL DML, DDL.....	41
3.1 SQL-DDL 1.....	41
3.2 SQL DML.....	46
3.3 SQL-DDL 2.....	51
4 Transaktionen.....	56
4.1 Recovery.....	56
4.2 Parallelbetrieb (Concurrency).....	58
4.3 SQL Isolationsebenen.....	63
5 Java Sprachanbindungen.....	66
5.1 JDBC Grundlagen.....	66
5.2 JDBC Architektur.....	67
5.3 Programmieren mit JDBC.....	70
5.4 Transaktionen unter JDBC.....	74
5.5 Sicherheit, Tuning für JDBC.....	75
6 Modellierung.....	81
6.1 Einführung.....	81
6.2 Die Entity-Relationship-Methode (ERM) nach Chen.....	83
6.3 Physikalische Abbildung von Entity-Relationship-Modellen.....	89
7 Normalisierung.....	95
7.1 Einführung.....	95
7.2 Normalformen.....	96
7.3 Zusammenfassung und Beispiele.....	101
8 Null-Werte.....	103
8.1 Einleitung.....	103
8.2 Null-Werte und skalare Ausdrücke.....	104
8.3 Null-Werte und die dreiwertige Logik.....	106
8.4 Null-Werte und JOINs.....	109
8.5 Null-Werte und Aggregatfunktionen.....	110
8.6 Null-Werte und Unterabfragen.....	111
9 Übungen.....	115
9.1 Allgemeines.....	115
9.2 SQL-Aufgaben.....	120
9.3 Java-Aufgaben.....	122
9.4 Datenbankmodellierung.....	123
9.5 Anhang: Datenbankbeschreibung für die SQL- und JDBC-Aufgaben.....	126

1 Einführung

1.1 Allgemeines

1.1.1 Organisation

- [Zur Person ...](#)
- Ziele, Inhalt, Umfang:
Siehe [Lehrveranstaltung Datenbanken 1 Labor \(IB INFB331.a\) im Modulkatalog.](#)

Weitere Informationen (insbesondere zum Labor) siehe bitte

<https://ilias.hs-karlsruhe.de>.

- Zeitplan, Material...
- Laboreröffnung...

1.1.2 Lernziele

- Vermitteln des Wissens zum Verständnis von Datenbanksystemen
- Zielorientierten Realisierung von komplexen Informationssystemen
- Profunde, sichere SQL-Kenntnisse
- Basiskenntnisse in Datenbankmodellierung und Normalisierung

1.1.3 Bücher

[Elmasri09]

Ramez A. Elmasri, Shamkant B. Navathe: "[Grundlagen von Datenbanksystemen](#)", Pearson Studium. Bachelorausgabe. 2009.

(Dies ist der kleinere Band, es gibt noch ein sehr viel ausführliches Datenbanken-Buch dieser Autoren)

[Goll11]

Joachim Goll: "[Methoden und Architekturen der Softwaretechnik](#)", Vieweg+Teubner Verlag, 2011.

[Kemper11]

A. Kemper , A. Eickler: "[Datenbanksysteme: Eine Einführung](#)", Oldenbourg Verlag, 2011.

[Saake03]

Gunter Saake, Kai-Uwe Sattler: "[Datenbanken und Java](#)", Dpunkt Verlag, 2003

[Schicker00]

Edwin Schicker: "[Datenbanken und SQL](#)", Teubner Verlag, 2000.

(Teile des Labors, unter anderem die Beispiel-Datenbank, sind diesem Buch entnommen)

Den Autoren [Schicker00] und [Goll11] sei an dieser Stelle für die vielen fruchtbaren Diskussionen und die sehr kollegiale Überlassung vieler Ideen und Materialien gedankt.

1.2 Motivation

1.2.1 Wozu Datenbanken?

Die Verwaltung von Informationen spielt heutzutage eine immer wichtigere Rolle. Es gibt kaum ein Informationssystem, das in der Lage ist, seine Informationen ohne eine Datenbank zu verwalten. Datenbankverwaltungssysteme oder Datenbankmanagementsysteme (kurz **DBMS**) und die darin verwalteten Datenbanken sind aus den heutigen Projektlandschaften nicht mehr wegzudenken.

Das Datenbankkonzept besteht im Wesentlichen aus diesen zwei Bereichen:

- den Verwaltungsprozessen des **Datenbankmanagementsystems** und
- den gespeicherten Daten in einer **Datenbank**.

Das Datenbankmanagementsystem **abstrahiert** für das Anwendungsprogramm die Daten, ermöglicht den Zugriff auf die Daten und ihre Modifikation und stellt die **Konsistenz** sicher [Kemper11].

1.2.2 Typische Probleme bei Informationsverarbeitung ohne DBMS

- Redundanz und Inkonsistenz
- Beschränkte Zugriffsmöglichkeiten
- Probleme beim Mehrbenutzerbetrieb
- Verlust von Daten
- Integritätsverletzung
- Sicherheitsprobleme
- Hohe Entwicklungskosten für Anwendungsprogramme

1.2.3 Typische Probleme bei Informationsverarbeitung ohne DBMS, Beispiel

- Fehlende/ungeeignete Namenskonventionen
- Schlampige Orthografie, Abkürzungsrecherche
- Unscharfes Wissen
- ...

Acct#	Name	Address	City	State	Zip	Note
5154155	Peter J. Lalonde	40 Beacon St.	Melrose, Mass		02176	ODP
5152335	LaLonde, Peter	76 George 617-210-0824	Boston	YES	MA	2111
5146261	Lalonde. Sofie	40 Bacon Street	Melrose		MA	CHK ID
87121	Pete & Soph Lalond	76 George Road	Boston	MASS		FR Alert
87458	P. Lalonde FBO	S.Lalonde 40 Becon Rd.	Melrose	MA	02	176

Bild 1.1: Wohin geht die Rechnung an die Lalondes?

1.2.4 Aufgaben von DBMS

Bereitstellen der Datenbankfunktionalität

- Datenbankabfragesprache
- Transaktionskonzept/Recovery
- Mehrbenutzer-/Netzwerkfähigkeit
- Gewährleisten der Integrität
 - Korrektheit
 - Vollständigkeit
 - Meist gewährleistet durch Constraints und Trigger

Gewährleisten der Sicherheit

- Integritätsverlust durch
 - "zufällige" Fehleingaben
 - "absichtliche" Fehleingaben
- Gegenmaßnahmen die ein DBMS bietet:
 - Zugangskontrolle
 - Audit-Funktion der Benutzeraktionen

Backup und Recovery:

Automatisches (Online-) Wiederherstellen (Recovery) nach System- oder Hardware-Fehler.

Übliche Fehler:

- Benutzer-Fehler
- (Betriebssystem-) Prozessfehler
- Platten-/Hauptspeicher-/Band-/... -Fehler

Verbergen der Rechnerarchitektur

- Ein-Prozessor-Systeme
- Symmetrische Multiprozessor-Systeme (SMP)

Plattformunabhängigkeit, Skalierbarkeit

- Windows XY, Unix Z, MVS, ...

1.3 Grundbegriffe

1.3.1 Basisbegriffe bei Tabellen

Tafel...

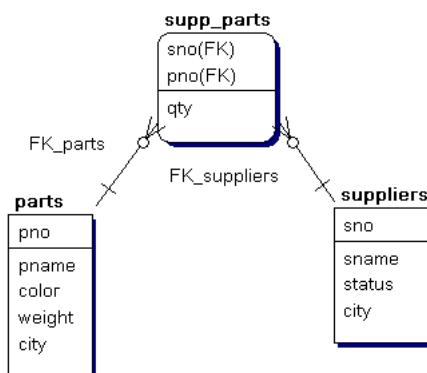
1.3.2 Beispieldatenbank der Vorlesung

Parts				
pno	pname	color	weight	city
P1	Nut	red	12	London
P2	Bolt	green	17	Paris
P3	Screw	blue	17	Rome
P4	Screw	red	14	London
P5	Cam	blue	12	Paris
P6	Cog	red	19	London

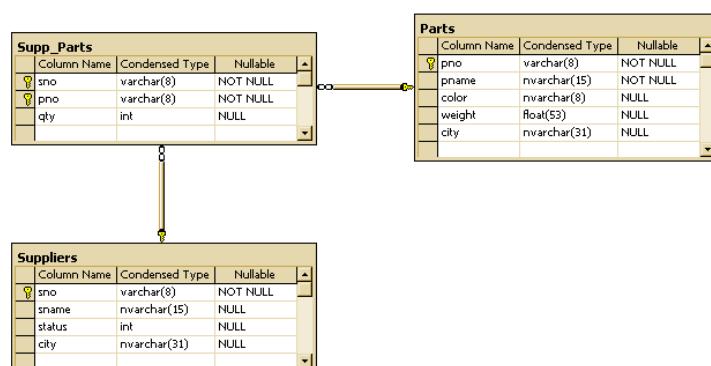
Suppliers		
sno	sname	status
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

Supp_Parts		
sno	pno	qty
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Logisches Schema (mehr zu solchen grafischen Notationen im hinteren Teil des Skriptes)



Physikalisches Schema



1.3.2.1 SQL-Skripts

Herstellerspezifische DDLs und DMLs finden Sie in der Zip-Datei [vor1_db.zip unter ILIAS.](#)

1.3.3 Das relationale Datenmodell

Das Relationale Datenmodell, kurz Relationen-Modell genannt, wurde 1970 von E. F. Codd entwickelt. Es besteht aus drei Teilen: Objekten, Operationen und Regeln.

Relationen (oder Tabellen)

Sie entsprechen den Objekten der selben Klasse in der Objektorientierten Programmierung.

Tupel (oder Datensatz, Record, Zeile)

Ein Tupel entspricht einem konkreten Objekt.

Attribut (Spalte)

Ein Attribut einer Klasse entspricht einem Attribut einer Relation und stellt eine Spalte einer Tabelle dar.

Domäne (Wertebereich)

Der Wertebereich eines Attributs muss festgelegt werden, damit bei Änderungen der Attributwerte Konsistenzprüfungen durchgeführt werden können.

Schlüssel

Schlüssel erlauben den selektiven Zugriff auf einzelne Tupel. Von Bedeutung sind Kandidatenschlüssel (einige Schlüssel), Primärschlüssel (gewählte Kandidatenschlüssel) und Fremdschlüssel.

Operationen

werden zur Abfrage und Änderung auf Relationen sowie zum Anlegen und Löschen von Relationen verwendet.

Regeln

dienen zur Gewährleistung der Konsistenz der Daten. Hierbei geht es zum Beispiel um die Entity-Integrität und die Referenzielle Integrität.

1.3.4 Begriffsdefinitionen

Definition: atomar

Der Begriff atomar (griech.: atomos "unteilbar") kennzeichnet ein Element oder einen Sachverhalt, der nicht weiter in seine Bestandteile zerlegbar ist.

Im Gegensatz dazu stehen die komplexen Strukturen, die aus mehreren Teilbestandteilen aufgebaut sind.

Definition: Domäne

Eine Domäne ist eine Grundmenge, deren Elemente atomar sind.

Beispiele sind die Natürlichen Zahlen; die ganzen Zahlen, die man mit 32 Bit darstellen kann oder auch Zeichenketten.

Allerdings darf in diesen Zeichenketten kein teilbarer Inhalt stehen. Inkorrekt ist also ein Element $s = \text{"Max Huber"}$. Es kann nämlich weiter zerlegt werden in $s_v = \text{"Max"}$, gehörend zu der Domäne aller Vornamen und in $s_n = \text{"Huber"}$, gehörend zu der Domäne aller Nachnamen.

Definition: Tupel

Ein Tupel ist über mehreren Domänen D_1, \dots, D_n (die auch gegebenenfalls gleich sein können) definiert. Jede dieser Domänen hat einen Attributnamen N_1, \dots, N_n . Das Tupel fasst semantisch zusammengehörige atomare Attribute zusammen zu einer komplexeren Struktur.

Herr Huber sieht also so aus: ("Max", "Huber"). Er wird also als Zweier-Tupel gespeichert, bestehend aus einem Attribut über der Domäne aller Vornamen und einem Attribut der Domäne aller Nachnamen.

Definition: Relation

Eine Relation fasst nun alle Tupel in einer Menge zusammen, die über den selben Domänen D_1, \dots, D_n mit den selben Attributnamen N_1, \dots, N_n definiert sind und die selbe semantische Zusammengehörigkeit haben.

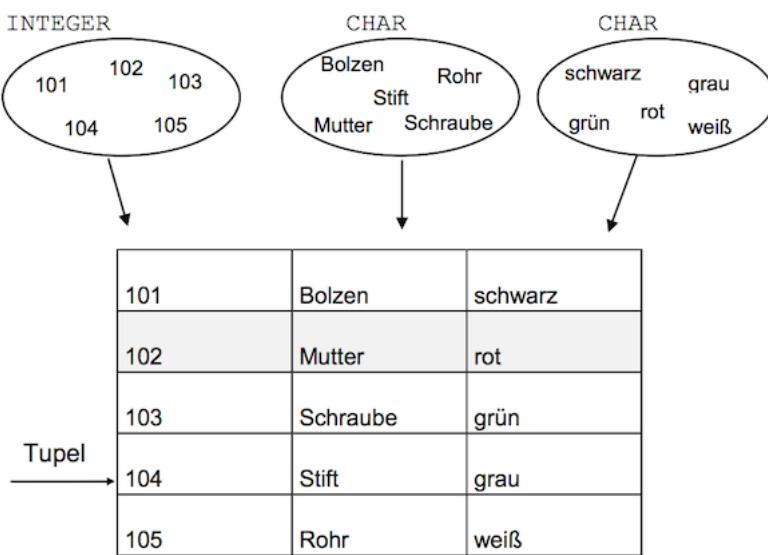


Bild 1.2: Domänen und Relationen

Definition: Grad

Der Grad n bezeichnet die Anzahl der Domänen, über denen die Tupel einer Relation definiert sind.

Definition: Kardinalität

Die Kardinalität bezeichnet die Anzahl der Tupel einer Relation.

Der Grad ist (relativ) fest, er ändert sich in der Regel nur von Release zu Release einer Software. Die Kardinalität ist sehr dynamisch, sie ändert sich stark zur Laufzeit der Software.

Satz: Eindeutigkeit

Es gibt in einer Relation keine zwei Tupel, die in ihren Attributwerten übereinstimmen. Die Reihenfolge, in der die Tupel einer Relation gespeichert werden, ist nicht definiert, d. h. es gibt keine Reihenfolge der Tupel. Genauso wenig ist die Reihenfolge, in der die Attribute einer Relation gespeichert werden, definiert.

Dies liegt in der Tatsache begründet, dass die Relationen *Mengen* von Tupeln sind. In einer Menge gibt es keine doppelten oder gar mehrfachen Elemente.

Insofern ist auch die übliche Darstellung von Relationen durch Tabellen, in der jede Zeile ein Tupel beherbergt, trügerisch: Während in Tabellen, wie zum Beispiel in Excel-Tabellen, es ohne Weiteres möglich ist, mehrere genau gleiche Zeilen einzutragen, ist dies in einer Relation nicht zulässig.

1.3.5 Modellierung von Beziehungen

Wie schon im Bild "Physikalisches Schema" weiter vorne zu sehen war, werden Relationen miteinander verknüpft, d.h. sie gehen Beziehungen ein, sie referenzieren sich gegenseitig. Bei Programmiersprachen können sich Objekte ja auch gegenseitig referenzieren, zum Beispiel durch Zeiger.

Beim Relationenmodell werden diese Beziehungen durch Schlüssel und Fremdschlüssel realisiert.

Kandidatenschlüssel

Eine Zusammenstellung von Attributwerten heißt Kandidatenschlüssel, wenn die Werte, die diese Attribute annehmen, stets ein Tupel **eindeutig identifizieren**. Ein Kandidatenschlüssel kann aus einem oder mehreren Attributnamen bestehen. Eine Relation kann einen oder mehrere Kandidatenschlüssel besitzen. In einer Relation existieren gemäß Definition keine zwei identischen Tupel. Deshalb muss es immer mindestens einen Kandidatenschlüssel - nämlich die Zusammenstellung aller Attributwerte eines Tupels - geben.

Ein Kandidatenschlüssel umfasst die Werte von so vielen Attributnamen wie nötig und so wenig wie möglich.

Primärschlüssel

Ein Primärschlüssel ist ein spezieller Kandidatenschlüssel, der vom Datenbankentwickler zum Primärschlüssel erklärt wird - er kann die Werte mehrerer Attribute umfassen - oder als neues Attribut (z. B. pno) speziell zu diesem Zweck eingeführt werden.

Im Normalfall wird der Primärschlüssel so unter den verfügbaren Kandidatenschlüsseln ausgewählt, dass er aus möglichst wenigen Attributen besteht.

Primärschlüssel werden meist unterstrichen dargestellt.

Surrogatschlüssel

Wird ein Primärschlüssel als neues Attribut eingeführt, wird er auch als Surrogatschlüssel bezeichnet. Er wird also nicht aus den Daten in der Tabelle abgeleitet.

Z. B. in Form einer eindeutigen fortlaufenden Nummer wie Artikelnummer oder Personalnummer, oft auch in Form einer generierten ID, dem sog. Universally Unique Identifier, kurz UUID.

Fremdschlüssel

Fremdschlüssel dienen dazu, jeweils zwei Relationen miteinander zu verknüpfen und die referentielle Integrität der Datenbank (siehe später) zu gewährleisten. Ein Fremdschlüssel einer Relation muss dabei immer einen existierenden Primärschlüssel einer anderen Relation referenzieren.

Fremdschlüsselattributen werden meist mit einem vorangestellten "#" notiert.

Relationenschema

Ein Relationenschema listet die in einem Datenbankschema vorhandenen Relationen textuell auf. Es wird der Name der Relation, gefolgt von den in Klammern gesetzten Attributnamen (ohne Domänenangabe) notiert.

Der (u.U. zusammengesetzte) Primärschlüssel der Relation wird unterstrichen, Fremdschlüssel erhalten ein "#" vor ihren Attributnamen.

Beispiel:

```
PARTS { pno, pname, color, weight, city }
SUPPLIERS { sno, sname, status, city  }
SUPP_PARTS { #sno, #pno, qty }
```

1.3.6 Beispiel einer Transaktion

(Kommt alles noch ganz genau ...)

```
select * from supp_parts
-- S1 verkauft 100 P1-Teile an S2...
begin transaction
    -- S1 100 P1 wegnehmen:
    update supp_parts set qty=200 where sno='S1' AND pno = 'P1'
    -- S2 100 P1 geben:
    update supp_parts set qty=400 where sno='S2' AND pno = 'P1'
    -- OK? Dann ...
commit
```

1.3.7 Einfaches Beispiel für (referentielle) Integrität

(Kommt alles noch ganz genau ...)

```
-- 
-- 1. Beziehungen im Diagramm zeigen, dann
select * from parts
select * from supp_parts
```

```
-- Es gibt keine P1-Teile mehr -> loeschen...
begin transaction;
    delete from parts where pno = 'P1';
    select * from supp_parts;
rollback;
```

1.4 Überblick über die Architekturen eines DBMS

1.4.1 Drei-Schema-Architektur

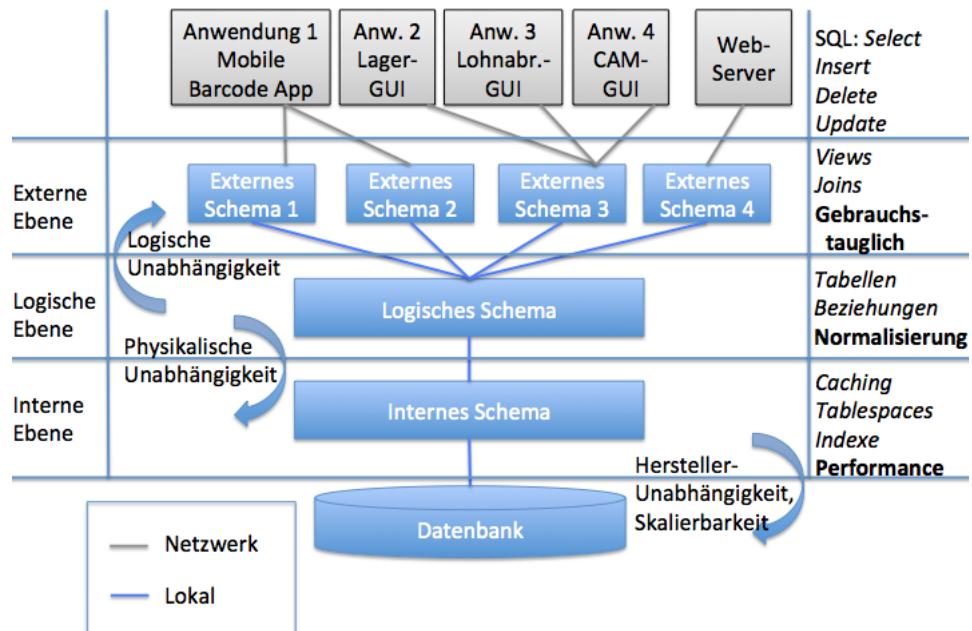


Bild 1.3: Drei-Schema-Architektur

Das ANSI/SPARC -Modell für Datenbanken ermöglicht eine logische und physische Datenunabhängigkeit.

Externes Schema

Das externe Schema der externen Ebene enthält verschiedene Sichten der Nutzer. Die externe Ebene stellt die Sicht auf die Daten und ihre Beziehungen dar. Neben den eigentlichen Daten werden auch die Integritätsbedingungen und Zugriffsrechte beschrieben.

Logisches Schema

Das logische Schema der logischen Ebene beschreibt die logische Struktur der Daten. Das logische Schema - auch konzeptionelles Schema genannt - ist ein logisches Modell der Daten in Tabellenform einschließlich der Beziehungen zwischen den Tabellen. Diese Struktur ist unabhängig von dem darunter liegenden internen Schema und dessen physischer Ausprägung. Damit erreicht man, dass die Programme unabhängig von der physischen Speicherung der Daten sind. Wenn möglich, sollen die Programme sogar unabhängig von der logischen Speicherung der Daten sein.

Internes Schema

Das interne Schema der internen Ebene verwaltet die interne Darstellung der Daten, die Organisation auf einem beliebigen Datenträger sowie die Zugriffsmethoden auf Dateiebene.

Transformationen

Ein externes und internes Schema können aus dem logischen Schema durch Transformationen erzeugt werden. Wird die interne Ebene verändert, müssen nur die Transformationsregeln verändert werden, externe Sichten bleiben bestehen. Die Anwendungen sind isoliert von der physischen Dateiorganisation (physische Datenunabhängigkeit).

Logische Datenunabhängigkeit

Logische Datenunabhängigkeit sagt aus, dass das sogenannte Externe Schema, d. h. die Sichtweise für das Programm und für die interaktiven Nutzer, von dem logischen Schema teilweise unabhängig ist.

Bei bestimmten Änderungen des logischen Schemas müssen die Programme nicht neu kompiliert und gebunden werden. So kann das logische Schema beispielsweise erweitert oder die Attributnamen des logischen Schemas geändert werden, ohne dass ein externes Schema beeinflusst wird. Jeder interaktive Benutzer oder jedes Programm kann ein eigenes externes Schema erzeugen, ohne das logische Schema zu ändern. Vollständige logische Datenunabhängigkeit lässt sich nicht erreichen.

Physische Datenunabhängigkeit

Physische Datenunabhängigkeit bedeutet, dass ein Programm davon unabhängig ist, wie die Daten gespeichert werden.

1.4.2 Datenbanksprachen

Als Datenbanksprache bezeichnet man Computersprachen, die für den Einsatz in Datenbanksystemen entwickelt wurden. Mit Hilfe der Datenbanksprache kommuniziert ein Benutzer oder ein Anwendungsprogramm mit dem Datenbanksystem.

DML

Data Manipulation Language zur Manipulation von Datenbankteilen

Die vier wichtigsten Funktionen hierbei werden auch häufig unter dem Begriff "CRUD" (Create, Read, Update, Delete) zusammengefasst.

DDL

Data Definition Language zur Definition des Datenbankschemata

DCL

Data Control Language zur Transaktions- und Rechteverwaltung und Daten-Administration

1.4.3 Datenbanksprachen in der Drei-Schema-Architektur

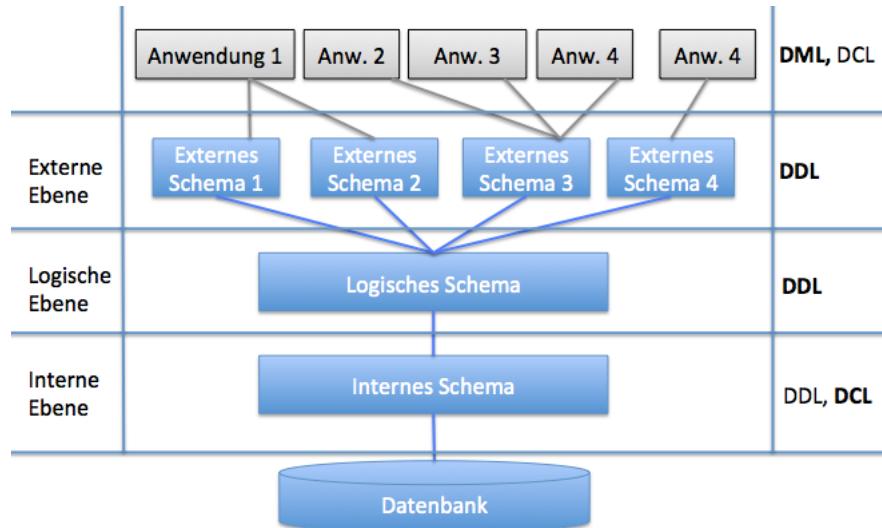


Bild 1.4: Datenbanksprachen in der Drei-Schema-Architektur

1.4.4 Externe Zugriffsmöglichkeiten auf DBMS

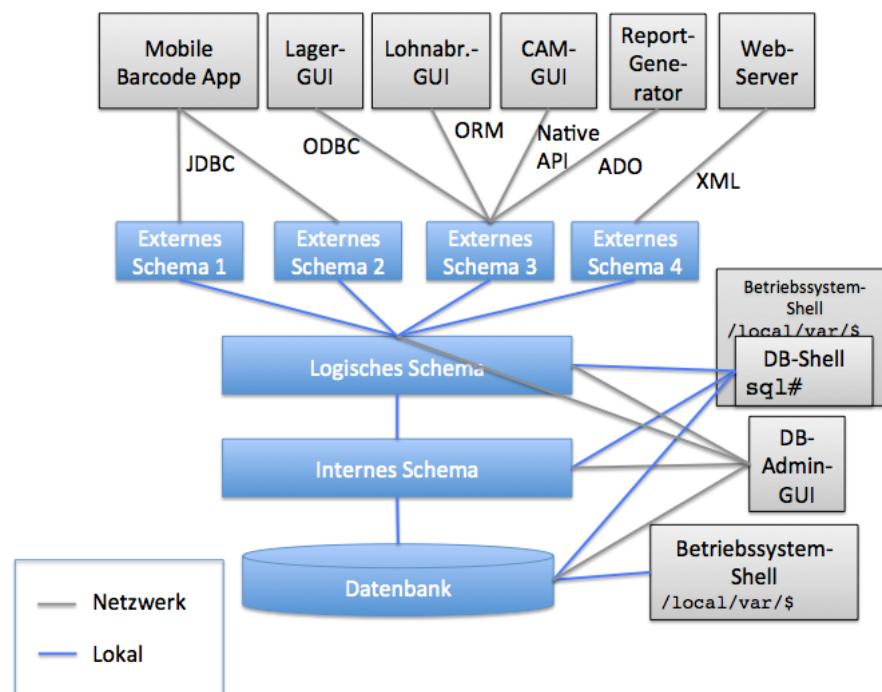


Bild 1.5: Externe Zugriffsmöglichkeiten auf DBMS

2 SQL

2.1 Übersicht

2.1.1 Entstehungsgeschichte

Structured Query Language (SQL) ist die **Standardsprache** für relationale Datenbanken.

Sie wurde als Zugriffssprache für das von dem Mathematiker E. F. **Codd** aufgestellte relationale Datenmodell entwickelt. SQL stellt die praktische Umsetzung der sogenannten relationalen Algebra dar, auf die aus Platzgründen nicht eingegangen werden kann.

Die Meilensteine der Standardisierung sind:

1987 SQL wird erstmals durch ANSI standardisiert

1989 erweiterter SQL Standard (Embedded SQL, Integrität)

1992 SQL-92 (auch SQL2 genannt)

1999 SQL:1999 (auch SQL3 genannt)

2003 SQL:2003

2008 SQL:2008 (noch nicht in allen Datenbanksystemen implementiert)

Die meisten Hersteller relationaler Datenbanken haben die Sprache SQL in ihrem Produkt um **einige Anweisungen** erweitert, da Standards üblicherweise der Entwicklung etwas hinterherhinken.

2.1.2 Eigenschaften von SQL

SQL ist eine Sprache zur:

- Datendefinition,
- Datenprüfung,
- strukturierten Abfrage,
- Aktualisierung,
- Sicherung der Konsistenz und
- Pflege des Datenbestandes.
- SQL ist eine **nichtprozedurale** und **mengenorientierte** Sprache.
- Sie gehört zu den Programmiersprachen der **vierten Generation**. In diesen Sprachen wird nur noch ausgedrückt, wie ein Ergebnis aussehen soll und nicht - wie bei Programmiersprachen der dritten Generation - wie der Rechner zu diesem Ergebnis kommt.
- SQL ist die am meisten verbreitete Sprache für relationale Datenbanken. Dies liegt unter anderem auch daran, dass die ursprünglich von IBM entwickelte Sprache 1987 durch ANSI **standardisiert** wurde.

Wesentliche Eigenschaften von SQL sind:

- SQL entspricht in der Grundstruktur und den verwendeten Schlüsselwörtern der **englischen Sprache** und ist auf sehr wenige Befehle beschränkt und somit leicht zu erlernen.
- SQL ist Teil des Datenbankmanagementsystems (DBMS) und die einzige Schnittstelle, um mit der Datenbank auf logischer oder externer Ebene zu kommunizieren.
- SQL ist sowohl eine einfache Abfragesprache als auch ein Entwicklungswerkzeug für Anwendungsprogrammierer. Sämtliche Anwender, die mit der Datenbank arbeiten, sei es der Datenbankadministrator, ein Anwendungsentwickler oder ein normaler Benutzer, der lediglich Abfragen startet, arbeiten mit demselben Werkzeug.
- Sämtliche relationale Datenbanken, die SQL unterstützen, benutzen dieselbe **standardisierte** Sprache. Die Sprache muss somit vom Benutzer nur einmal erlernt werden. Er kann sie auf jedem beliebigen System - vom PC bis hin zum Großrechner - gleichermaßen einsetzen. Natürlich gibt es auch hier herstellerspezifische Befehle und Syntax. Diese Unterschiede äußern sich aber meist nur bei Zusatzfunktionen.
- SQL ist in der Lage, **heterogene Rechnersysteme** zu verbinden, und schafft somit die Möglichkeit zur Realisierung verteilter heterogener relationaler Datenbanken.
- SQL arbeitet mit Datenbankobjekten und Operatoren auf diesen Objekten. Weiterhin verwaltet SQL Privilegien, die festlegen, wer welche Operation auf welche Objekte ausführen darf.

2.2 Datentypen

2.2.1 Numerische Datentypen

integer (oder auch **integer4**)

In der Regel 4 Byte

smallint (oder auch **integer2**)

In der Regel 2 Byte

numeric(p, q)

Dezimalzahl mit genau p Stellen, davon q hinter dem Dezimalpunkt

decimal(p, q)

Dezimalzahl mit *mindestens* p Stellen, davon q hinter dem Dezimalpunkt

real

Einfach genaue Gleitpunktzahl, meist 4 Byte

float(p)

Gleitpunktzahl, mindestens p Stellen genau ([Bemerkung](#))

double

Doppelt genaue Gleitpunktzahl, meist 8 Byte

2.2.2 Datentypen: Zeichenketten

char[acter](n)

Zeichenketten mit genau n Zeichen, Speicherbedarf ist immer n

character varying(n) oder varchar(n)

Zeichenkette mit höchstens n Zeichen, der Speicherbedarf ist abhängig von der gespeicherten Zeichenkette, i.d.R. also < n

2.2.3 Weitere Datentypen

bit(n), bit varying(n)

date, time, timestamp

blob (raw), clob (binary oder character large object)

2.2.4 Datentypen: Operatoren

- Bei Zahlen: +, -, *, /, %
- Bei Zeichenketten: ||
- Bei Datums-/Zeitangaben: +, -
- Boolesche Verknüpfungen: and, or, not
- Vergleiche: =, >, <, >=, <=, <>
- Typkonversion: cast(<ausdruck> as <datentyp>)

2.3 Einfache Abfragen

Einfache Datenabfrage in SQL

2.3.1 Die SELECT Anweisung

```
SELECT  < expression >
  FROM  < table-name >
    [WHERE  < search-condition >]
    [GROUP BY...[HAVING...]]
    [ORDER BY...]
```

2.3.2 Selektion

Frage/Anforderung: Welche Teile mit welchen Merkmalen gibt es überhaupt?

SQL:

```
SELECT *
  FROM parts
```

Resultat:

```
pno  pname  color  weight  city
----  -----  -----  -----  -----
P1    Nut     red    12.0    London
P2    Bolt    green   17.0    Paris
P3    Screw   blue   17.0    Rome
P4    Screw   red    14.0    London
P5    Cam     blue   12.0    Paris
P6    Cog     red    19.0    London
(6 row(s) affected)
```

2.3.3 Projektion

Frage/Anforderung: Welche Farben haben die Teile?

SQL:

```
SELECT color
FROM parts
```

Resultat:

```
color
-----
red
green
blue
red
blue
red
```

```
(6 row(s) affected)
```

2.3.4 Projektion: DISTINCT

Frage/Anforderung: Welche Farben haben die Teile überhaupt? Keine Mehrfachnennungen bitte ...

SQL:

```
SELECT DISTINCT color
FROM parts
```

Resultat:

```
color
-----
blue
green
red
```

```
(3 row(s) affected)
```

2.3.5 Benutzerdefinierte Spaltennamen

SQL:

```
SELECT pno AS part_no,
       pname AS part_name,
       color AS part_color
  FROM parts
```

Resultat:

```
part_no  part_name  part_color
-----  -----  -----
P1      Nut        red
P2      Bolt       green
P3      Screw      blue
P4      Screw      red
P5      Cam        blue
P6      Cog        red
(6 row(s) affected)
```

Das Schlüsselwort AS kann weggelassen werden, sollte es aber nicht.

2.3.6 Sortierung

Frage/Anforderung: Alle Teile, absteigend nach Farbe, falls gleich, dann nach Name aufsteigend sortiert

SQL:

```
SELECT pno, color
  FROM parts
 ORDER BY
       color DESC,
       pno ASC
```

Resultat:

```
pno  color
----- 
P1   red
P4   red
P6   red
P2   green
P3   blue
P5   blue
(6 row(s) affected)
```

Das Schlüsselwort ASC kann weggelassen werden, sollte es aber nicht.

Keine stabile Sortierung garantiert!

2.3.7 Restriktion

Frage/Anforderung: Welche Teile sind in Paris gelagert?

SQL:

```
SELECT pname, color
  FROM parts
 WHERE city = 'Paris'
```

Resultat:

```
pname  color
----- -----
Bolt   green
Cam    blue
(2 row(s) affected)
```

Man beachte, dass in SQL Zeichenketten in einfache, nicht in doppelte Hochkommata eingeschlossen werden. Es gibt aber Hersteller, die das trotzdem akzeptieren -> schlechte Portabilität.

2.3.8 Intervallangaben

Verknüpfte Restriktion mit Intervallangaben:

Frage/Anforderung: Welche roten Teile wiegen zwischen 13 und 20 kg?

SQL:

```
SELECT pname, weight
  FROM parts
 WHERE color = 'red' AND
       weight BETWEEN 13 AND 20
```

Resultat:

```
pname weight
----- -----
Screw 14.0
Cog   19.0
(2 row(s) affected)
```

Versuchen Sie, Intervallabfragen stets mit BETWEEN zu formulieren, dies erhöht die Lesbarkeit.

Folgende Alternative zu BETWEEN ist daher schlecht:

SQL:

```
SELECT *
  FROM parts
 WHERE color = 'red'
       AND weight >= 13 AND weight <= 20
```

2.3.9 Pattern Matching

Frage/Anforderung: Welche Teilenamen haben wenigstens 2 Buchstaben und fangen mit 'C' an?

SQL:

```
SELECT pno, pname
  FROM parts
 WHERE pname LIKE 'C_%'
```

Resultat:

```
pno      pname
----- -----
P5      Cam
P6      Cog
(2 row(s) affected)
```

Wildcards in SQL: %, _

Entsprechen folgenden Wildcards in Unix, Windows ...: *, ?

Escape-Zeichen: \

Beispiele: SQL:

```
... WHERE file LIKE 'C:\\user\\_tmp\\\\tst.%'
-- praktische Alternative:
... WHERE file LIKE 'C:\\user@_tmp\\tst.%' ESCAPE '@'
```

2.3.10 Berechnete Spalten

Frage/Anforderung: Welche Teile wiegen mehr als 30 Pfund (1 lbs = 0,4536 kg)?

SQL:

```
SELECT pno,
       weight / 0.4536 AS wgt,
       'lbr' AS unit
  FROM parts
 WHERE weight / 0.4536 > 30
 ORDER BY weight
```

Resultat:

```
pno      wgt    unit
----- -----
P4      30.86  lbr
P2      37.47  lbr
P3      37.47  lbr
P6      41.88  lbr
(4 row(s) affected)
```

(Die Spalte unit ist auch eine 'berechnete' Spalte ...)

2.3.11 Interne Reihenfolge bei SELECTs

Warum wird beim vorherigen Beispiel zwei mal gerechnet und nicht die Spalte wgt in der WHERE-Bedingung eingesetzt?

1. Kreuzprodukt über alle Tabellen, die FROM spezifiziert (hier nur eine, später mehr ...)
2. Restriktion: Zeilen bestimmen, welche die WHERE-Bedingung erfüllen (wgt hier unbekannt!)
3. Projektion: Spalten gemäß der SELECT-Klausel herauspicken/umbenennen
4. Gruppenbildung gemäß GROUP BY
5. Gruppen-Restriktion gemäß HAVING
6. Mengenbildung mit anderen SELECTs (z.B. durch UNION)
7. Sortieren gemäß ORDER BY

2.3.12 Übungen

Nr.	Aufgabe
-----	---------

SEL.1 Alle Namen von Teilen, die mehr als 15 kg schwer sind

SEL.2 Alle Namen von Teilen (mit ! Markierung), die mehr als 15 kg schwer sind

SEL.3 Gibt es Hersteller-Städte ohne "R/r"? Wenn ja, welche ...

SEL.4 Alle Teile, die ein ungerades Gewicht haben

SEL.5 Alle Teile, die 13-19 kg wiegen

SEL.6 Alle Teile, die 13, 14, oder 19 kg wiegen

2.4 Unterabfragen

(Subqueries)

Unterabfragen sind in Klammern gesetzte "neue" Abfragen, welche die Hauptabfrage mit Werten versorgen.

Es gibt zwei Varianten:

1. Unterabfragen, die *einen* Wert liefern
(Scalar Subqueries)
2. Unterabfragen, die *mehrere* Werte liefern

Unterabfragen mit einer Ergebniszeile

Frage/Anforderung: Welche Teile liegen in der selben Stadt, in der S1 sitzt?

SQL:

```
SELECT pno, pname
  FROM parts
 WHERE city =
  (SELECT city
    FROM suppliers
   WHERE sno = 'S1')
```

Resultat:

```
pno      pname
----- -----
P1       Nut
P4       Screw
P6       Cog
(3 row(s) affected)
```

- Das Ergebnis Unterabfrage darf nicht mehr als eine Zeile haben
- Falls das Ergebnis leer ist, wird NULL zurückgegeben (dazu später mehr ...)
- Das Ergebnis der Unterabfrage wird mit Attributen der Hauptabfrage verglichen (mit $=, <, \leq, >, \dots$)

Unterabfragen mit mehreren Ergebniszellen

- In der WHERE-Bedingung der Hauptabfrage sind Mengenoperationen nötig, wenn die Unterabfrage mehrere Ergebniszellen liefert
- **IN** prüft, ob ein Wert in der Unterabfrage enthalten ist
- **EXISTS** prüft, ob wenigstens eine Zeile der Unterabfrage eine Bedingung erfüllt
- $\Theta \text{ ANY}$ oder $\Theta \text{ SOME}$ prüfen, ob irgend eine Zeile einen der Operatoren für Mengenvergleiche (Θ (Theta-Operatoren)) erfüllen. Diese sind $=, <, \leq, >, \dots$
- $\Theta \text{ ALL}$ prüft, ob alle Zeilen den Operator Θ erfüllen

Unterabfragen mit IN

Frage/Anforderung: Welche Teile sind in Städten gelagert, in denen auch Händler sitzen?

SQL:

```
SELECT pno, pname, city
  FROM parts
 WHERE city IN
  (SELECT city FROM suppliers)
```

Resultat:

```
pno  pname  city
----  ----
P1   Nut    London
P2   Bolt   Paris
```

```
P4    Screw  London  
P5    Cam    Paris  
P6    Cog    London
```

(5 row(s) affected)

Der Anweisung

```
SELECT *  
  FROM parts  
 WHERE city IN  
 (SELECT city FROM suppliers)
```

entspricht

```
SELECT *  
  FROM parts  
 WHERE city IN  
 ('Athens', 'London', 'Paris')
```

oder auch

```
SELECT *  
  FROM parts  
 WHERE city = 'Athens'  
 OR city = 'London'  
 OR city = 'Paris'
```

2.4.1 Übungen

Nr.	Aufgabe
SUB.1	Alle Teile, die auch ordentlich (>100) verkauft werden

2.4.2 Unterabfragen mit ALL

Frage/Anforderung: Welche Teile sind schwerer als das schwerste blaue Teil?

SQL:

```
SELECT pno, pname, color  
  FROM parts  
 WHERE weight > ALL  
 (SELECT weight FROM parts  
   WHERE color = 'blue')
```

Resultat:

pno	pname	color
P6	Cog	red

2.4.3 Unterabfragen mit ANY

Frage/Anforderung: Wer hat wie viel Schrauben (screw) verkauft?

SQL:

```
SELECT sno, qty
  FROM supp_parts
 WHERE pno = ANY
  (SELECT pno FROM parts
 WHERE pname = 'Screw')
```

Resultat:

sno	qty
S1	400
S1	200
S4	300

(3 row(s) affected)

2.4.4 Unterabfragen mit EXISTS

Frage/Anforderung: Welche Händler sitzen in einer Stadt, die auch Teile hat?

SQL:

```
SELECT sname, city
  FROM suppliers
 WHERE EXISTS
  (SELECT *
    FROM parts
   WHERE city = suppliers.city)
```

Resultat:

sname	city
Smith	London
Jones	Paris
Blake	Paris
Clark	London

(4 row(s) affected)

Hier spricht man auch von einer *korrelierten* Unterabfrage, da die Unterabfrage direkt Bezug auf die Hauptabfrage nimmt.

Da das Attribut `city` in beiden Relationen enthalten ist, muss durch Voranstellen des Namens der Tabelle der Hauptabfrage spezifiziert werden, welches Attribut gemeint ist.

Dabei ist es oft üblich, durch das Schlüsselwort AS den Tabellennamen abzukürzen (im Labor unter Oracle leider *nicht* möglich ...).

SQL:

```
SELECT sname, city
  FROM suppliers AS s
 WHERE EXISTS
( SELECT * FROM parts
  WHERE city = s.city)
```

Mit NOT lassen sich mächtige Abfragen erstellen:

Frage/Anforderung: *Welche Händler sitzen in einer Stadt, die keine Teile hat?*

SQL:

```
SELECT sname, city
  FROM suppliers
 WHERE NOT EXISTS
( SELECT *
    FROM parts
   WHERE city = suppliers.city)
```

Resultat:

```
sname      city
----- -----
Adams      Athens
(1 row(s) affected)
```

2.4.5 Übungen

- | Nr. | Aufgabe |
|-------|--|
| SUB.2 | Alle Teile, die auch ordentlich (>100) verkauft werden als als korrelierte Unterabfrage |
| SUB.3 | Alle Teile, die auch ordentlich (>100) verkauft werden als als Theta-Unterabfrage |

2.5 Funktionen

2.5.1 Arten von Funktionen

Es werden zwei Arten unterschieden:

1. Skalare Funktionen, die sich auf ein (u.U. berechnetes) Attribut beziehen
2. Aggregatfunktionen, die sich auf eine Menge von Zeilen beziehen (Set Functions)

2.5.2 Skalare Funktionen

Mathematische Funktionen:

ABS ACOS ASIN ATAN ATN2 CEILING COS COT EXP FLOOR LOG LOG10 PI
POWER RADIANS **ROUND** SIGN SIN SQRT SQUARE TAN

Funktionen für Zeichenketten:

CHARINDEX/INSTR LEFT LEN(GTH) LOWER **LTRIM** RIGHT **RTRIM** SUBSTRING
UPPER

Funktionen für Zeit- und Datumsmanipulation (leider sehr uneinheitlich bei den Herstellern)

CURRENT DATE CURRENT TIME **CURRENT_TIMESTAMP** **DAY** **MONTH** SYSDATE
TRUNC (d, [fmt]) TO_DATE **YEAR**

2.5.3 Aggregatfunktionen

Diese Funktionen dienen der Berechnung von Summe, Maximum etc. eines Attributs über mehrere Zeilen hinweg. Ohne GROUP BY heißt das: Über alle Zeilen eines SELECTs hinweg.

Beispiele: AVG MAX MIN **SUM** STDEV **COUNT**

Frage/Anforderung: *Größte, kleinste und durchschnittliche Zahl von Teile-Verkäufen?*

SQL:

```
SELECT MIN(qty) AS mini,
       MAX(qty) AS maxi,
       AVG(qty) AS schnitt
  FROM supp_parts
```

Resultat:

```
mini  maxi  schnitt
-----  -----
100    400    258
(1 row(s) affected)
```

NULL-Werte werden nicht mit betrachtet, außer bei COUNT(*)

Frage/Anforderung: *Tabelle anlegen, dabei auch ein paar NULL-Werte hinterlegen. Dann Vergleich COUNT/COUNT(*)*

SQL:

```
CREATE TABLE nullBsp (n INT);
INSERT INTO nullBsp (n) VALUES(NULL);
INSERT INTO nullBsp (n) VALUES(1);
INSERT INTO nullBsp (n) VALUES(2);
SELECT COUNT(*) AS tutti FROM nullBsp;
SELECT COUNT(n) AS really FROM nullBsp;
SELECT AVG(n) schnitt from nullBsp;
-- not allowed:
-- select avg(*) as stutti from nullBsp
-- use cast to obtain real-average:
SELECT avg (CAST(n AS real)) AS
          dschnitt FROM nullBsp
DROP TABLE nullBsp
```

Resultat:

```
...
tutti
-----
3
(1 row(s) affected)

really
-----
2
(1 row(s) affected)

Warning: Null value is eliminated
by an aggregate or other
SET operation.

schnitt
-----
1
(1 row(s) affected)

Warning: Null value ...
dschnitt
-----
1.5
(1 row(s) affected)
```

2.6 Gruppierungen

(GROUP BY)

2.6.1 Gruppierungen

Durch das Schlüsselwort GROUP BY lassen sich Aggregatfunktionen auch auf Teilmengen von Zeilen anwenden. Damit lassen sich sehr einfach mächtige Statistiken berechnen.

Frage/Anforderung: *Wie viele Teile wurden insgesamt durch einen Händler verkauft?*

SQL:

```
SELECT sno, SUM(qty) AS total
  FROM supp_parts
 GROUP BY sno
```

Resultat:

sno	total
S1	1300
S2	700
S3	200

```
S4      900  
(4 row(s) affected)
```

Durch das Schlüsselwort HAVING lassen sich die Teilmengen, die zum Beispiel summiert werden, einschränken. WHERE schränkt einzelne Zeilen ein, HAVING ganze Gruppen von Zeilen.

Frage/Anforderung: Wie viele Teile wurden insgesamt durch einen Händler verkauft? Keinen Händler listen, der nur einmal verkauft hat.

SQL:

```
SELECT sno, SUM(qty) AS total  
  FROM supp_parts  
 GROUP BY sno  
 HAVING count (qty) > 1
```

Resultat:

sno	total
S1	1300
S2	700
S4	900

```
(3 row(s) affected)
```

Frage/Anforderung: Wie viele Teile wurden insgesamt durch einen Händler verkauft? Keinen Händler listen, der nur einmal verkauft hat. Verkäufe kleiner als 200 sollen nicht mit gezählt werden.

SQL:

```
SELECT sno, SUM(qty) AS total  
  FROM supp_parts  
 WHERE qty >= 200  
 GROUP BY sno  
 HAVING count (qty) > 1
```

Resultat:

sno	total
S1	1100
S2	700
S4	900

```
(3 row(s) affected)
```

- Die hinter GROUP BY angegebene Spaltenliste bestimmt die Zeilen, die gruppiert werden
- Da HAVING sich auf die Gruppen bezieht, verwendet es oft Aggregatfunktionen wie COUNT
- Alle Spalten des SELECTs müssen in der GROUP BY-Klausel gebunden werden, außer jenen Spalten des SELECTs, die Aggregatfunktionen sind

SQL:

```
SELECT sno, pno,
       SUM(qty) AS total
  FROM supp_parts
 GROUP BY sno
```

Resultat:

```
Server: Msg 8120, Level 16,
        State 1, Line 1
Column 'supp_parts.pno' is invalid in
the select list because it is not
contained in either an aggregate
function or the GROUP BY clause.
```

2.6.2 Interne Reihenfolge bei SELECTs

1. Kreuzprodukt über alle Tabellen, die FROM spezifiziert (hier nur eine, später mehr ...)
2. Restriktion: Zeilen bestimmen, welche die WHERE-Bedingung erfüllen
3. Projektion: Spalten gemäß der SELECT-Klausel herauspicken/umbenennen
- 4. Gruppenbildung gemäß GROUP BY**
- 5. Gruppen-Restriktion gemäß HAVING**
6. Mengenbildung mit anderen SELECTs (z.B. durch UNION)
7. Sortieren gemäß ORDER BY

2.6.3 Übungen

Nr.	Aufgabe
GRP.1	Teilenummer, Gesamtzahl und Durchschnitt je verkauftem Teil
GRP.2	" ohne nur ein mal verkaufte Teile
GRP.3	" nur für Teil p1, p5
GRP.4	Farben, die einzigartig sind
GRP.5	Teile, die Farben haben, die es nur ein mal gibt

2.7 Verbunde

(JOINS)

Durch das Schlüsselwort JOIN lassen sich mehrere Tabellen verbinden.

Damit kann man insbesondere die im logischen Datenmodell etablierten Primärschlüssel/Fremdschlüssel-Beziehungen in Abfragen auflösen.

2.7.1 JOINS

Alte Syntax: Man kann zwei Tabellen auch durch

- Aufführen mehrere Tabellennamen nach dem FROM Schlüsselwort und
- Verknüpfen des Primärschlüssel/Fremdschlüssel-Paars in der WHERE-Restriktion

verknüpfen

Frage/Anforderung: *Wie lauten die Namen der P1-Lieferanten?*

SQL:

```
SELECT suppliers.sname, supp_parts.pno
  FROM supp_parts, suppliers
 WHERE supp_parts.sno = suppliers.sno
   AND supp_parts.pno = 'P1'
```

Resultat:

```
sname  pno
----- 
Smith  P1
Jones  P1

(2 row(s) affected)
```

Neue Syntax (SQL2-Joins): Hier wird mit den Schlüsselworten JOIN und ON klar spezifiziert, was gemeint ist:

Frage/Anforderung: *Wie lauten die Namen der P1-Lieferanten?*

SQL:

```
SELECT s.sname, sp.pno
  FROM supp_parts sp
 JOIN suppliers s
    ON s.sno = sp.sno
 WHERE sp.pno = 'P1'
```

Resultat:

```

sname  pno
-----
Smith  P1
Jones  P1

(2 row(s) affected)

```

(Hinweis: Neue Syntax geht nicht mit Oracle 8 ...)

2.7.2 JOIN: alter gegen neuen Stil

```

-- SQL:
SELECT      s.sname, sp.pno
FROM        supp_parts AS sp①
          suppliers AS s
WHERE       s.sno = sp.sno
AND         sp.pno = 'P1'

-- SQL2: Klare Trennung zwischen JOIN-Bedingung und Restriktion
SELECT      s.sname, sp.pno
FROM        supp_parts AS sp
INNER JOIN suppliers AS s
ON          sp.sno = s.sno
WHERE       sp.pno = 'P1'

```

Bild 2.1: Vergleich der JOIN-Syntaxen

2.7.3 Wie berechnet ein DBMS einen JOIN?

Das folgende Bild zeigt die Basistabellen, die verbunden werden.

Der Übersichtlichkeit halber wurden in der linken Tabelle im Vergleich zu den vorherigen Beispielen die S5-Zeile weggelassen.

Table Suppliers AS s				
sno	sname	status	city	rowID
S1	Smith	20	London	1
S2	Jones	10	Paris	2
S3	Blake	30	Paris	3
S4	Clark	20	London	4

Table Supp_Parts
AS sp

sno	pno	qty	rowID
S1	P1	300	1
S1	P2	200	2
S1	P3	400	3
S1	P4	200	4
S1	P5	100	5
S1	P6	100	6
S2	P1	300	7
S2	P2	400	8
S3	P2	200	9
S4	P2	200	10
S4	P4	300	11
S4	P5	400	12

Bild 2.2: Basistabellen des JOINS

Zunächst werden die beiden Tabellen durch **Kreuzprodukt**-Bildung Zeile für Zeile miteinander verbunden.

Wenn man nach dem FROM-Schlüsselwort zwei Tabellen - durch Komma getrennt - angibt und die Ergebnismenge nicht in der WHERE-Restriktion einschränkt, passiert genau das.

```
SELECT s.sno AS "s.sno", sp.sno AS "sp.sno"
FROM   supp_parts AS sp, suppliers AS s
```

s.rowID	sp.rowID	s.sno	sp.sno
1	1	S1	S1
1	2	S1	S1
1	3	S1	S1
1	4	S1	S1
1	5	S1	S1
1	6	S1	S1
1	7	S1	S2
1	8	S1	S2
1	9	S1	S3
1	10	S1	S4
1	11	S1	S4
1	12	S1	S4
2	1	S2	S1
2	2	S2	S1
2	3	S2	S1
2	4	S2	S1
2	5	S2	S1
2	6	S2	S1
2	7	S2	S2
2	8	S2	S2
2	9	S2	S3
2	10	S2	S4
2	11	S2	S4
2	12	S2	S4

s.rowID	sp.rowID	s.sno	sp.sno
3	1	S3	S1
3	2	S3	S1
3	3	S3	S1
3	4	S3	S1
3	5	S3	S1
3	6	S3	S1
3	7	S3	S2
3	8	S3	S2
3	9	S3	S3
3	10	S3	S4
3	11	S3	S4
3	12	S3	S4
4	1	S4	S1
4	2	S4	S1
4	3	S4	S1
4	4	S4	S1
4	5	S4	S1
4	6	S4	S1
4	7	S4	S2
4	8	S4	S2
4	9	S4	S3
4	10	S4	S4
4	11	S4	S4
4	12	S4	S4

Bild 2.3: Kreuzprodukt des JOINS

Danach werden die Zeilen selektiert, welche die ON-Bedingung erfüllen

```
SELECT s.sname, sp.pno
  FROM supp_parts sp
  JOIN suppliers s
  ON s.sno = sp.sno
 WHERE sp.pno = 'P1'
```

s.rowID	sp.rowID	s.sno	sp.sno
1	1	S1	S1
1	2	S1	S1
1	3	S1	S1
1	4	S1	S1
1	5	S1	S1
1	6	S1	S1
2	7	S2	S2
2	8	S2	S2
3	9	S3	S3
4	10	S4	S4
4	11	S4	S4
4	12	S4	S4

Bild 2.4: Verknüpfung gemäß ON beim JOIN

Auf die so verbundenen Zeilen wird jetzt wie beim normalen SELECT auch die Restriktion und Projektion angewandt.

Projektion (SELECT s.sname, sp.pno)

Restriktion
(`WHERE sp.pno = 'P1'`)

s.rowID	s.sno	s.sname	sp.rowID	sp.sno	sp.pno
1	S1	Smith		1	S1	P1	
1	S1	Smith		2	S1	P2	
1	S1	Smith		3	S1	P3	
1	S1	Smith		4	S1	P4	
1	S1	Smith		5	S1	P5	
1	S1	Smith		6	S1	P6	
2	S2	Jones		7	S2	P1	
2	S2	Jones		8	S2	P2	
3	S3	Blake		9	S3	P2	
4	S4	Clark		10	S4	P2	
4	S4	Clark		11	S4	P4	
4	S4	Clark		12	S4	P5	

Bild 2.5: Restriktion und Projektion des JOINS

Zusammenfassung:

Schritt 1

Kreuzprodukt aller Tabellen

Schritt 2

Zusammengehörige Zeilen gemäß ON-Bedingung bestimmen

Schritt 3

Restriktion gemäß WHERE

Schritt 4

Projektion, um die gewünschten Spalten zu bestimmen (* liefert alle Spalten aller Tabellen)

2.7.4 Regeln für JOIN-Bedingungen

- Die Typen der verbundenen Spalten müssen kompatibel sein. Aus Performance-Gründen ist es ratsam, dass die Spaltentypen gleich sind.
- Die Spaltennamen brauchen nicht gleich zu sein.
- Man kann auch mit den Theta-Operatoren $>$, $<$, \geq , ... verbinden (Theta-Join).
- Man kann - sollte aber nicht - auch Restriktions-Bedingungen nach dem ON-Schlüsselwort absetzen.
- Die erlaubte Gesamtzahl von verbundenen Tabellen hängt vom DBMS ab.
- Wird mit $=$ verbunden, spricht man auch vom **Equi-Join**, was wohl die häufigste Variante ist.

2.7.5 Natural JOINS

Werden beim JOIN die gleichen Spaltennamen verbunden, kann man bei einigen DBMS auch das NATURAL-Schlüsselwort einsetzen.

Frage/Anforderung: Wie lauten die Namen der P1-Lieferanten?

SQL:

```
SELECT s.sname, sp.pno
  FROM supp_parts sp
NATURAL JOIN suppliers s
 WHERE sp.pno = 'P1'
```

Resultat:

```
sname pno
----- -
Smith P1
Jones P1
```

2.7.6 Kaskadierte JOINS

Natürlich kann man auch mehr als 2 Tabellen durch JOINS verbinden.

Frage/Anforderung: Wie lauten die Namen der Lieferanten und Teile, von denen mehr als 300 verkauft worden sind?

SQL:

```
SELECT s.sname, p.pname, qty
  FROM supp_parts sp
INNER JOIN suppliers s
    ON sp.sno = s.sno
INNER JOIN parts p
    ON sp.pno = p.pno
 WHERE qty > 300
```

Resultat:

```
sname pname qty
----- -
Smith Screw 400
Jones Bolt 400
Clark Cam 400
(3 row(s) affected)
```

```

SELECT *
FROM suppliers s
    INNER JOIN supp_parts sp
        ON s.sno = sp.sno
    INNER JOIN parts p
        ON p.pno = sp.pno

```

sno	sname	status	city	sno	pno	qty	pno	pname	color	weight	city
S1	Smith	20	London	S1	P1	300	P1	Nut	red	12.0	London
S1	Smith	20	London	S1	P2	200	P2	Bolt	green	17.0	Paris
S1	Smith	20	London	S1	P3	400	P3	Screw	blue	17.0	Rome
S1	Smith	20	London	S1	P4	200	P4	Screw	red	14.0	London
S1	Smith	20	London	S1	P5	100	P5	Cam	blue	12.0	Paris
S1	Smith	20	London	S1	P6	100	P6	Cog	red	19.0	London
S2	Jones	10	Paris	S2	P1	300	P1	Nut	red	12.0	London
S2	Jones	10	Paris	S2	P2	400	P2	Bolt	green	17.0	Paris
S3	Blake	30	Paris	S3	P2	200	P2	Bolt	green	17.0	Paris
S4	Clark	20	London	S4	P2	200	P2	Bolt	green	17.0	Paris
S4	Clark	20	London	S4	P4	300	P4	Screw	red	14.0	London
S4	Clark	20	London	S4	P5	400	P5	Cam	blue	12.0	Paris

Bild 2.6: Kaskadierte JOINS

Beispiel

2.7.7 Übung JOIN.1

Frage/Anforderung: *Lesbare Hitparade der Verkäufe*

Resultat:

qty	sname	pname
400	Clark	Cam
400	Jones	Bolt
...		
100	Smith	Cog

2.7.8 Übung JOIN.2

Frage/Anforderung: *Anzahl Verkäufe je Verkäufer, alphabetisch nach Verkäufer*

Resultat:

Verk	#
Blake	200
Clark	900
Jones	700
Smith	1300

2.7.9 Übung JOIN.3

Einsatz des Kreuzproduktes für Datenanalysen.

Frage/Anforderung: *Vergleichsliste aller Verkäufer, die das gleiche Teil gleich oft verkauft haben. Ziel ist es, mögliche Geschäftsbeziehungen zu erkennen.*

Resultat:

qty	pno	sno	sno
-----	-----	-----	-----

```

300    P1      S2      S1
200    P2      S3      S1
200    P2      S4      S3
200    P2      S4      S1
(4 row(s) affected)

```

2.7.10 Inner/Outer JOINS

Beim Vergleich in der ON-Bedingung kann es vorkommen, dass eine der beiden Seiten leer, d.h. = NULL ist.

Der INNER JOIN verlangt, dass dies nicht der Fall ist. INNER ist der Default.

OUTER JOINS erlauben, dass

- entweder die linke Tabelle (LEFT OUTER JOIN)
- oder die rechte Tabelle (RIGHT OUTER JOIN)
- oder eine der beiden Tabellen (FULL OUTER JOIN)

keine Entsprechung in der anderen Tabelle braucht, d.h. dass die andere Tabelle beim gebundenen Attribut NULL sein darf.

2.7.11 LEFT OUTER JOIN

Frage/Anforderung: Welche Teile liegen in der selben Stadt wie welche Händler? Händler ohne entsprechendes Teil sollen auch gelistet werden!

SQL:

```

SELECT sno, pno
  FROM suppliers s
LEFT OUTER JOIN parts p
    ON s.city = p.city

```

Resultat:

sno	pno
S1	P1
S1	P4
S1	P6
S2	P2
S2	P5
S3	P2
S3	P5
S4	P1
S4	P4
S4	P6
S5	NULL

```
(11 row(s) affected)
```

2.7.12 OUTER JOINS in Oracle 8

Bei Oracle 8 existiert eine andere Schreibweise für OUTER JOINS, da dort ja auch das Schlüsselwort JOIN unbekannt ist:

Frage/Anforderung: Welche Teile liegen in der selben Stadt wie welche Händler? Händler ohne entsprechendes Teil sollen auch gelistet werden!

SQL:

```
SELECT sno, pno
  FROM suppliers , parts
 WHERE
    suppliers.city =      parts.city(+)
```

Resultat:

sno	pno
S1	P1
...	
S4	P6
S5	NULL

(11 row(s) affected)

Es ist also an dem Schlüsselattribut, das auch leer sein darf, die Zeichenkette (+) anzubringen.

2.7.13 RIGHT OUTER JOIN

Frage/Anforderung: Welche Teile liegen in der selben Stadt wie welche Händler? Teile ohne entsprechenden Händler sollen auch gelistet werden!

SQL:

```
SELECT sno, pno
  FROM suppliers s
RIGHT OUTER JOIN parts p
  ON s.city = p.city
```

Resultat:

sno	pno
S1	P1
S4	P1
S2	P2
S3	P2
NULL	P3
S1	P4
S4	P4
S2	P5
S3	P5
S1	P6

S4 P6

(11 row(s) affected)

2.7.14 FULL OUTER JOIN

Frage/Anforderung: Welche Teile liegen in der selben Stadt wie welche Händler? Alleinstehende Teile, Händler sollen auch gelistet werden!

SQL:

```
SELECT sno, pno
FROM suppliers s
FULL OUTER JOIN parts p
ON s.city = p.city
```

Resultat:

sno	pno
S1	P1
S4	P1
S2	P2
S3	P2
NULL	P3
S1	P4
S4	P4
S2	P5
S3	P5
S1	P6
S4	P6
S5	NULL

(12 row(s) affected)

2.7.15 FULL OUTER JOINs in Oracle 8

Hinweis: FULL OUTER JOINs können in Oracle 8 über die mengenmäßige Verknüpfung (UNION) von zwei SELECTs erreicht werden.

Frage/Anforderung: Welche Teile liegen in der selben Stadt wie welche Händler? Alleinstehende Teile, Händler sollen auch gelistet werden!

SQL:

```
SELECT sno, pno
  FROM suppliers , parts
 WHERE
    suppliers.city =    parts.city(+)
UNION
SELECT sno, pno
  FROM suppliers , parts
 WHERE
    suppliers.city(+) =    parts.city
```

Resultat:

sno	pno
S1	P1
...	
S3	P2
NULL	P3
S5	NULL

(12 row(s) affected)

2.7.16 Übung OUTER.1: NOT IN for databases not having NOT IN

Frage/Anforderung: Alle Händler aus Städten, wo es keine Teile gibt ...

SQL:

```
SELECT sname, city
  FROM suppliers
 WHERE city NOT IN
 (SELECT city FROM parts)
```

Resultat:

sname	city
Adams	Athens

Nett, aber in MySQL 3.x gibt's kein IN (SELECT ...)

2.8 Mengenoperationen

(Set Operators)

Die mit SELECT gewonnenen Zeilensetzen lassen sich folgendermaßen mengentheoretisch verknüpfen:

- UNION: Zeilen vereinigen. Duplikate werden nicht übernommen.
- UNION ALL: Zeilen vereinigen. Duplikate werden übernommen.
- INTERSECT: Schnittmenge beider SELECTs.
- MINUS: Restmenge erster SELECT - zweiter SELECT

Bis auf UNION lassen sich die selben Ergebnisse mit IN/NOT IN erzielen.

2.8.1 UNION

Frage/Anforderung: Alle Städte, wo Händler oder Teile sind ...

SQL:

```
SELECT city FROM suppliers
UNION
SELECT city FROM parts
```

Resultat:

```
city
-----
Athens
London
Paris
Rome
(4 row(s) affected)
```

2.8.2 Regeln für UNION

- Beliebig viele Zeilenmengen können vereinigt werden
- Die Spaltennamen der Zeilenmengen können verschieden sein
- Die Spaltentypen müssen kompatibel sein
- Die erste Zeilenmenge gibt die Typen vor

2.8.3 Interne Reihenfolge bei SELECTs

1. Kreuzprodukt über alle Tabellen, die durch FROM/JOIN spezifiziert werden
2. Restriktion: Zeilen bestimmen, welche die WHERE-Bedingung erfüllen
3. Projektion: Spalten gemäß der SELECT-Klausel herauspicken/umbenennen
4. Gruppenbildung gemäß GROUP BY
5. Gruppen-Restriktion gemäß HAVING
6. Mengenbildung mit anderen SELECTs (z.B. durch UNION)
7. Sortieren gemäß ORDER BY

3 SQL DML, DDL

3.1 SQL-DDL 1

Data-Definition-Language

Das Thema SQL-DDL wird in zwei Teilen präsentiert. Einer jetzt gleich, um die Voraussetzungen für DML (Data-Manipulation-Language) zu schaffen.

Danach kommt eine DML-Einführung, die zum Verständnis der VIEWS und Indexe besser zwischengeschoben wird (Abschnitt DDL 2).

3.1.1 Tabelle anlegen

```
CREATE TABLE < table-name >
  (spaltendefinition1, ..., spaltendefinitionN
  [, integritätsregel1, ..., integritätsregelM] )
```

wobei

```
spaltendefinitionI ::= 
  spaltenname typangabe
  [DEFAULT klausel]
  [spaltenintegritätsregel]
```

- Der Tabellenname muss innerhalb eines Schemas eindeutig sein.
- Der Spaltenname muss innerhalb einer Tabelle eindeutig sein.
- Eine wichtige Spaltenintegritätsregel ist NOT NULL.

3.1.2 CREATE TABLE Beispiel

Frage/Anforderung: 2-Spaltige Tabelle, keine NULL-Werte erlaubt

SQL:

```
CREATE TABLE sample
  (no INT NOT NULL,
  value FLOAT NOT NULL)
```

3.1.3 Integritätsbedingungen

Integritätstypen:

1. *Operationale Integrität*: Gewährleisten der Funktion nach Hardware-Fehlern, Vandalismus ..., Sichern gegen Probleme, die durch Mehrbenutzerbetrieb entstehen (später ...).
2. *Semantische Integrität*: Konsistenz zur Laufzeit gemäß den Bedingungen, die aus der Mini-Welt folgen:

- Entitäts-Integrität: Eine Zeile ist in der Tabelle eindeutig (SQL: CREATE UNIQUE INDEX, UNIQUE constraints, PRIMARY KEY constraints).
- Wertebereichs-Integrität: Ein Wert wird auf eine gültige Domäne beschränkt (SQL: CHECK constraints, DEFAULT definitions, NOT NULL definitions).
- Referentielle Integrität: Fremdschlüssel-Beziehungen werden überwacht (SQL: FOREIGN KEY constraints).
- Benutzerdefinierte Integrität (SQL: Trigger).

3.1.4 Primärschlüssel

Bestimmt Spalte(n), die einen Satz eindeutig charakterisiert/-en (Entitäts-Integrität).

Frage/Anforderung: *Studierendentabelle mit Nr, Name. Nr ist der Primärschlüssel*

SQL:

```
DROP TABLE p;
CREATE TABLE p
( nr INT NOT NULL,
  name VARCHAR(32),
  PRIMARY KEY (nr)
);
INSERT INTO p
  VALUES (1, 'Ulf');
INSERT INTO p
  VALUES (2, 'Ute');
INSERT INTO p
  VALUES (1, 'Uwe');
```

Resultat:

```
(1 row(s) affected)
(1 row(s) affected)
Server: Msg 2627, Level 14,
        State 1, Line 1
Violation of PRIMARY KEY
constraint 'PK_p_55F4C372'.
Cannot insert
duplicate key in object 'p'.
The statement has
    been terminated.
```

3.1.5 Zusammengesetzte Primärschlüssel

Falls eine Spalte allein nicht eindeutig ist, können auch mehrere Spalten für einen Primärschlüssel definiert werden.

Frage/Anforderung: *In der Verkaufsliste supp_parts soll kein Teil vom gleichen Händler mehrfach geführt werden.*

SQL:

```
ALTER TABLE Supp_Parts
```

```
ADD CONSTRAINT PK_Supp_Parts
PRIMARY KEY (           sno,           pno      )
```

Mit ALTER TABLE kann man auch Spalten hinzufügen, im Typ ändern und andere/weitere Integritätsbedingungen definieren/löschen.

3.1.6 Eindeutige Spalten

Auch Nicht-Primärschlüssel-Spalten können mit UNIQUE auf Eindeutigkeit hin überprüft werden lassen. Solche Spalten nennt man auch Schlüsselkandidaten.

Frage/Anforderung: Studierendentabelle mit Nr, Name. Nr ist der Primärschlüssel, Name soll eindeutig sein.

SQL:

```
DROP TABLE p;
CREATE TABLE p
( nr INT NOT NULL,
  name VARCHAR(32),
  PRIMARY KEY (nr),
  UNIQUE (name)
);
INSERT INTO p
  VALUES (1, 'Ulf');
INSERT INTO p
  VALUES (2, 'Ute');
INSERT INTO p
  VALUES (3, 'Ulf');
```

Resultat:

```
...
Violation of UNIQUE KEY
constraint 'UQ__p__5AB9788F'.
Cannot insert duplicate
key in object 'p'.
The statement has
been terminated.
```

3.1.7 Check-Constraint

Setzt Bedingung für erlaubte Spaltenwerte.

CHECK < bedingung >

wobei die bedingung wie in der WHERE-Klausel formuliert wird.

Frage/Anforderung: Stud.-Tabelle mit Name, Anrede; Anrede nur "Frau" oder "Herr"

SQL:

```
DROP TABLE p;
CREATE TABLE p
```

```

( nr INT NOT NULL,
  name VARCHAR(32),
  anr CHAR(4),
  PRIMARY KEY (nr),
  CHECK (
    anr IN ('Frau',
             'Herr')
  )
);
INSERT INTO p
  VALUES (1, 'Ulf',
          'Herr');
INSERT INTO p
  VALUES (2, 'Ute',
          'Frau');
INSERT INTO p
  VALUES (3, 'Don',
          'Mr.');

```

Resultat:

```

Server: Msg 547, Level 16,
        State 1, Line 1
INSERT statement conflicted
with
COLUMN CHECK constraint
'CK_p_anrede_681373AD' ...

```

3.1.8 Default-Constraint

Setzt Standardwerte, falls ein Attribut beim INSERT nicht spezifiziert wird.

Frage/Anforderung: *Stud.-Tabelle mit Name, falls Name nicht spezifiziert wird, dann den gerade angemeldeten Datenbankbenutzer einsetzen.*

SQL:

```

DROP TABLE p;
CREATE TABLE p
( nr INT NOT NULL,
  name VARCHAR(32)
  DEFAULT current_user,
  PRIMARY KEY (nr)
);
INSERT INTO p
  (nr)
  VALUES (1234);
SELECT * FROM p;

```

Resultat:

nr	name
1234	dbo

(1 row(s) affected)

3.1.9 Referentielle Integrität

Durch die FOREIGN KEY Klausel lassen sich Fremdschlüsselbeziehungen überwachen.

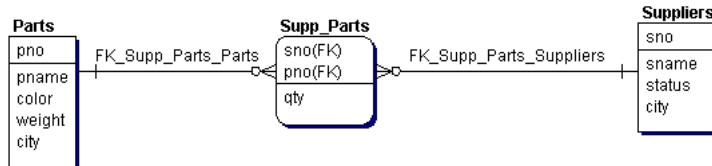


Bild 3.1: Referentielle Integrität in der Beispieldatenbank

```
CONSTRAINT < constraint_name > FOREIGN KEY
(
    < spaltenname >
) REFERENCES < herkunftstabelle > (
    < spaltenname_herkunftstabelle >
)
{ ON DELETE
    [NO ACTION|CASCADE|SET NULL|SET DEFAULT] }
{ ON UPDATE
    [NO ACTION|CASCADE|SET NULL|SET DEFAULT] }
```

Mit den Schlüsselwörtern ON DELETE, ON UPDATE lässt sich festlegen, was passieren soll, wenn ein Satz in der Herkunftstabelle gelöscht/geändert wird:

NO ACTION

Der DELETE/UPDATE der Herkunftstabelle wird verweigert, falls es einen abhängigen Satz gibt.

CASCADE

Der DELETE/UPDATE der Herkunftstabelle wird an die abhängige Tabelle durchgereicht.

SET NULL

Die betroffenen Sätze in der abhängigen Tabelle werden auf NULL gesetzt und somit zu Waisen (Orphans).

SET DEFAULT

Die betroffenen Sätze in der abhängigen Tabelle werden auf den DEFAULT-Wert, der in der CREATE-TABLE-Anweisung spezifiziert wurde (z.B. einen Fremdschlüssel, der verwaiste Sätze als solche kennzeichnet), gesetzt.

Frage/Anforderung: *Die Tabelle supp_parts ist von der Tabelle parts und der Tabelle suppliers abhängig. Jede Änderung soll an die Tabelle supp_parts weitergereicht werden.*

SQL:

```
CREATE TABLE supp_parts (
    sno varchar (8) NOT NULL ,
    pno varchar (8) NOT NULL ,
    qty int NULL ,
```

```

CONSTRAINT PK_Supp_Parts PRIMARY KEY
( sno, pno ) ,
CONSTRAINT FK_Supp_Parts_Parts FOREIGN KEY
( pno )
REFERENCES Parts ( pno )
    ON DELETE CASCADE ON UPDATE CASCADE ,
CONSTRAINT FK_Supp_Parts_Suppliers FOREIGN KEY
( sno )
REFERENCES Suppliers ( sno )
    ON DELETE CASCADE ON UPDATE CASCADE
)

```

3.2 SQL DML

Data-Manipulation-Language

3.2.1 Zeilen einfügen

```

INSERT INTO < table-name >
[(col1, col2, ..., colN)]
VALUES (val1, val2, ..., valN)

```

- Die Anzahl N der Spaltennamen (erster Klammerausdruck) und der spezifizierten Werte (zweiter Klammerausdruck) muss übereinstimmen.
- Die jeweilige Zuordnung erfolgt über die Position in den Klammerausdrücken.
- Alle Spalten, die nicht im ersten Klammerausdruck definiert werden, werden entweder
 - auf einen Default-Wert, der beim Anlegen der Tabelle für diese Spalte definiert worden ist (bei Zeitstempel-Spalten häufig CURRENT_TIMESTAMP) oder
 - auf NULL

gesetzt. In diesem Fall ist eine Angabe der Spaltennamen wegen der späteren Nachvollziehbarkeit unverzichtbar.

- Falls auf die Angabe der Spaltennamen im ersten Klammerausdruck verzichtet wird, müssen alle Spalten ohne Default- bzw. mit dem NOT-NULL-Constraint spezifiziert werden.

3.2.2 INSERT Beispiel

Frage/Anforderung: *Es gibt einen neuen Händler S6 ...*

SQL:

```

INSERT INTO suppliers
(sno, sname, status, city)
VALUES
('S6', 'Miller', 10, 'Orlando');
SELECT * FROM suppliers;

```

Resultat:

(1 row(s) affected)

sno	sname	status	city
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens
S6	Miller	10	Orlando

(6 row(s) affected)

3.2.3 INSERT per Abfrage

Sollen Tabellen insbesondere vom Inhalt her (teilweise) kopiert werden, so kann man dies mit `INSERT INTO` machen. Oft verletzt dies aber die Normalisierung!

Frage/Anforderung: Alle Zeilen doppelt, neue Zeilen mit 'cc'-Präfix ...

SQL:

```
BEGIN TRANSACTION
INSERT INTO parts
  (pno, pname, color,
   weight, city)
  SELECT
    CONCAT('cc', pno), pname,
    color, weight, city
   FROM parts
SELECT * FROM parts
ROLLBACK
```

Resultat:

pno	pname
ccP1	Nut ...
ccP2	Bolt ...
...	
ccP6	Cog
P1	Nut
...	
P6	Cog ...

Frage/Anforderung: Schnell viele Zeilen (INSERT-Lasttest)

SQL:

```
CREATE TABLE sample (no int, value float);
INSERT INTO sample
  (no, value)
VALUES
```

```

(1, 0.5);

INSERT INTO sample
SELECT no + 1, (value + 1) / value
FROM sample;
-- INSERT mehrfach wiederholen ...
    SELECT COUNT(*) FROM sample;
DROP TABLE sample;

```

Resultat:

```
-----
2097152
```

3.2.4 Zeilen löschen

DELETE

```
      FROM < table-name >
      [WHERE < search-condition > ]
```

- Wird die WHERE-Restriktion weggelassen (bzw. vergessen!), so werden alle Zeilen gelöscht. Die Tabellenstruktur bleibt aber erhalten.
- Es ist daher insbesondere bei interaktiven SQL-Sitzungen ratsam, erst mit einer

```
SELECT COUNT(*) FROM < table-name >
      WHERE < search-condition >
```

Anweisungen zu prüfen, wie viel Zeilen die WHERE-Klausel passieren und somit später gelöscht werden.

- Aus den selben Gründen sollte die DELETE-Anweisung in eine Transaktion (später ...) verpackt werden.

3.2.5 DELETE Beispiel

Frage/Anforderung: *Es gibt keine Händler S1..S3 mehr*

SQL:

```
BEGIN TRANSACTION;
SELECT COUNT(*) FROM supp_parts;
SELECT COUNT(*) FROM suppliers
  WHERE sno <= 'S3';
DELETE FROM suppliers
  WHERE sno <= 'S3';
SELECT * FROM suppliers;
SELECT COUNT(*) FROM supp_parts; -- !!
ROLLBACK;
```

Resultat:

```
-----
12
```

```
-----  
3
```

```
(3 row(s) affected)
```

sno	sname	status	city
S4	Clark	20	London
S5	Adams	30	Athens

```
-----  
3
```

3.2.6 DELETE Seiteneffekte

- Wird beim Einreichen des Foreign-Key-Constraints festgelegt, dass das Löschen weitergegeben werden soll, so kann eine DELETE sehr viel mehr Zeilen als ursprünglich vermutet löschen!
- Man muss also vor dem Löschen das konzeptionelle Schema und die daraus folgenden Integritätsbedingungen genau kennen!
- Transaktionen einsetzen.

3.2.7 Tabellen löschen

Ganze Tabellen (Tabellenstruktur nebst Inhalt) löscht man mit:

```
DROP TABLE < table-name >  
      [CASCADE]
```

CASCADE sorgt dafür, dass auch Sätze in anderen Tabellen, die sich auf die zu löschen Tabelle beziehen, gelöscht werden.

3.2.8 Zeilen ändern

```
UPDATE   < table-name >  
        SET col1 = value, ...  
        [WHERE  < search-condition > ]
```

- Wird die WHERE-Restriktion weggelassen (bzw. vergessen!), so werden alle Zeilen geändert.
 - Es ist daher insbesondere bei interaktiven SQL-Sitzungen ratsam, erst mit einer
- ```
SELECT COUNT(*) FROM < table-name >
 WHERE < search-condition >
```

Anweisungen zu prüfen, wie viel Zeilen die WHERE-Klausel passieren und somit später geändert werden.

- Aus den selben Gründen sollte die UPDATE-Anweisung in eine Transaktion (später ...) verpackt werden.

### 3.2.9 UPDATE Beispiel

Frage/Anforderung: Händler Miller zieht nach Houston um und bekommt Status 15

SQL:

```
BEGIN TRANSACTION;
SELECT * FROM suppliers
 WHERE sno = 'S6';

UPDATE suppliers
 SET
 status = 15,
 city = 'Houston'
WHERE sno = 'S6';

SELECT * FROM suppliers
 WHERE sno = 'S6';

ROLLBACK;
```

Resultat:

```
sno sname status city
---- ----- ----- -----
S6 Miller 10 Orlando

(1 row(s) affected)

sno sname status city
---- ----- ----- -----
S6 Miller 15 Houston
```

### 3.2.10 INSERT, UPDATE Beispiel

SQL:

```
INSERT INTO suppliers
 (sno, sname, status, city)
VALUES
 ('S6', 'Miller', 10, 'Orlando');

INSERT INTO supp_parts (sno, pno, qty)
 VALUES ('S6','P1', 1200);

UPDATE supp_parts SET qty=1300
 WHERE sno='S6' AND pno='P1';

UPDATE suppliers SET sno='S7'
 WHERE sno='S6';

SELECT * FROM suppliers;
SELECT * FROM supp_parts WHERE qty > 1000 ;
```

## 3.3 SQL-DDL 2

Data-Definition-Language, 2. Teil

### 3.3.1 Sichten (Views)

- Virtuelle Tabellen
- Berechnung der "Tupel" zur Laufzeit
- Zur Anpassung an spezielle Benutzerbedürfnisse und zum Datenschutz (d.h. zum Bau des externen Schemas)
- Zum Verbergen von komplexen Datenstrukturen (z.B. Primär-Fremdschlüssel-Beziehungen, d.h. Auflösen von Joins in anwendergerechte Form)
- Datentypen, Constraints, etc. werden von den Basistabellen übernommen

Vorteile:

- Adaption an verschiedene Benutzerklassen:
  - Nur relevante Daten werden angezeigt (sowohl relevante Spalten als auch relevante Zeilen).
  - Spalten können benutzerfreundlich - z.B. in der entsprechenden Landessprache, Maßeinheit - präsentiert werden.
- Das interne Schema kann geändert (meist erweitert) werden und Altanwendungen bleiben weiter lauffähig.
- Komplexe SELECTs - deren Entwicklung durchaus zeitintensiv und damit kostenintensiv ist - werden zentral (am Server) gehalten und sind damit für andere Anwendungen verfügbar.
- Views sind im Gegensatz zu Stored Procedures, die ebenfalls am Server gehalten werden, schon in SQL92 standardisiert und damit relativ portabel.
- Einige Zugriffe lassen sich nur mit Views bewerkstelligen, z.B. mehrere GROUP BYs verbinden (mit JOIN), UNIONs verbinden.

Syntax:

```
CREATE VIEW <view name>
[(<column name> [, <column name> ...])]
 AS <query expression>
 [WITH CHECK OPTION]
```

Die CHECK OPTION stellt sicher, dass INSERTs, UPDATEs und DELETEs auf dem View die WHERE-Bedingung nicht verletzen.

In der query-expression - die ansonsten weitgehend dem SELECT entspricht - sollte es kein ORDER BY geben.

### 3.3.2 Beispiele für Sichten (Views)

Frage/Anforderung: *Sicht, die alle roten Teile umfasst.*

SQL:

```
CREATE VIEW red_parts
AS SELECT *
 FROM parts
 WHERE color = 'red';

SELECT * FROM red_parts;
```

Resultat:

```
pno pname color weight city
---- ----- ----- -----
P1 Nut red 12.0 London
P4 Screw red 14.0 London
P6 Cog red 19.0 London

(3 row(s) affected)
```

### 3.3.3 INSERT in Sichten

Frage/Anforderung: *Mehrere Teile über den View red\_parts einfügen*

SQL:

```
INSERT INTO red_parts
VALUES('P7', 'Washer',
'pink', 12., 'Rome');
INSERT INTO red_parts
VALUES('P8', 'Cog',
'red', 5., 'Rome');
INSERT INTO red_parts
VALUES('P9', 'Nut',
'red', 15., 'Rome');

SELECT * FROM red_parts;
SELECT pno, color
FROM parts
 WHERE pno > 'P6';
```

Resultat:

```
pno color city
----- -----
P1 red London
P4 red London
P6 red London
P8 red Rome
P9 red Rome

pno color
----- -----
```

```
P7 pink
P8 red
P9 red
```

Frage/Anforderung: *Sicht, die alle schweren roten Teile umfasst.*

SQL:

```
CREATE VIEW highweight_red_parts
AS SELECT *
 FROM red_parts
 WHERE weight >= 10.00
WITH CHECK OPTION;

SELECT pno, weight
FROM highweight_red_parts;
```

Resultat:

```
pno weight
---- -----
P1 12.0
P4 14.0
P6 19.0
P9 15.0

(4 row(s) affected)
```

Frage/Anforderung: *Nutzen der CHECK OPTION*

SQL:

```
INSERT INTO
 highweight_red_parts
VALUES
('PA', 'Washer',
 'red', 5, 'Stuttgart');
```

Resultat:

```
Server: Msg 550,
Level 16, State 1,
Line 1
The attempted insert or
update failed because
the target view either
specifies WITH CHECK OPTION
or spans a view that
specifies WITH CHECK OPTION
and one or more rows resulting
from the operation did not
qualify under the CHECK OPTION
constraint.
The statement has been terminated.
```

### 3.3.4 Zusammenfassendes Beispiel

Frage/Anforderung: Eine Sicht mit der Anzahl aller Händler und Anzahl Teile je Stadt.

SQL:

SQL-Browser ...

Resultat:

| city    | TAnz | HAnz |
|---------|------|------|
| Athens  | NULL | 1    |
| London  | 3    | 2    |
| Orlando | NULL | 1    |
| Paris   | 2    | 2    |
| Rome    | 1    | NULL |

Probleme:

- 2 GROUP BYs werden benötigt
- Stadtname soll in genau einer Spalte stehen

### 3.3.5 Änderungen auf Sichten

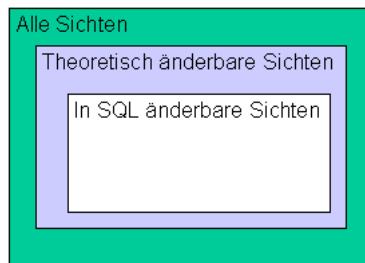


Bild 3.2: Mögliche Änderungen auf Sichten

Folgende Voraussetzungen muss eine Sicht in SQL erfüllt haben, damit sie änderbar ist:

- Nur eine Basisrelation.
- Der Schlüssel muss - komplett - sichtbar bleiben.
- Keine ...
  - Aggregate,
  - Gruppierungen und
  - Duplikateliminierung (DISTINCT)

erlaubt.

### 3.3.6 Indexe

Durch Indexe lassen sich Abfragen oft beschleunigen. Das geht aber auf Kosten eines zusätzlichen Speicherbedarfs und einer verlangsamten Bearbeitung von DML-Befehlen wie INSERT, UPDATE und DELETE.

```

CREATE [UNIQUE] [CLUSTERED] INDEX
[< index-name >]
ON < table-name >
(< column-name >, ...)
[ASC | DESC]

```

Herstellerspezifisch variiert diese Syntax, es können meist Dinge wie Belegungsgrad, Art des Index (B\*-Index, Hash-Index) zusätzlich spezifiziert werden.

Die meisten Hersteller erzeugen automatisch Indexe für

- Primärschlüssel und
- UNIQUE-Constraints

### 3.3.7 Beispiel für Indexe

Frage/Anforderung: *Beschleunigen eines GROUP-BYs*

SQL:

```

CREATE TABLE sample (no int, value float);
INSERT INTO sample
 (no, value)
VALUES
 (1, 0.5);

 INSERT INTO sample
 SELECT no + 1, (value + 1) / value
 FROM sample;
-- INSERT INTO 20 mal wiederholen ...
 SELECT no, COUNT(*) FROM sample group by no;
CREATE CLUSTERED INDEX Idx3 ON sample(no);
 SELECT no, COUNT(*) FROM sample group by no;
DROP TABLE sample;

```

Resultat:

| no  |       |
|-----|-------|
| 15  | 38760 |
| 7   | 38760 |
| 14  | 77520 |
| ... |       |

(21 row(s) affected)

| no  |     |
|-----|-----|
| 1   | 1   |
| 2   | 20  |
| 3   | 190 |
| ... |     |

(21 row(s) affected)

# 4 Transaktionen

## 4.1 Recovery

### 4.1.1 Fehler-Gegenmaßnahmen des DBMS

Transaktionen (später mehr): SQL:

```
SELECT * FROM supp_parts WHERE pno='P1'
SELECT SUM(qty) FROM supp_parts -- check
-- shift 100 P1-parts from s1 to s2:
BEGIN TRAN
UPDATE supp_parts SET qty = 200 WHERE sno='S1' AND pno='P1'
UPDATE supp_parts SET qty = 400 WHERE sno='S2' AND pno='P1'
COMMIT -- or ROLLBACK
SELECT SUM(qty) FROM supp_parts -- check
-- and back...:
BEGIN TRAN s2_s1
UPDATE supp_parts SET qty = 300 WHERE sno='S1' AND pno='P1'
UPDATE supp_parts SET qty = 300 WHERE sno='S2' AND pno='P1'
COMMIT
```

Was geschieht, wenn eine Transaktion "*in der Mitte*" abgebrochen wird? Ein **Recovery** findet statt. (Mehr dazu, insbesondere alternative Ansätze wie *Multi-Versioning-Concurrency-Control (MVCC)* dann in Datenbanken 2.)

### 4.1.2 Log-Dateien

- DBMS speichert zusätzlich zu den geänderten Daten die Änderungen an der Datenbank in separaten Log-Dateien ab.
- Die geänderten Daten verbleiben (wenn möglich) im Puffer.
- Aus Sicherheitsgründen werden jedoch die ursprünglichen und die geänderten Daten in der Log-Datei wirklich auf die Platte geschrieben (persistiert).
- Log-Dateien werden üblicherweise auf einer anderen physikalischen Platte gespeichert.

### 4.1.3 Funktion der Log-Datei

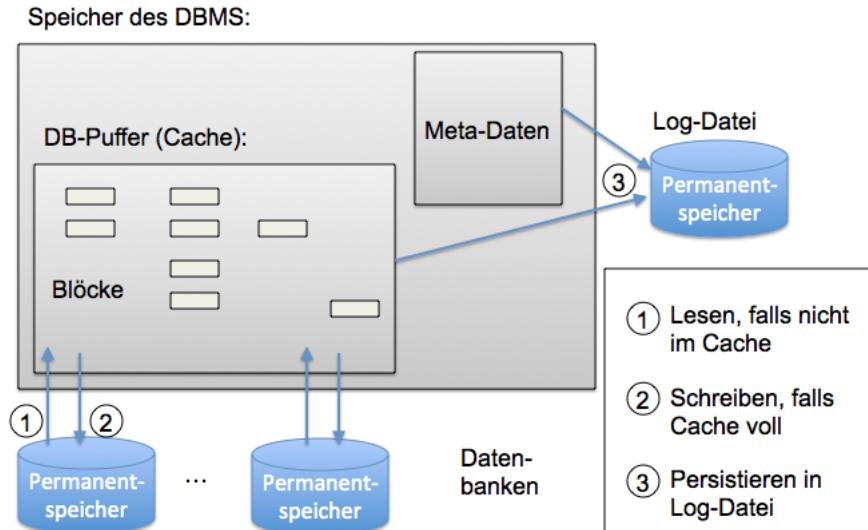


Bild 4.1: Funktion der Log-Datei und des DB-Puffers

### 4.1.4 Ablauf einer Transaktion mit der Log-Datei

Tafel...

### 4.1.5 Meta-Daten

- Welche Blöcke sind im Cache?
- Welche davon wurden geändert?
- Welche Transaktionen sind offen?

### 4.1.6 Meta-Daten: Experiment 1

SQL:

```
Sp_who -- für userids
Sp_lock -- für Locks
BEGIN TRAN s2_s1
UPDATE supp_parts SET qty = 333
 WHERE sno='S1' AND pno='P1'
Sp_lock -- für Locks
```

### 4.1.7 Meta-Daten: Experiment 2

SQL:

```
BEGIN TRAN s2_s1
UPDATE supp_parts SET qty = 444
 WHERE sno='S1' AND pno='P1'
UPDATE supp_parts SET qty = 444
 WHERE sno='S2' AND pno='P1'
COMMIT
-- in anderem Fenster:
select * from syscacheobjects order by dbid desc
-- Dort kann man die gecachten Blöcke sehen.
-- In der letzten Spalte steht,
```

```
-- wegen welcher Sql-Anweisung das
-- war. Dann
select * from parts
```

#### 4.1.8 Log-Dateien: Bemerkungen

- Meist reicht ein Block in der Log-Datei für eine Transaktion.
- Dieser Block wird mit Ende der Transaktion auf der Platte gespeichert.
- Das Before-Image jedoch wird sofort in der Log-Datei abgespeichert.
- Log-Dateien wachsen schnell: Backups machen sie klein.
- Einige Hersteller kennen weitere Art von Log-Dateien: Archive-Log-Dateien.

#### 4.1.9 Fehlerbehebung mit Log-Datei

**Lokaler Fehler** (nur eine Transaktion betroffen):

- DBMS setzt zurück auf das Before-Image.
- Meldung an Admin/User.

#### 4.1.10 Fehlerbehebung mit Log-Datei

**Hardcrash** (Hardware, etwa Platte fällt aus):

1. Platte mit Datenblöcken ausgefallen:

- DBMS wird durch Admin gestoppt.
- Alle offenen Transaktionen werden zurückgesetzt.
- Neue Platte einbauen.
- Backup wieder einspielen.
- Nachvollziehen der Änderungen mit Hilfe der Log-Datei.

2. Platte mit Log-Datei ausgefallen:

- Obwohl ein Notbetrieb möglich ist, sollte man ein inkrementelles Backup machen, den Betrieb stoppen, und die Platte tauschen.

## 4.2 Parallelbetrieb (Concurrency )

**Situation:**

Viele Anwender greifen gleichzeitig auf dieselben Daten zu.

**Problem:**

Gewährleistung des Konsistenz bei mehreren konkurrierenden Zugriffen.

**Lösung:**

Konzept der Transaktionen

**Basisprinzip der Concurrency:**

Jede Transaktion läuft ab, als sei sie alleine.

## 4.2.1 Definition Transaktion

- Eine Transaktion ist eine Folge von Operationen, die als ein einzelner logischer Arbeitsschritt aufgefasst wird.
- Eine Transaktion muss folgende vier Eigenschaften erfüllen (ACID-Prinzip):
  1. Atomicity,
  2. Consistency,
  3. Isolation und
  4. Durability.

## 4.2.2 Das ACID-Prinzip

### Atomicity

Alle Operationen einer Transaktion werden als eine Einheit betrachtet, deshalb werden entweder alle Operationen oder gar keine ausgeführt.

### Consistency

Auch bei gleichzeitigem Zugriff durch mehrere Benutzer wird die Datenbank von einem in einen anderen konsistenten Zustand überführt.

### Isolation

Eine Transaktion kann nur auf konsistente Daten zugreifen, jedoch keine Zwischenergebnisse einer gleichzeitig ablaufenden Transaktion verwenden.

### Durability

Die Änderungen an den Daten stehen nach Ausführung der Transaktion dauerhaft zur Verfügung.

## 4.2.3 Fehlerklassen bei parallelen Transaktionen

Folgende Fehlerklassen gibt es beim Parallelbetrieb von Transaktionen:

- Lost-Update-Problem
- Dirty Read
- Non-Repeatable Read
- Phantom

## 4.2.4 Lost-Update-Problem

- Ein Lost-Update in einer Transaktion ist eine Veränderung in dieser Transaktion, die von einer anderen Transaktion überschrieben wird.
- Keine Transaktion weiß etwas über die andere.

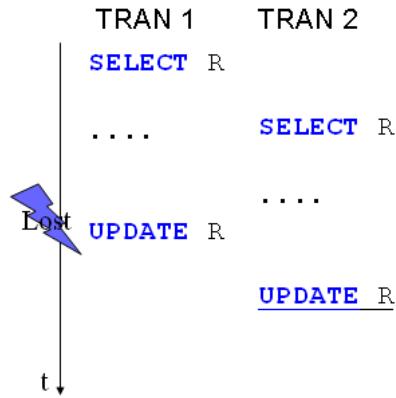


Bild 4.2: Zwei Transaktionen, die einen Lost Update erzeugen.

#### 4.2.5 Dirty Read

(Manchmal auch 'Uncommitted Dependency' genannt)

- Ein Dirty Read in einer Transaktion TRAN1 ist ein Lesevorgang, der veränderte Zeilen einer anderen, noch nicht terminierten (z.B. durch COMMIT oder ROLLBACK), Transaktion TRAN2 liest.
- Falls z.B. TRAN2 per ROLLBACK terminiert oder TRAN2 ein weiteres mal Änderungen an R vornimmt, dann hat TRAN1 die falschen Daten gelesen.

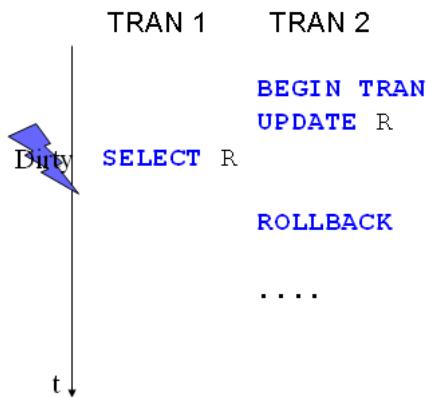


Bild 4.3: Zwei Transaktionen, die einen Dirty Read erzeugen.

#### 4.2.6 Dirty Read: Experiment

SQL:

```

-- 1st window
SELECT * FROM parts
SET TRANSACTION ISOLATION LEVEL
 READ UNCOMMITTED -- später...
SELECT * FROM parts
-- 2nd window
BEGIN TRAN
UPDATE parts SET weight=4711
 WHERE pno='P5'
ROLLBACK

```

Resultat:

```

pno pname color weight
----- -----
...
P5 Cam blue 12.0

-- 2nd Select:
pno pname color weight
----- -----
...
P5 Cam blue 4711.0
...

```

## 4.2.7 Non-Repeatable Read

(Manchmal auch 'Inconsistent analysis' genannt)

- Ein Non-Repeatable Read in einer Transaktion ist ein Lesevorgang, der im Falle von mehrmaligem Lesen zu unterschiedlichen Ergebnissen führt.

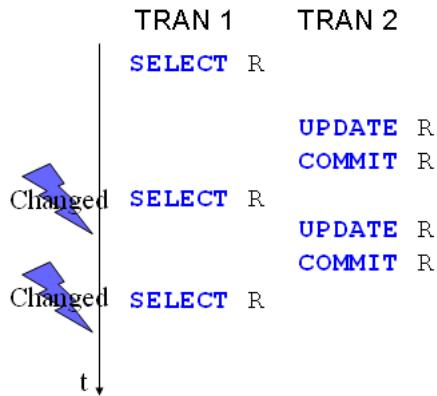


Bild 4.4: Zwei Transaktionen, die einen Non-Repeatable Read erzeugen.

## 4.2.8 Non-Repeatable Read: Experiment

SQL:

```

-- 1st window
UPDATE parts SET weight=4711
 WHERE pno='P1'; COMMIT
UPDATE parts SET weight=2345
 WHERE pno='P1'; COMMIT
UPDATE parts SET weight=12
 WHERE pno='P1'; COMMIT

-- 2nd window
SET TRANSACTION ISOLATION LEVEL
 READ COMMITTED -- später ...
BEGIN TRAN
SELECT * FROM parts
SELECT * FROM parts
COMMIT

```

Resultat:

```
pno pname color weight
```

```

...
P5 Cam blue 4711.0
...
P5 Cam blue 2345.0
...

```

#### 4.2.9 Phantom Read

- Ein Phantom in einer Transaktion ist ein Lesevorgang, der eine Menge von Zeilen liest und im Falle von mehrmaligem Lesen eine unterschiedliche Anzahl von Zeilen erhält.

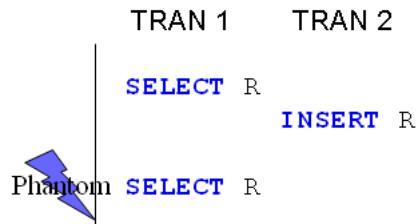


Bild 4.5: Zwei Transaktionen, die einen Phantom-Read erzeugen.

#### 4.2.10 Phantom Read: Experiment

SQL:

```

-- 2nd window
BEGIN TRAN
INSERT INTO parts
 VALUES ('PHT', 'phantom',
 'red', 42, 'Opera'); -- (1)
DELETE FROM parts WHERE pno='PHT'
COMMIT
-- 1st window
SET TRANSACTION ISOLATION LEVEL
 read uncommitted -- serializable -- später ...
BEGIN TRAN
SELECT COUNT(*) FROM parts -- before (1)
SELECT COUNT(*) FROM parts -- after (1)
COMMIT

```

Resultat:

```

6
-- after (1):

7

```

## 4.3 SQL Isolationsebenen

Parallelbetrieb kann vom DBMS durch Serialisierung der gewünschten Operationen erreicht werden.

Allerdings ist eine so strenge Serialisierung oft nicht nötig und aus Performance-Gründen auch nicht wünschenswert.

Daher definiert SQL verschiedene Isolationsebenen, die bestimmen, in wie weit eine Transaktion isoliert von den anderen abläuft.

### 4.3.1 SQL Isolationsebenen

Folgende Isolationsebenen gibt es:

- Read Uncommitted (schwächste Ebene, in der lediglich verlangt wird, dass physikalisch falsche Daten nicht gelesen werden können).
- Read Committed (meist Standard-Isolationsebene)
- Repeatable Read
- Serializable (höchste Ebene, in der Transaktionen komplett isoliert von einander ablaufen).

### 4.3.2 SQL Isolationsebenen und Fehlerklassen

Folgende Isolationsebenen lassen folgende Fehler zu:

| Isolationsebene         | Mögl. Fehler: |                    |              |
|-------------------------|---------------|--------------------|--------------|
|                         | Dirty Read    | Nonrepeatable Read | Phantom Read |
| <b>Read Uncommitted</b> | Ja            | Ja                 | Ja           |
| <b>Read Committed</b>   | Nein          | Ja                 | Ja           |
| <b>Repeatable Read</b>  | Nein          | Nein               | Ja           |
| <b>Serializable</b>     | Nein          | Nein               | Nein         |

### 4.3.3 SQL Isolationsebenen und Lost Updates

- Transaktionen müssen in der Isolationsebene "Repeatable Read" betrieben werden, wenn der Fehlertyp "Lost Update" sicher vermieden werden soll.
- Falls die Updates der Transaktionen nicht von in der Transaktion zuvor gemachten Abfragen abhängen und einzelige Updates sind, dann reicht die Isolationsebene "Read Committed" aus.

#### 4.3.4 Beispiel: Isolationsebenen und "Lost Updates"

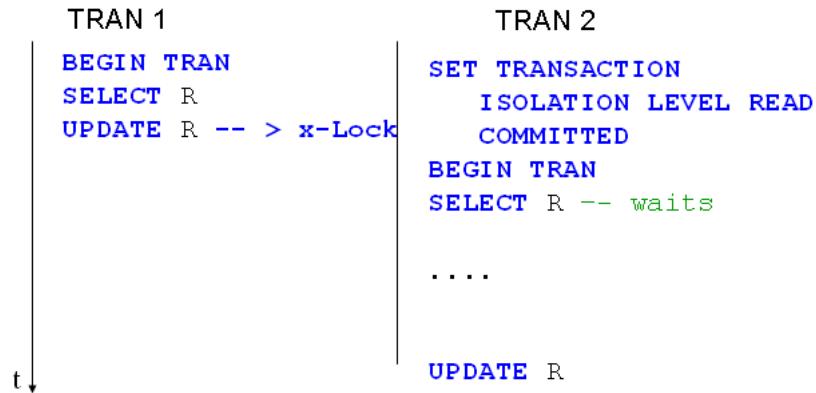


Bild 4.6: Zwei Transaktionen, die einen Lost Update vermeiden.

#### 4.3.5 Beispiel: Isolationsebenen und "Dirty Reads"

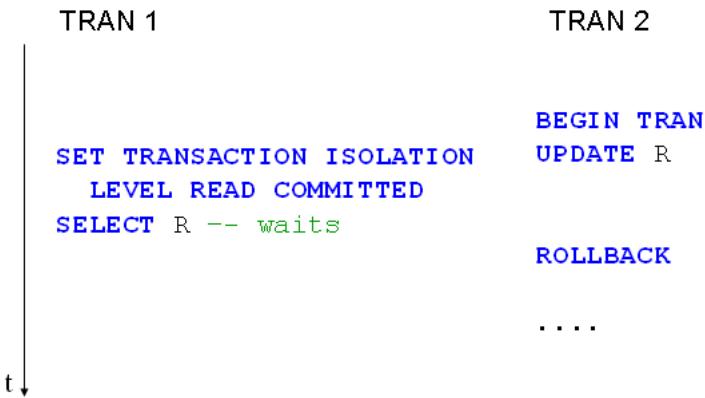


Bild 4.7: Zwei Transaktionen, die einen Dirty Read vermeiden.

#### 4.3.6 Beispiel: Isolationsebenen und "Nonrepeatable Reads"

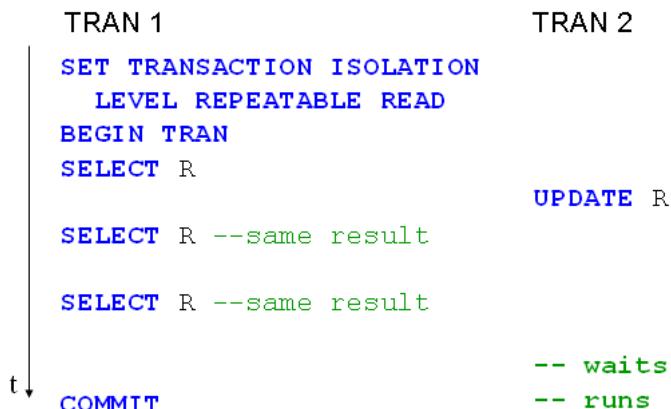


Bild 4.8: Zwei Transaktionen, die einen Nonrepeatable Reads vermeiden.

#### 4.3.7 Beispiel: Isolationsebenen und "Phantome"

|                                                                                             |                                 |
|---------------------------------------------------------------------------------------------|---------------------------------|
| TRAN 1                                                                                      | TRAN 2                          |
| <pre>SET TRANSACTION ISOLATION<br/>    LEVEL SERIALIZABLE<br/>BEGIN TRAN<br/>SELECT R</pre> | <pre>INSERT R</pre>             |
| <pre>SELECT R<br/>COMMIT</pre>                                                              | <pre>-- waits<br/>-- runs</pre> |

**t** ↓

Bild 4.9: Zwei Transaktionen, die einen Phantom Read vermeiden.

# 5 Java Sprachanbindungen

## 5.1 JDBC Grundlagen

### 5.1.1 JDBC Einordnung

- Portable Datenbankprogrammierung unter Java.
- Das JDBC Paket `java.sql` ist eine Standard-API zur Ausführung von SQL Anweisungen unter Java.
- Das Paket umfasst Klassen und Schnittstellen, um SQL-Anweisungen abzusetzen und geg. die Ergebnisse von RDBMS abzuholen.
- JDBC hat ein Framework, mit dem unterschiedlichste Treiber für die verschiedenen Datenbanken dynamisch geladen werden können.
- Es gibt sogar auch Treiber, um den Zugriff auf Textdateien mit SQL zu ermöglichen!
- Keine neue Datenbankabfragesprache - eher eine objektorientierte API für SQL.
- Im Gegensatz dazu: ESQL/SQLJ bei denen SQL Anweisungen direkt in den C+-/Java-Code integriert sind und durch einen Vorübersetzer in API Anweisungen übersetzt werden.

### 5.1.2 Ziele des Designs von JDBC

#### API für SQL

- Fokus auf die Ausführung von SQL-Anweisungen und Empfang von deren Ergebnissen.
- Es gibt auch höherwertige APIs (SQLJ, eingebettetes SQL in Java und JDO, Java Data Objects, EJBs, ...).

#### SQL Konformität

- Bereinigen von nicht-konformen Sprachkonstrukten einzelner Hersteller (Outer Joins, STPs, Datumsangaben/-funktionen).
- JDBC erlaubt es, jeglichen String, der dann als SQL interpretiert wird, an das DBMS zu senden. Dies kann jedoch auch zu Ausnahmen (Exceptions) führen.
- JDBC soll auf verbreiteten DB-Interfaces basieren.
- Die JDBC-Schnittstellen sollen konsistent zum sonstigen Java-System sein.
- Keep it simple.
- Wenn möglich, soll starke, statische Typisierung eingesetzt werden. Ziel ist die frühe Prüfung der Typisierung zur Compile-Zeit.
- Die üblichen Programmieraufgaben sollen einfach bearbeitbar sein.
- Je Aufgabe eine Methode (wenig Steuerung der Funktionalität über Parameter).

### 5.1.3 JDBC Vor- und Nachteile

Vorteile

Nachteile

- Einfachheit
- Einfach handhabbar und installierbar
- Security
- Stabilität
- Plattform-Unabhängigkeit
- DBMS-Unabhängigkeit
- Kosteneffizient
- Kurze Entwicklungszyklen möglich
- Geschwindigkeit
- Es können sich Nicht-Standard-SQL-Funktionen leicht einschleichen.
- Versionsprobleme
- Große Teile der SQL-Funktionalität, wie Korrektheit der Tabellen-/Attribut-/...-namen, Zugriffsrechte und Typkompatibilitäten werden erst zur Laufzeit geprüft. Intensives Testen - gerade auch der Ausnahmen und Fehlermöglichkeiten - z.B. mit JUnit - ist daher Pflicht.
- Bei dabbiger (süddt. für naiv) Programmierung leichtes Opfer für SQL-Injection Angriffe (s.u.).

## 5.2 JDBC Architektur

### 5.2.1 JDBC und N-Tier Architekturen

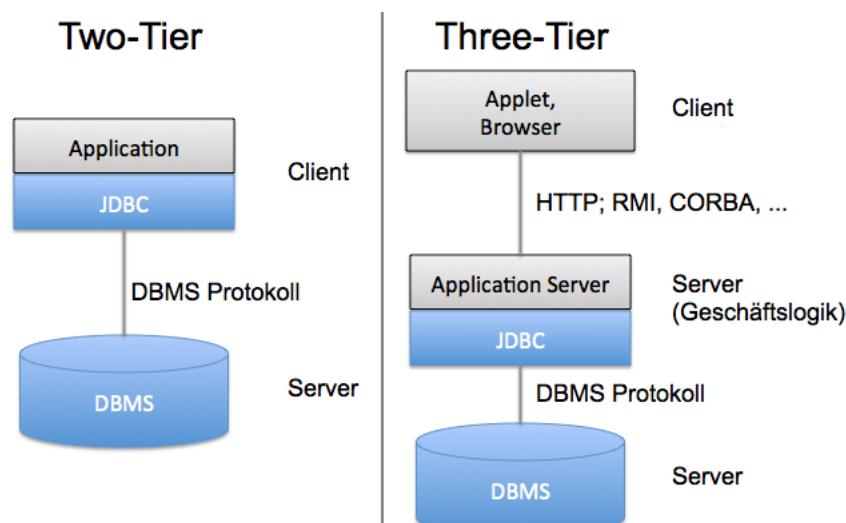


Bild 5.1: N-Tier mit JDBC

## 5.2.2 JDBC Treiber

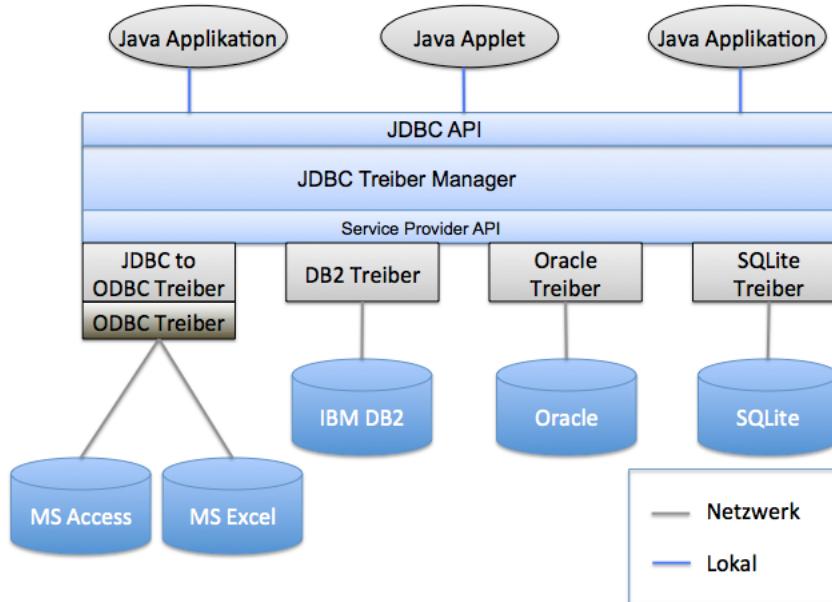


Bild 5.2: Treiber-Anbindung

## 5.2.3 JDBC Treiber Typen

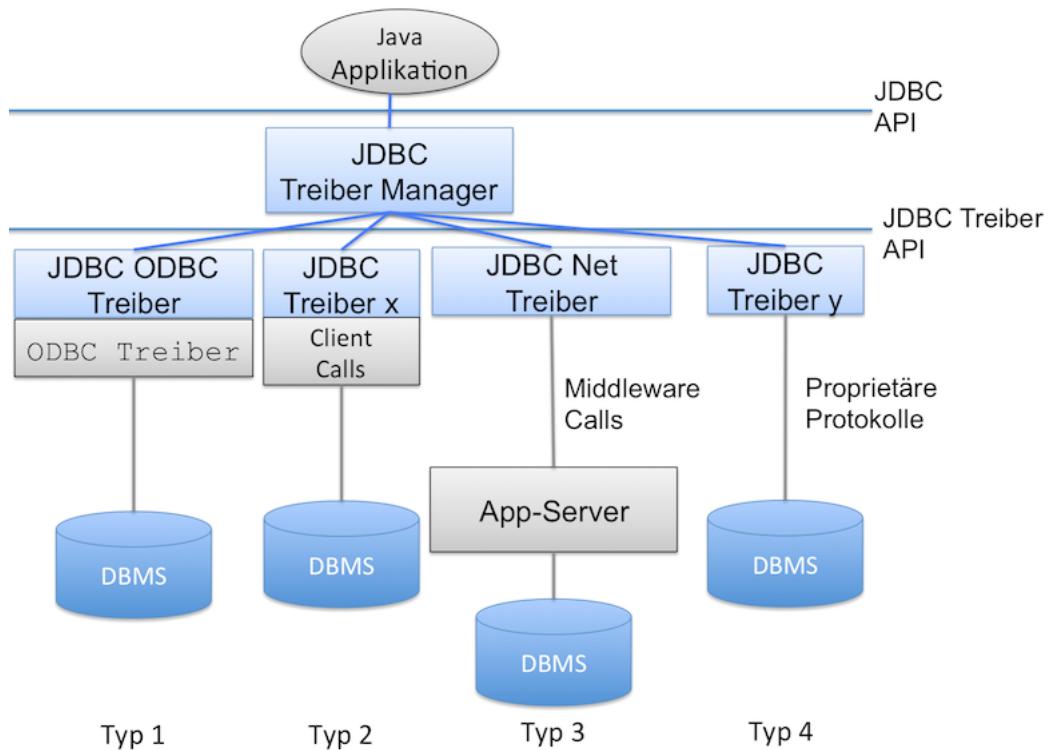


Bild 5.3: Treiber Typen von JDBC

Nach Prof. Mario Jeckle, † 2004 [<http://www.jeckle.de/>]

### Typ 1

Die historisch älteste Variante bildet der Typ 1 Treiber. Strenggenommen verkörpert er selbst keinen Datenbanktreiber, sondern lediglich eine Umsetzungsschicht die einem existierenden ODBC-Treiber vorgeschaltet wird. Die Abbildung belegt diesen Treibertyp daher mit dem Begriff JDBC-ODBC-Bridge, da er lediglich den

Brückenschlag zwischen den beiden Standards vornimmt und sich in der konkreten Anwendung auf die Umsetzung zwischen den beiden Protokollen beschränkt, ohne realen Zugriff auf die Datenbank zu erhalten. Dieser ist dem ODBC-Treiber vorbehalten, der im allgemeinen Falle mit einer weiteren Umsetzungsstufe kommuniziert, welche die generischen ODBC-Aufrufe in konkrete DBMS-spezifische wandelt. Während sowohl der JDBC-ODBC-Brückentreiber als auch der ODBC-Treiber selbst für verschiedene DBMS verwendet werden können, muss für jedes konkrete DBMS eine herstellerspezifische, d.h. an das verwendete DBMS angepasste, Bibliothek vorliegen.

#### **Typ 2**

Für den Fall eines Typ 2 Treibers entfällt diese durch ODBC geschaffene zusätzliche Indirektionsstufe zugunsten der Adaption der Konversionskomponente, welcher die Wandlung der Aufrufe in das DBMS-native Protokoll obliegt, an das JDBC-Protokoll und ihrer Integration in den JDBC-Treiber selbst. Die Natur der Kommunikation des Java-Anteils des Treibers mit den Nativen ist im Rahmen der durch die JDBC-Spezifikation gegebenen Definition nicht festgelegt. Durch die Integration der DBMS-nativen Treiberanteile in den JDBC-Treiber muss dieser für jedes anzusprechende DBMS neu erstellt werden. Eine Wiederverwendung der JDBC-spezifischen Anteile, die für die Clientkommunikation eingesetzt werden, kann hierbei nicht erfolgen.

#### **Typ 3**

Der Fall der (partiellen) Konkretisierung dieser Kommunikationsbeziehung zu einem beliebigen DBMS-neutralen Protokoll wird durch einen Typ 3 Treiber aufgegriffen. Hier wird die DBMS-spezifische Komponente (in der Abbildung grau dargestellt) als vom JDBC-Treiber separiertes Modul aufgefasst, dass mit diesem mittels eines festgelegten neutralen Protokolls kommuniziert. Durch diese Separierung, die auch durch Installation auf physisch getrennten Maschinen - der DBMS-spezifische Anteil könnte beispielsweise auf einem Middleware-Server untergebracht werden - fundiert werden kann, gelingt die Wiederverwendung des JDBC-Treiberanteils, der mit verschiedenen DBMS-spezifischen Bibliotheken über das gewählte Protokoll kommunizieren kann.

#### **Typ 4**

Der Typ 4 Treiber stellt die letzte durch die JDBC-Spezifikation vorgesehene Ausprägung dar. Er konzipiert eine vollständig in Java implementierte Zugriffsschicht, die in sich geschlossen ist. Sie besitzt daher lediglich die notwendige JDBC-Schnittstelle zur Kommunikation mit der Java-Applikation und eine DBMS-Spezifische zum Zugriff auf die Datenquelle. Die Vorteile dieser Architekturvariante liegen in ihrer Portabilität und den geringen Installations- und Wartungsaufwänden, die aus der Reduktion der Kommunikationsbeziehungen resultieren. So kann ein solcher Treiber durch einfache Integration in die Java-Applikation verwendet werden und bedarf keiner Installationen oder Modifikationen an der verwendeten Ausführungsumgebung. Gleichzeitig offenbart sich diese Lösung jedoch als technisch aufwendig in der Umsetzung, sobald DBMS verschiedener Hersteller angesprochen werden sollen, da die JDBC-Anteile des Treibers nicht separat wiederverwendet werden können.

## Laufzeitverhalten

Hinsichtlich des Laufzeitverhaltens zeigt sich deutlich die Schwäche der Typ 1 Treiber, welche in der inhärent notwendigen Doppelkonversion (JDBC zu ODBC und ODBC zu nativem Aufruf) begründet liegt. Daher sind Treiber dieses Typs als Übergangsscheinung hin zu "echten" JDBC-Treibern, d.h. Treibern der restlichen Typen, anzusehen und sollten in Produktivumgebungen nicht eingesetzt werden.

Die Vorteile der Typ 2 und 3 Treiber seitens der Ausführungsgeschwindigkeit liegen in den nativen Codeanteilen begründet, welche für das jeweilige verwendete DBMS optimiert werden können. Zwar spricht der leichte Installations- und Administrationsaufwand eindeutig für Typ 4 Treiber, jedoch fallen diese in ihrer Leistungsfähigkeit durch die ausschließliche Verwendung der Programmiersprache Java teilweise deutlich hinter Treiber des Typs 2 und 3, mitunter sogar hinter solche des Typs 1, zurück.

Sie verkörpern jedoch den aus konzeptioneller Sicht zu bevorzugenden Ansatz hinsichtlich Portabilität und Vergleichbarkeit der erzielten quantitativen Ergebnisse. Typischerweise kommen im produktiven Einsatz jedoch Treiber der Typen 2 und 4 zum Einsatz, die entweder durch den Hersteller des DBMS mitgeliefert werden (Typ 2) oder auf der Basis publizierter Schnittstellen plattformunabhängig für genau ein spezifisches DBMS entwickelt wurden (Typ 4). Generell formuliert das JDBC-Konzept auf dieser Ebene noch keine Einschränkung hinsichtlich der unterstützten DBMS-Typen und ist generell auf verschiedenste Datenquellen anwendbar. Durch die Struktur des API und die verfügbaren Treiber kristallisieren sich jedoch relationale DBMS als Hauptanwendungsgebiet dieser Zugriffsschnittstelle heraus.

## 5.3 Programmieren mit JDBC

### 5.3.1 Die JDBC API

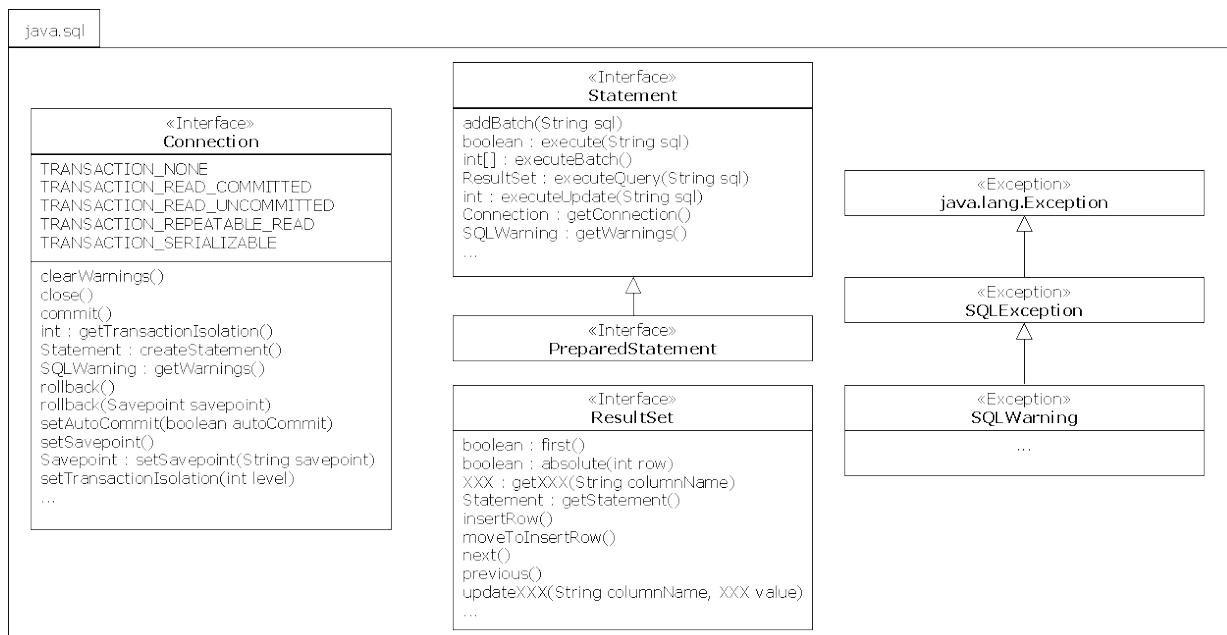


Bild 5.4: JDBC Pakete

## 5.3.2 Ablauf einer JDBC-Anwendung

### Verbindsaufbau

- Explizites Laden eines Datenbanktreibers durch Classloader
- oder Automatisches Laden eines Datenbanktreibers (durch Setzen der "sql.drivers" Property in den System Properties)
- Angabe von
  - Benutzer
  - Passwort
  - Server
  - Port
  - Datenbank
  - Schemaentweder über Strings oder über Properties.

### Aufbereiten der SQL-Anweisungen

- am besten als PreparedStatements

### Empfang der Ergebnisse

- Zeilenweise über Resultset

### Aufräumen

Schließen von

- ResultSet
- (Prepared-) Statement
- Verbindung

mit der jeweiligen close - Methode.

Es ist äußerst wichtig, jede dieser Ressourcen explizit **so früh wie möglich** freizugeben, da sie am Server erhebliche Ressourcen binden und auch zu Sperren und Verlangsamungen anderer Anwendungen führen können, die mit dem selben DBMS arbeiten. Der Java-Garbage-Collector macht dies nicht (immer)!

## 5.3.3 Beziehungen zwischen den Interfaces

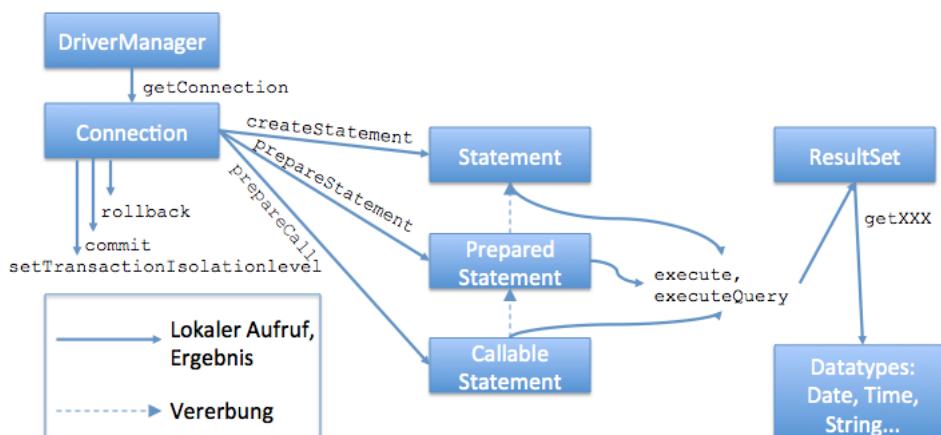


Bild 5.5: Beziehungen zwischen den Interfaces

### 5.3.4 JDBC Datentypen-Behandlung

Zum Beispiel im ResultSet sind folgende Abbildungen der SQL- auf die Java-Datentypen möglich:

|                    | TINYINT | SMALLINT | INTEGER | BIGINT | REAL | FLOAT | DOUBLE | DECIMAL | NUMERIC | BIT | CHAR | VARCHAR | LONGVARCHAR | BINARY | VARBINARY | LONGVARBINARY | TIME | TIMESTAMP | CLOB | BLOB | ARRAY | REF | STRUCT | JAVA OBJECT |  |
|--------------------|---------|----------|---------|--------|------|-------|--------|---------|---------|-----|------|---------|-------------|--------|-----------|---------------|------|-----------|------|------|-------|-----|--------|-------------|--|
| getByte            | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getShort           | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getInt             | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getLong            | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getFloat           | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getDouble          | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getBigDecimal      | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getBoolean         | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getString          | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           |        |           |               |      |           |      |      |       |     |        |             |  |
| getBytes           |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         |               |      |           |      |      |       |     |        |             |  |
| getDate            |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         |               |      |           |      |      |       |     |        |             |  |
| getTime            |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         |               |      |           |      |      |       |     |        |             |  |
| getTimestamp       |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         |               |      | X         | X    |      |       |     |        |             |  |
| getAsciiStream     |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         | X             | X    |           |      |      |       |     |        |             |  |
| getBinaryStream    |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         | X             |      |           |      |      |       |     |        |             |  |
| getCharacterStream |         |          |         |        |      |       |        |         |         |     |      |         |             | X      | X         | X             | X    |           |      |      |       |     |        |             |  |
| getClob            |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |           | X    |      |       |     |        |             |  |
| getBlob            |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |           |      | X    |       |     |        |             |  |
| getArray           |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |           |      |      | X     |     |        |             |  |
| getRef             |         |          |         |        |      |       |        |         |         |     |      |         |             |        |           |               |      |           |      |      |       | X   |        |             |  |
| getObject          | X       | X        | X       | X      | X    | X     | X      | X       | X       | X   | X    | X       | X           | X      | X         | X             | X    | X         | X    | X    | X     | X   | X      | X           |  |

Bild 5.6: JDBC Datentypen [<http://docs.oracle.com/javase/1.3/docs/guide/jdbc/getstart/mapping.html>]

### 5.3.5 Beispiel JDBC Treiberauswahl

```
import java.sql.*;
import java.util.*;
class Select {
 public static void main(String[] argv) {
 try {
 String url = "";
 if (argv.length == 0 ||
 argv[0].equalsIgnoreCase("odbc"))
 {
 // Create a URL
 Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
 url = "jdbc:odbc:db_sql2000";
 }
 else // Native MySQL driver
 if (argv[0].equalsIgnoreCase("mysql"))
 {
 Class.forName("org.gjt.mm.mysql.Driver");
 // Create a URL
 url = "jdbc:mysql://localhost:3306/db";
 }
 }
 }
}
```

### 5.3.6 Beispiel JDBC Verbindungsaufbau

```
// Login Data:
Properties props = new Properties();
props.put("user", "guest");
props.put("password", "");

// Connect to the database at that URL.
Connection con =
 DriverManager.getConnection(url, props);
```

### 5.3.7 Beispiel JDBC SELECT

```
// Execute a SELECT statement
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
 "SELECT sno, status, city FROM Suppliers");

// Step through the result rows.
System.out.println("Got results for " + url);
while (rs.next()) {
 // get the values from the current row:
 String sno = rs.getString(1);
 int status = rs.getInt(2);
 String city = rs.getString("city");
 // Now print out the results:
 System.out.print(" sno=" + sno);
 System.out.print(" status=" + status);
 System.out.print(" city=" + city);
 System.out.print("\n");
}
```

### 5.3.8 Aufräumen: So viel Zeit muss sein...

Gilt nicht nur für JDBC!

```
// Clean up!
rs.close();
stmt.close();
con.close();
```

### 5.3.9 JDBC Fehlerbearbeitung

```
} catch (java.lang.Exception ex) {
 System.out.println("SQL Exception: " +
 ex.getLocalizedMessage());

 ex.printStackTrace();
} // catch
} // main
} // class Select
```

### 5.3.10 Erweiterte JDBC Fehlerbearbeitung

```
} catch (SQLException ex) {
 System.out.println("SQL Exception: " +
```

```

 ex.getLocalizedMessage());

SQLException nextException = ex.getNextException();
while (nextException != null){
 System.out.println("SQL Exception: " +
 nextException.getLocalizedMessage());
 nextException = nextException.getNextException();
}
}

ex.printStackTrace();
} // catch

```

Ausgabe:

```

SQL Exception: Batch-Eintrag 0 UPDATE suppliers SET status = -1 *
status WHERE sno = 'S1'
wurde abgebrochen. Rufen Sie 'getNextException' auf, um die Ursache
zu erfahren.
SQL Exception: ERROR: relation "suppliers" does not exist
Position: 8

```

## 5.4 Transaktionen unter JDBC

### 5.4.1 Beispiel JDBC Update, Transaktionsmanagement

```

import java.util.*;
import java.sql.*;
import javax.swing.*;
class Update {
 public static void main(String argv[]) {
 // ... (s.o.)

 // Prepare Transaction
 con.setTransactionIsolation(
 Connection.TRANSACTION_SERIALIZABLE);
 con.setAutoCommit(false);

 // Create a statement to update
 Statement stmt = con.createStatement();
 int nAffected = stmt.executeUpdate(
 "UPDATE suppliers SET status = 34 " +
 "WHERE status = 20");

 JOptionPane.showMessageDialog(null,
 nAffected +
 " rows updated. " +
 "Use db-browser-tool to show...\n",
 "Info", JOptionPane.INFORMATION_MESSAGE);

 // Restore, clean up
 con.rollback();
 stmt.close();
 con.close();
 } catch (java.lang.Exception ex) {
 //...
}

```

## 5.4.2 Steuerung von Transaktionen mit JDBC

- Für jede Verbindung, die geöffnet wird, kann die benötigte Transaktionsebene z.B. mit `con.setTransactionIsolationLevel(TRANSACTION_SERIALIZABLE)` eingestellt werden.

Man kann also auf eine Datenbank mit **verschiedenen Transaktionsebenen** zugreifen, indem man mehrere Verbindungen öffnet, damit man stark blockierende Transaktionsebenen wie TRANSACTION\_SERIALIZABLE nur bei Transaktionen einsetzen muss, die das nötig haben und möglichst kurz laufen.

Es ist hersteller- und treiber-spezifisch verschieden, ob man die Transaktionsebene nur unmittelbar nach dem Öffnen der Connection setzen kann oder auch zu einem beliebigen anderen Zeitpunkt.

- Nicht jedes Datenbanksystem unterstützt alle Isolations-Ebenen. Welche Isolations-Ebenen von einer Datenbank unterstützt werden, kann man über die Metadaten-Schnittstelle in JDBC ermitteln. Die Klasse **DatabaseMetaData** stellt hierfür die Methode `supportsTransactionIsolationLevel()` zur Verfügung.
- Die meisten Treiber bauen Verbindungen im AutoCommit-Modus auf, d.h. jeder Aufruf eines Statements ist eine geschlossene Transaktion. Das kann - zum Beispiel beim Insert von 1.000.000 Zeilen - die Performance dramatisch drosseln.
- Will man mehrere Statements, z.B. für Updates oder Massen-Inserts zusammenfassen, muss man zunächst diesen Modus mit `con.setAutoCommit(false);` abschalten.
- Danach kann man mit `.commit` Transaktionen beenden und mit `.rollback()` zurückrollen.
- Sagt man mit `con.setReadOnly( true )` freundlicherweise dem Server, dass man nur lesen will, wird ihm die Arbeit erleichtert, die Performance steigt.

## 5.5 Sicherheit, Tuning für JDBC

### 5.5.1 Vorbereitete SQL-Anweisungen

(Prepared Statements)

- In der SQL-Anweisung werden Platzhalter ('?') für Parameter definiert und an ein PreparedStatement übergeben.
- Diese Platzhalter werden dann später mit Set-Operationen `setString (int pos, String value)`, `setByte(...)`, `setShort(...)`, ... mit Werten belegt.
- Dann kann mit `executeQuery()` ein ResultSet abgeholt werden.

- DML-Anweisungen werden entsprechend mit der Methode executeUpdate() abgesetzt.
- Bei jedem weiteren Aufruf von executeXxx() müssen nur die Parameter neu gesetzt werden, die sich ändern sollen.

## 5.5.2 Beispiel Prepared Statements

```
// ... (s.o.)

// Create a statement to update
String stmtUpdate =
 "UPDATE suppliers SET sname = ? " +
 "WHERE sno = ? ";

// Prepare Update
PreparedStatement stmt =
 con.prepareStatement(stmtUpdate);
stmt.setString(1, "Joey's");
stmt.setString(2, "S1");
int nAffected = stmt.executeUpdate();

JOptionPane.showMessageDialog(null,
 nAffected +
 " rows updated. " +
 "Use db-browser-tool to show...\n",
 "Info", JOptionPane.INFORMATION_MESSAGE);

// Restore, clean up...
```

## 5.5.3 SQL-Injektion

Mit Hilfe von Prepared-Statements lässt sich die SQL-Injektion vermeiden, da der ganze gegebene Text, der z.B. aus einem GUI-Textfeld oder einer URL stammt, als Text für einen ?-Parameter verwendet wird und nicht als SQL interpretiert wird.

Beispiel für SQL-Injektion:

**Der Programmierer erwartet:**

<http://webserver/cgi-bin/find.cgi?ID=42>

**Daraus gebautes SQL:**

SELECT author, subjekt, text FROM artikel WHERE ID=42

**Der Angreifer schickt:**

<http://webserver/cgi-bin/find.cgi?ID=42;UPDATE+USER+SET+TYPE='admin'+WHERE+ID=23>

**Daraus gebautes SQL:**

SELECT author, subjekt, text FROM artikel WHERE ID=42;  
UPDATE USER SET TYPE='admin' WHERE ID=23

Weiteres Beispiel für SQL-Injektion:

**Der Programmierer erwartet:**

[http://webserver/search.aspx?keyword\(sql](http://webserver/search.aspx?keyword(sql)

Daraus gebautes SQL:

```
SELECT url, title FROM myindex WHERE keyword LIKE '%sql%'
```

Der Angreifer schickt:

```
http://webserver/search.aspx?keyword=sql'+;GO+EXEC+cmds
hell('format+C')+--
```

Daraus gebautes SQL:

```
SELECT url, title FROM myindex WHERE keyword LIKE '%sql' ;
GO EXEC cmdshell('format C') --%
```

## 5.5.4 SQL-Injektion -- ;-)



Bild 5.7: Robert'); Drop Table Students; --



Bild 5.8: Any user input can be harmful... [http://blog.ioactive.com/2013/03/sql-injection-in-wild.html]

## 5.5.5 Row Prefetching

- Beim Row Prefetching kann man die Anzahl  $n$  der Zeilen festlegen, die für ein Statement/Rowset konkret auf einmal vom Server zum Client übertragen werden. Wenn  $n==10$  ist und man 1024 Sätze lesen muss, werden also 103, nicht 1024 Kommunikationen zwischen Client und Server nötig. 10 Zeilen werden jeweils am Client gepuffert.
- Zu kleinen Zahlen  $n$  von Zeilen können dazu führen, dass viel Kommunikation mit dem Datenbanksystem durchgeführt wird, da relativ oft neue Ergebniszellen abgerufen werden müssen
- Bei einem hohen Wert dagegen wird mehr Speicher verbraucht, man liest unter Umständen auch Dinge, die man gar nicht mehr gebrauchen kann. Bei sehr großen  $n$  steigt selbstverständlich die Latenz, was bei interaktiven Anwendungen stört.
- Der Oracle JDBC-Treiber verwendet z. B. intern einen voreingestellten Wert von 10 Zeilen.

Beispiele:

```
Statement stmt = con.createStatement();
stmt.setFetchSize(500);
...
String sql = "select * from parts order by weight desc";
ResultSet rs = stmt.executeQuery(sql);
```

### 5.5.6 Stapelaktualisierung

- Wie beim Verzicht auf AutoCommit sind starke Performanceverbesserungen möglich.
- Es werden SQL-Anweisungen nicht einzeln an den Datenbankserver geschickt sondern sie werden zunächst auf Client-Seite gesammelt und anschließend gebündelt zur Ausführung an den Server übergeben.

```
// In dem String-Array snoOfNegatedStati seien die zu ändernden
// Supplier-IDs gespeichert, deren Stati negiert werden sollen,
// die z.B. einer Buchprüfung entstammen ...
String sql =
 "UPDATE suppliers SET status = -1 * status WHERE sno = ?";
PreparedStatement prep = con.prepareStatement(sql);
for (int i = 0; i < snoOfNegatedStati.length; i++) {
 prep.setString(1, snoOfNegatedStati[i]);
 // Hinzufügen der Anweisung zum Stapel
 // Beim ersten Aufruf wird das Statement-Objekt
 // in den Stapel-Modus versetzt
 prep.addBatch();
}
// Ausführen des Stapels
int[] ergebnis = prep.executeBatch();
System.out.println(ergebnis.length +" Zeilen aktualisiert.");
prep.close();
```

### 5.5.7 Connection Pooling

- Wie schon erwähnt, sollten Verbindungen und Transaktionen nur **kurz** geöffnet sein, um den Server nicht unnötig unter Last zu setzen.
- Anwendungs- und Web-Server setzen diese Forderung regelmäßig um, da sie ja typischerweise eine Anfrage erhalten, am DBMS etwas nachschlagen und geg. ändern und dann eine Antwort zurückgeben. Danach wird die Verbindung zum DBMS **nicht mehr gebraucht**.
- Da der Verbindungsaufbau aber sehr **aufwändig** ist (Überprüfung von Benutzer, Passwort, Zugriffsrechten; Bereitstellen von Ressourcen) ist es einfacher, wenn Verbindungen nur einmal angelegt und dann wiederverwendet werden. Dies wird durch Connection Pooling realisiert.

Beispiel (für Postgres [<http://www.postgresql.org/docs/7.4/static/jdbc-datasource.html>]; die Implementierungen des Poolings sind leider herstellerspezifisch):

```
public class PoolExamp {
 Connection con = null;
 Jdbc3PoolingDataSource pool = null;
```

```

void initPool ()
{
 pool = new Jdbc3PoolingDataSource();
 pool.setDataSourceName("vorl_db");
 pool.setServerName("localhost");
 pool.setDatabaseName("vorl_db");
 pool.setUser("bike");
 pool.setPassword("xxx");
 pool.setMaxConnections(10);
}

void connectPooled()
{
 try {
 con = pool.getConnection();
 // use connection in other class methods as usually
 } catch (SQLException ex) {
 System.out.println("SQL Exception: " +
 ex.getLocalizedMessage());
 ex.printStackTrace();
 }
}

```

Ergebnisse:

Messrechner: 2.93 GHz Intel Core 2 Duo, 4 GB 1067 MHz DDR3

Experiment: 10.000 mal lokale Connection auf, zu

| $t_{\text{Non-Pooled}}$ [s] | $\sigma_{\text{Non-Pooled}}$ | ( $\sigma$ ist die Standardabweichung) | $t_{\text{Pooled}}$ [s] | $\sigma_{\text{Pooled}}$ |
|-----------------------------|------------------------------|----------------------------------------|-------------------------|--------------------------|
| 9,557                       | 0,311                        |                                        | 0,563                   | 0,023                    |

→ Faktor 17,0 ....

### 5.5.8 Aufruf einer Gespeicherten Prozedur (Stored Procedure)

Gegeben sei folgende STP (MS-SQL2000):

```

-- DROP PROCEDURE SalesByName

CREATE PROCEDURE SalesByName
 @SupplierName nvarchar(15),
 @@total int output
AS
BEGIN
 SELECT @@total = sum(qty)
 FROM suppliers AS s
 INNER join supp_parts AS sp
 ON sp.sno = s.sno
 WHERE s.sname = @SupplierName
END

```

Diese lässt sich wie folgt unter JDBC aufrufen:

```

// Connection etc. see above...

// Statement to call a stored procedure:
String stmtCall = "{call SalesByName(?, ?)}";

```

```
// Prepare call:
CallableStatement stmt =
 con.prepareCall(stmtCall);
// Set IN-parameter
stmt.setString(1, "Clark");
// Register second place-holder being an OUT-parameter
stmt.registerOutParameter(2, java.sql.Types.INTEGER);
// Execute call
stmt.executeUpdate();
int total = stmt.getInt(2);
// Show in message-box:
JOptionPane.showMessageDialog(null,
 "Clark sold " + total + " parts.",
 "Info", JOptionPane.INFORMATION_MESSAGE);
```

# 6 Modellierung

## 6.1 Einführung

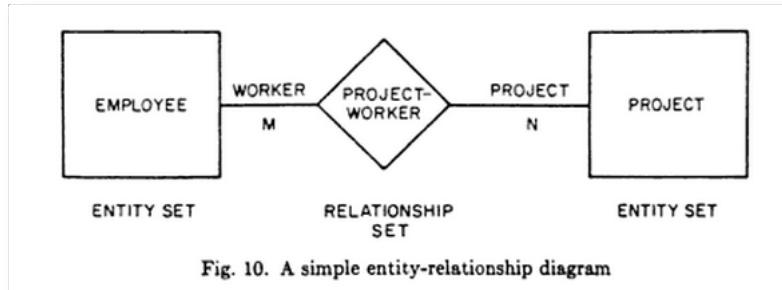


Bild 6.1: Einfaches ER-Modell

[Chen, Peter Pin-Shan: The Entity-Relationship Model--Toward a Unified View of Data, ACM Transactions on Database Systems 1/1/1976, S. 9-36]

Das Entity-Relationship-Modell (ERM) stellt das älteste und allgemeinste Modell zur strukturierten Modellierung von Daten dar. Es wurde 1970-76 von Chen [Che76] definiert und später vielfach weiterentwickelt.

### 6.1.1 Ziele

- Die Bedeutung der (Informations-) Modellierung im Software Entwicklungs-Prozess kennen
- Die Schritte zur Modellbildung kennen
- Modellbildung dokumentieren können

### 6.1.2 Antiziele



Bild 6.2: Neben Spaghetti-Code gibt es auch Spaghetti-Datenbanken

| Acct#   | Name               | Address                 | City          | State | Zip   | Note     |
|---------|--------------------|-------------------------|---------------|-------|-------|----------|
| 5154155 | Peter J. Lalonde   | 40 Beacon St.           | Melrose, Mass |       | 02176 | ODP      |
| 5152335 | LaLonde, Peter     | 76 George 617-210-0824  | Boston        | YES   | MA    | 2111     |
| 5146261 | Lalonde, Sofie     | 40 Bacon Street         | Melrose       |       | MA    | CHK ID   |
| 87121   | Pete & Soph Lalond | 76 George Road          | Boston        | MASS  |       | FR Alert |
| 87458   | P. Lalonde FBO     | S.Lalonde 40 Beacon Rd. | Melrose       | MA    | 02    | 176      |

Bild 6.3: Wohin geht die Rechnung an die Lalondes?

### 6.1.3 Wiederholung ...

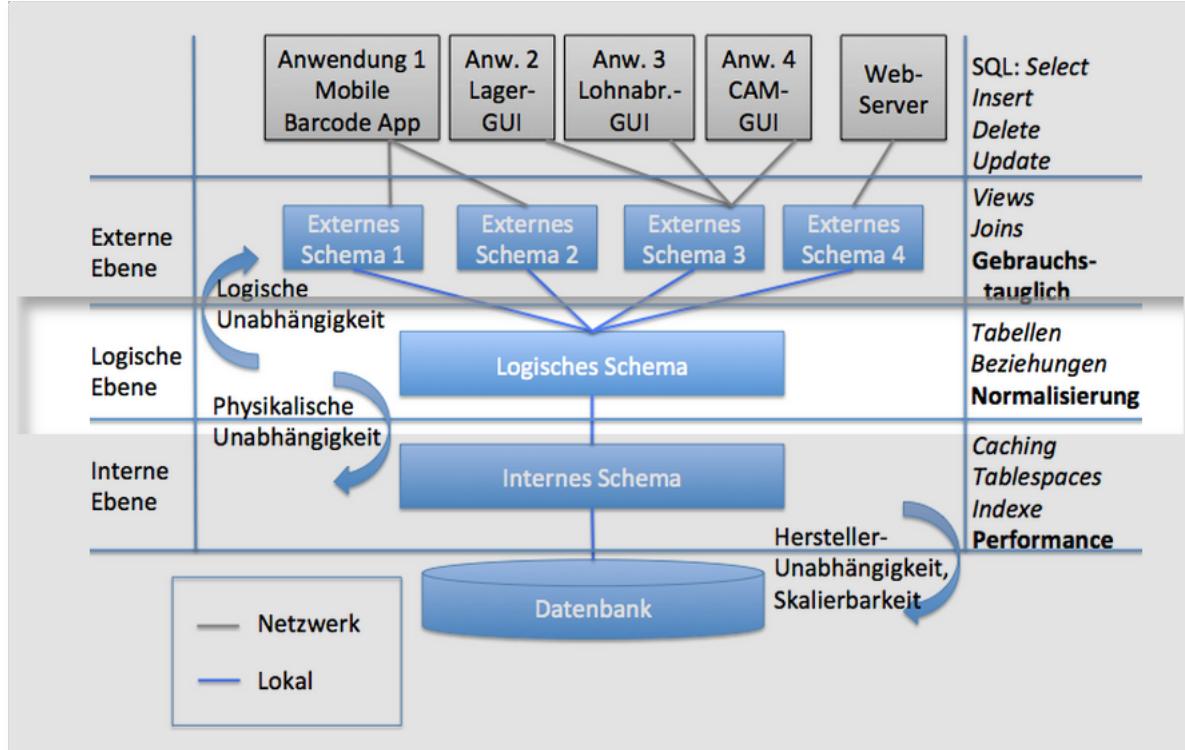


Bild 6.4: Logische Ebene in der Drei-Schema-Architektur

### 6.1.4 Ziele der Modellierung

- **Bestandsaufnahme** durch Untersuchung der Realwelt
- Reduzierung der Realwelt auf das **wesentliche** Informationsangebot
- **Frühzeitige** Begriffsdefinition
- Frühzeitige **Kommunikation** zwischen Systemdesigner und Fachabteilung
- (Grafisches) **Sprachmittel** ist die Entity-Relationship Methode

### 6.1.5 Grundsätzliche Vorgehensweise

1. Datenmodellierung mit der Entity-Relationship-Methode (dieses Kapitel).
2. Normalisierung (nächstes Kapitel).

## 6.2 Die Entity-Relationship-Methode (ERM) nach Chen

### 6.2.1 Grafische Grundelemente

Grundelemente von Entity-Relation-Diagrammen (ERD)



Bild 6.5: Grafische Grundelemente

### 6.2.2 Beispiel: Fuhrpark in einem Kleinbetrieb

- Der Fuhrpark besteht aus einem grünen Chrysler mit dem Kennzeichen S-LS 1719 und einem roten BMW mit dem Kennzeichen M-KT 2004.
- Der Kleinbetrieb beschäftigt die Mitarbeiter Müller, Schmied und Bauer.
- H. Müller gehört der grüne Chrysler, H. Bauer der rote BMW. Im Rahmen eines Leasingmodells stellen sie Ihre Fahrzeuge dem Betrieb zur Verfügung.
- Zusätzlich zu den jeweiligen Fahrzeughaltern, darf H. Schmied den roten BMW von H. Bauer fahren.

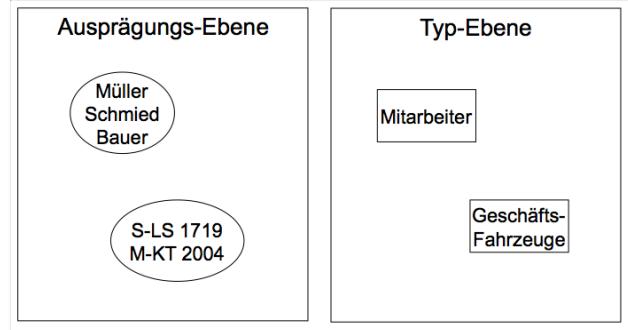


Bild 6.6: Ausprägungs- und Typ-Ebene

### 6.2.3 Entitäten

- Klassifizierung von Entitäten führt zu Entitätstypen
- Entitätstypen sind zeitlich **konstant** bzgl. Existenz und Beschreibung (Mitarbeiter)
- Entitätsausprägungen sind zeitlich **variant** bzgl. Existenz (Müller, Schulz)
- Der Bezeichner für einen Entitätstyp steht im **Plural**, der Bezeichner für eine Entitätsausprägung steht im **Singular**
- Kleinste Einheit** eines Informationsmodells
- Reale Dinge (Fahrzeug, Mitarbeiter)
- Aktivitäten, Ereignisse (Kündigung, Maschinenausfall)
- Abstrakte Zusammenhänge (Projekt)
- Entitäten werden durch **Attribute** beschrieben

## 6.2.4 Attribute

- Identifizierende und beschreibende Attribute
- Schlüsselattribute (Primärschlüssel, Sekundärschlüssel, Fremdschlüssel)
- Für **Entitäten** und **Beziehungen** definiert

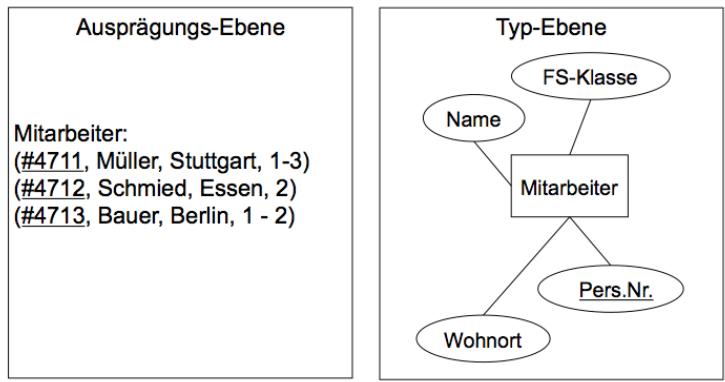


Bild 6.7: Attribute

## 6.2.5 Beziehungen

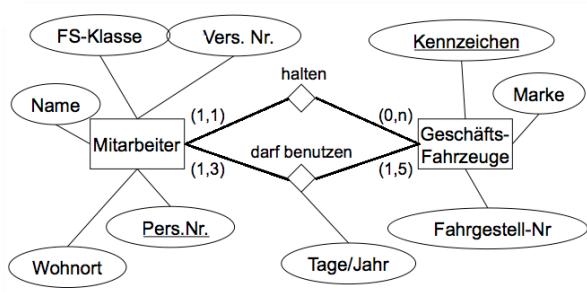


Bild 6.8: Beziehungen

- Multiplizität (Mindest~, Maximal~, Synonym: Kardinalität)
  - Komplexität (Anzahl der Entitätstypen, die an einem Beziehungstyp beteiligt sind, hier jeweils 2)
- Attribute für Beziehungen
- Rekursive (synonym reflexive) Beziehung mit Rollennamen

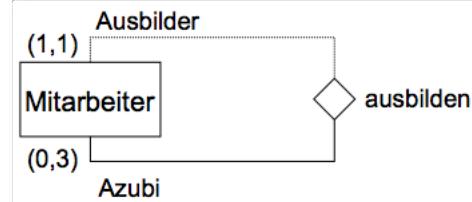


Bild 6.9: Rekursive Beziehung

## 6.2.6 Kardinalitäten von Beziehungen

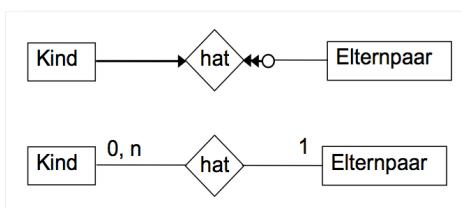


Bild 6.10: Beispiel für verschiedene Notationen für Kardinalitäten

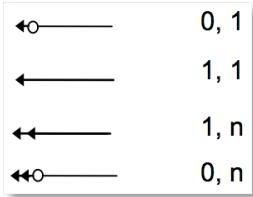
Pfeil-Notation:

### 1:1-Beziehung

Jede Seite der Beziehung lässt sich genau einer Entität zuordnen. Beide Entitäten haben die Kardinalität 1.

### 1:n-Beziehung

Eine Entität in der Beziehung ist vielen anderen Entitäten zugeordnet, d. h., eine Seite der Beziehung hat eine Kardinalität größer als 1.



### n:m-Beziehung

Auf beiden Seiten der Beziehung sind mehrere Entitäten beteiligt, d. h., beide Seiten haben eine Kardinalität größer als 1.

Bild 6.11: Alternative Pfeil-Notation für Kardinalitäten

Krähenfußnotation-Notation:

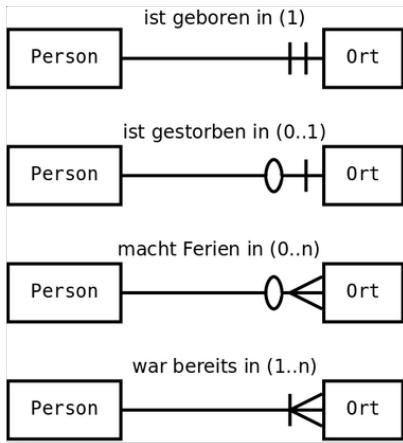


Bild 6.12: Alternative Krähenfußnotation-Notation  
(Martin-Notation) für Kardinalitäten

Schließlich findet man bei vielen Autoren auch die abkürzende Schreibweise \* statt (0, \*).

Beispiel n:m-Beziehung:

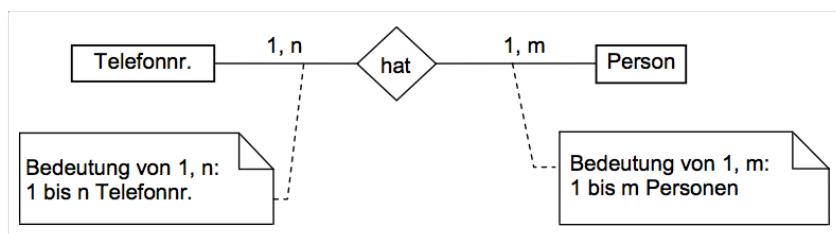


Bild 6.13: N:M-Beispiel

## 6.2.7 Vererbungsbeziehungen

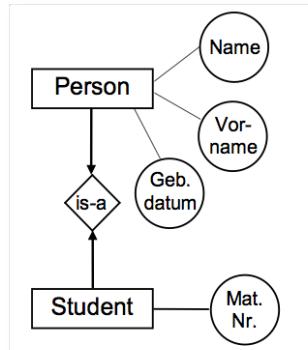


Bild 6.14: Is-a Beispiel

Vererbung, bzw. Generalisierung und Spezialisierung, wird wie bei der Objektorientierung verwendet, um auszudrücken, dass Gemeinsamkeiten (d. h. gemeinsame Attribute) von Entitätstypen in einem separaten, übergeordneten Entitätstyp untergebracht sind, mit dem die ursprünglichen Entitätstypen eine besondere Beziehung eingehen. Dadurch wird die redundante Modellierung in beiden Entitätstypen vermieden und gleichzeitig ihre Gemeinsamkeit explizit im Modell deutlich gemacht.

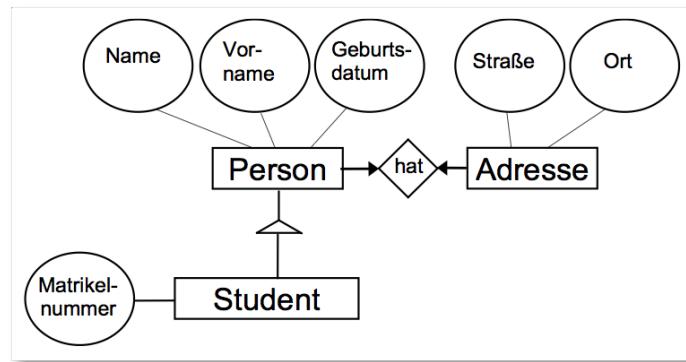


Bild 6.15: Is-a Beziehung: Neue Notation

Damit gelöstes Problem: Teilweise "leere" Attribute

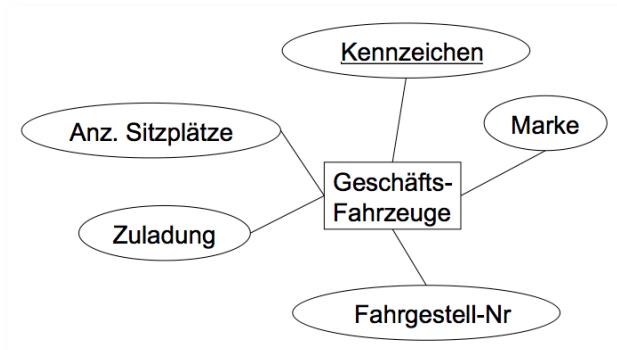


Bild 6.16: Beispiel: Teilweise "leere" Attribute

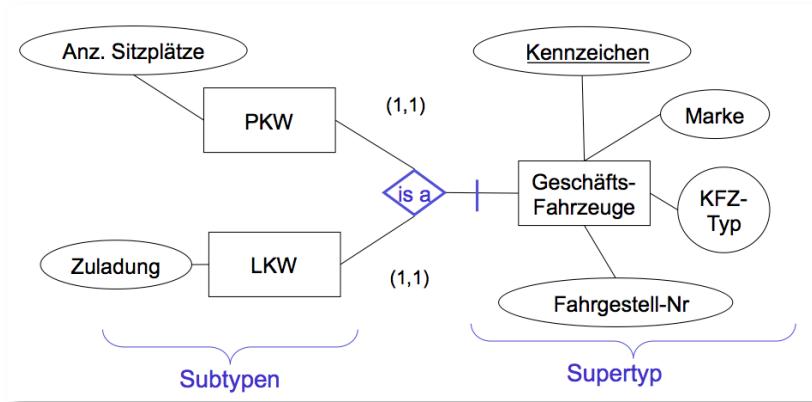


Bild 6.17: Is-a Vererbungsbeziehung

## 6.2.8 Heuristiken für Entitäten

Die folgenden Regeln können helfen, zu entscheiden, ob "etwas" eher Entität (E1-E4), Attribut (A1-A3) oder Beziehung (B1-B3) ist.

### E1: Substantiv

Wenn die verbale Beschreibung ein Substantiv enthält, dann versuchen Sie es als Entität zu modellieren.

### E2: Im Auge behalten

Wenn Sie irgendetwas im Auge behalten wollen, dann ist dieses "irgendetwas" wahrscheinlich eine Entität.

### E3: Brett vorm Kopf

Wenn Sie sich nicht entscheiden können, ob "irgendetwas" eine Entität, ein Attribut oder eine Beziehung ist, dann modellieren Sie "irgendetwas" vorläufig als Entität.

### E4: ID

Wenn der Name eines Datenelementes auf -name, -nummer, -code endet, dann kann es sich um den Schlüssel einer Entität handeln.

### A1: Wert

Wenn etwas genau einen Wert annehmen kann, dann ist es ein Attribut und keine Entität.

### A2: Zuordnung

Wenn dieselben Attributwerte (oder auch nur einer) in mehreren Entitätsausprägungen gleich auftauchen, dann benutzen Sie Fremdschlüssel und ordnen Sie diese Attribute per Beziehung zu.

### A3: Null-Werte vermeiden

Wenn das zugeordnete Attribut nicht zu allen Ausprägungen eines Entitätstyps passt, dann versuchen Sie den Entitätstyp in mehrere Entitätstypen zu unterteilen und das Attribut einer Unterteilung zuzuordnen (**Schwache Entität**).

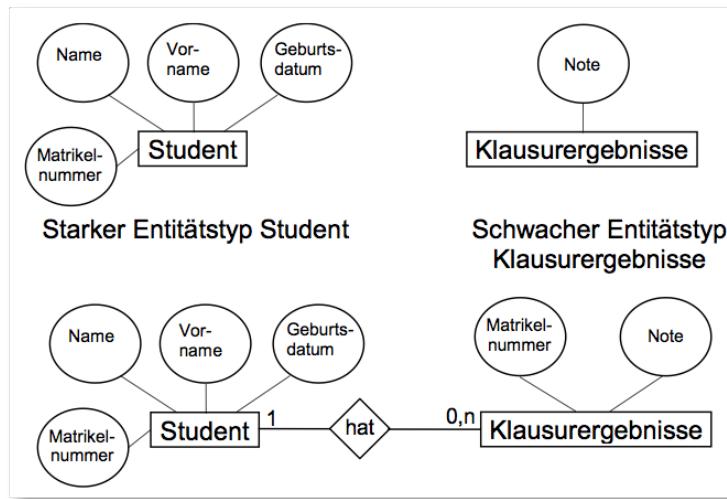


Bild 6.18: Starke und schwache Entitätstypen

Merke:

- Schwache Entitäten können/dürfen ohne ihre starken Beziehungspartner nicht weiter existieren. Sie müssen dann gelöscht werden.
- Starke und schwache Entität besitzen i.d.R. die gleichen Schlüssel
- Starke und schwache Entität besitzen keine gemeinsamen Attribute

#### R1: Verb

Wenn eine verbale Beschreibung ein Verb enthält, dann versuchen Sie, eine Beziehung zu modellieren.

#### R2: Isolierte Entität

Wenn es im ERD eine unverbundene Entität gibt, dann suchen Sie nach Relationen zu anderen Entitäten.

#### R3: Fremdschlüssel

Wenn in einem logischen Satz zwei oder mehr eindeutige Schlüssel enthalten sind, dann ist darin vermutlich eine Beziehung verborgen.

### 6.2.9 Übungen

#### Übung ER.1

Waren werden an Kunden verschickt.

#### Übung ER.2

Bei uns arbeiten an jedem Projekt mindesten zwei Mitarbeiter.

#### Übung ER.3 (Tierpflege)

In einem Zoo gibt es verschiedene Tierpfleger, die für die Pflege der Tiere zuständig sind. Jeder Tierpfleger besitzt zur Identifizierung eine eindeutige Nummer.

Jedes Tier befindet sich in einer bestimmten Unterkunft. Diese Unterkunft kann ein Freigehege, ein Löwenkäfig oder eine Sumpfanlage sein. Jedes Tier besitzt einen Namen und zur Identifizierung eine eindeutige Nummer. Die Unterkünfte der jeweiligen Tiere besitzen ebenfalls eine Nummer.

Für bestimmte Tiere können nun mehrere Tierpfleger zuständig sein. Andererseits betreut aber jeder Tierpfleger mehrere Tiere. Um die Auslastung der einzelnen Tierpfleger zu erkennen und auszuwerten, wer mit welchem Tier am besten zurecht kommt, wird der Pflegeaufwand pro Tier festgehalten.

### Übung ER.4 (Internat, Hausaufgabe)

Zeichnen Sie bitte ein ERM-Diagramm für folgende Sachverhalte. Attribute müssen nicht gezeichnet werden.

- Ein Betreuer wird in der Regel in ein bis zwei Fächern ausgebildet, z.B. Physik. Aber es gibt auch Autodidakten ohne entsprechende Ausbildung.
- Ein Betreuer betreut mindestens fünf und höchstens zwölf Schüler. Umgekehrt hat jeder Schüler einen festen Betreuer, der für seine Ausbildung verantwortlich ist.
- Im Internat werden für die Freizeitgestaltung verschiedene Kurse, z.B. Basteln, Fußball, etc. angeboten. Für einen Kurs schreiben sich bis zu 30 Schüler ein. Ein Kurs existiert auch dann, wenn sich überhaupt kein Schüler dafür einschreibt.
- Jeder Kurs wird nur von einem Betreuer geleitet.
- Außerdem gibt es gemeinnützige Aufgaben, wie z.B. Rasenmähen. Ein Schüler muss mindestens eine Aufgabe übernehmen, und alle Aufgaben müssen erledigt werden.

## 6.3 Physikalische Abbildung von Entity-Relationship-Modellen

Ausgangspunkt des Datenbank-Entwurfs ist das Datenmodell der Systemanalyse, welches mit Hilfe der Entity-Relationship-Modellierung erstellt wurde. Dieses Datenmodell ist nun auf Relationen, die in einem relationalen Datenbankmanagementsystem (RDBMS) gespeichert werden, und in Funktionen einer Datenbank umzusetzen.

Die Abbildung der verschiedenen Beziehungen zwischen Entitäten auf eine relationale Datenbank wird in den folgenden Unterkapiteln vorgestellt.

### 6.3.1 Datenbankentwurf für 1:1-Beziehungen

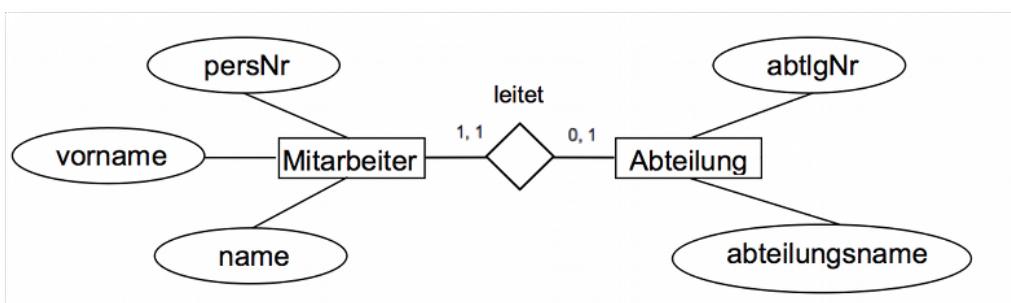


Bild 6.19: 1:1-Beziehung

#### Möglichkeit 1: Entwurf von zwei Relationen

MITARBEITER (persNr, name, vorname)

ABTEILUNG (abtlgNr, abteilungsname, #leiterPersNr)

## Möglichkeit 2: Entwurf einer einzigen Relation

ABTEILUNG (abtlgNr, abteilungsname, ltrName, ltrVorname)

Dies ist allerdings nur sinnvoll, wenn der Entitätstyp MITARBEITER keine weiteren Beziehungen eingeht und deshalb nicht eigenständig zu modellieren ist.

### 6.3.2 Datenbankentwurf für Is-a-Beziehungen (Vererbung)

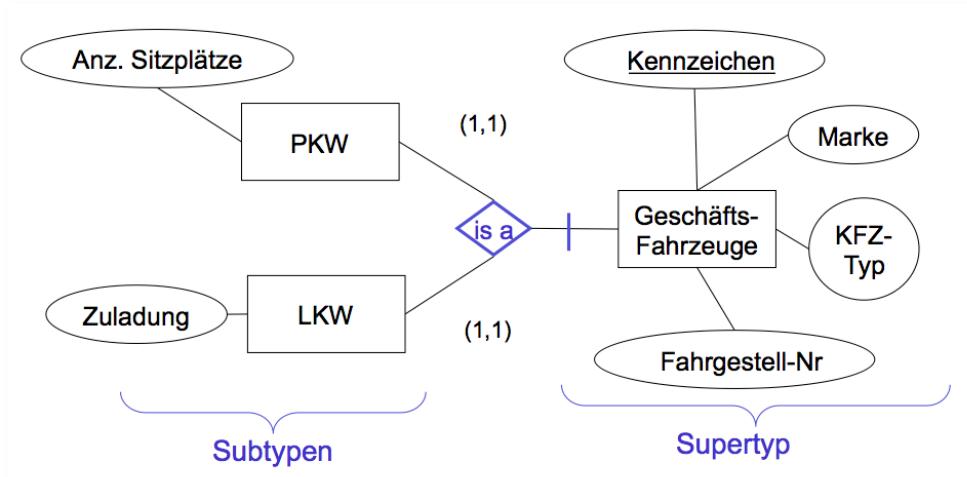


Bild 6.20: Is-a-Beziehung

Drei Möglichkeiten:

1. Eine gemeinsame Relation mit teilweise leeren Attributen
2. Für jeden Subtyp werden Attribute des Supertyps mit übernommen
3. Für Supertyp und jeden Subtyp eine Relation

#### Möglichkeit 1: Gemeinsame Relation

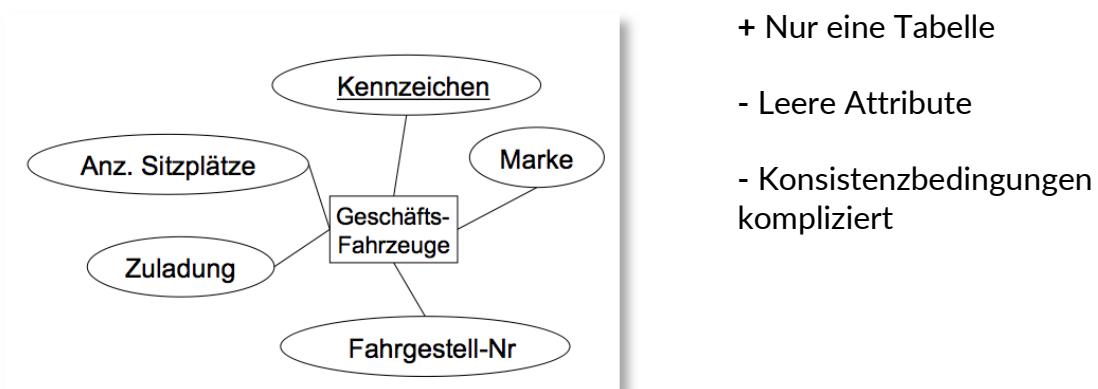
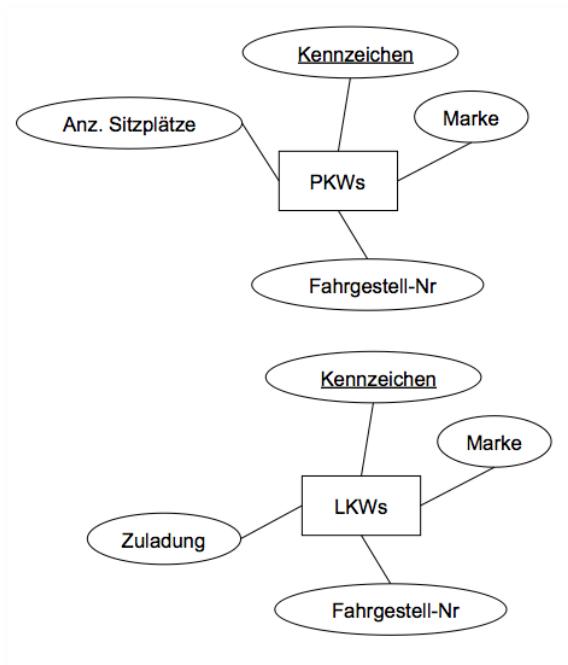


Bild 6.21: Is-a durch Gemeinsame Relation

#### Möglichkeit 2: Attributübernahme

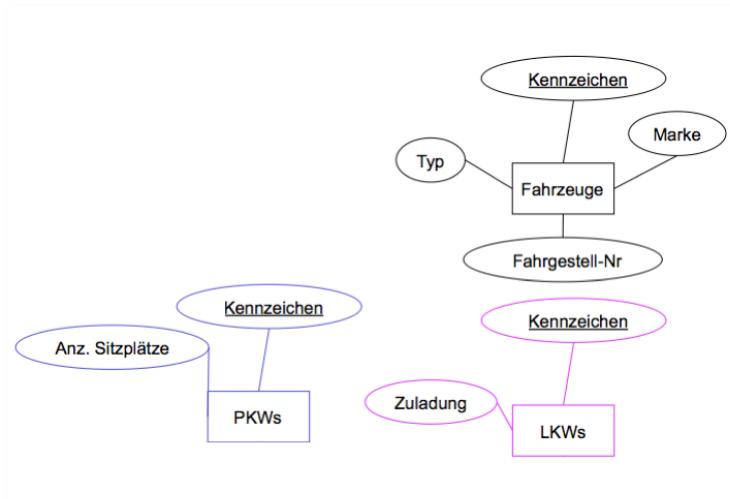


+ Keine leeren Attribute

- Supertyp muss an mehreren Stellen gepflegt werden

Bild 6.22: Is-a durch Attributübernahme

### Möglichkeit 3: Je eine Relation



+ Klare Semantik

+ Konsistente Pflege  
Supertyp

- Schlüssel mehrfach

- Zusätzliches  
Partitionierungsattribut  
(Typ)

Bild 6.23: Is-a durch je eine Relation

### 6.3.3 Datenbankentwurf für 1:n-Beziehungen

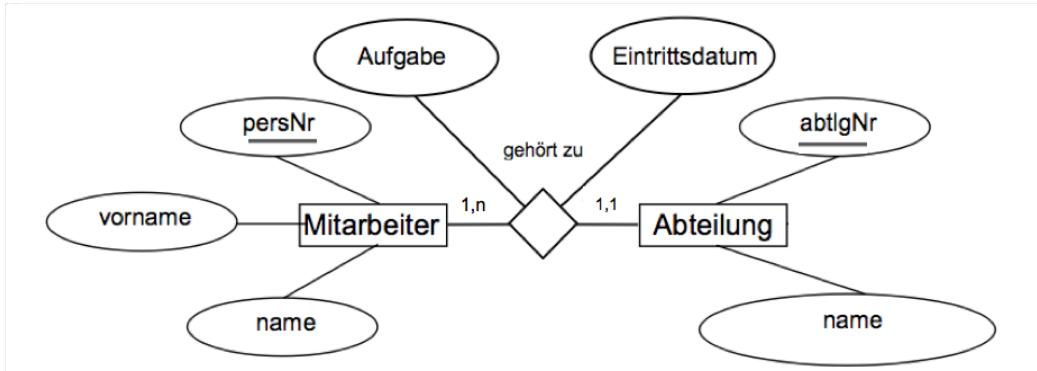


Bild 6.24: 1:n Beziehung

#### Standardmethode:

2 Tabellen, Beziehungs-Attribute wandern auf die "n"-Seite.

MITARBEITER (persNr, name, vorname, #abtlgNr, **eintritt**, **aufgabe**)

ABTEILUNG (abtlgNr, name)

#### Proaktive Methode:

Gleich als n:m-Beziehung modellieren, siehe nächsten Abschnitt. Dann bleiben die Beziehungsattribute bei der Beziehung und man wird nicht überrascht, wenn doch mal ein Mitarbeiter in zwei Abteilungen arbeitet.

### 6.3.4 Datenbankentwurf für n:m-Beziehungen

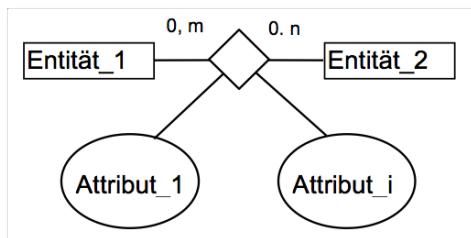


Bild 6.25: Allgemeine n:m Beziehung

#### Entwurf für allgemeine n:m-Beziehungen ( $n > 1, m > 1$ ):

- Die beiden Entitäten werden direkt in zwei Relationen Relation\_1 und Relation\_2 überführt.
- Die Beziehung zwischen der Relation\_1 und der Relation\_2 wird durch eine neue dritte Relation ausgedrückt. Dort werden auch die Beziehungsattribute gespeichert. Man nennt die dritte Relation auch oft *Link-Relation*.
- Der Primärschlüssel der dritten Relation setzt sich aus den Primärschlüsselattributen der Relation\_1 und aus den Primärschlüsselattributen der Relation\_2 zusammen oder es werden Surrogatschlüssel eingesetzt.

Beispiel:

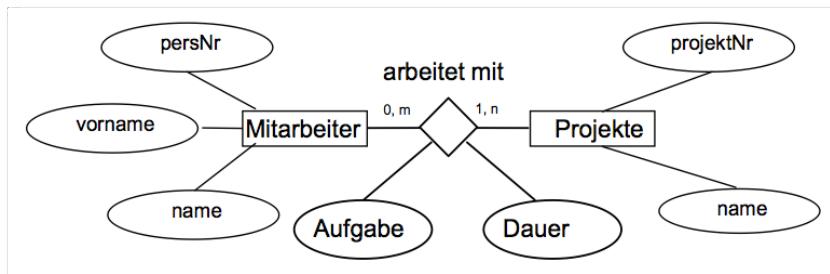


Bild 6.26: Beispiel für n:m Beziehung

MITARBEITER (persNr, name, vorname)

PROJEKTE(projektNr, name)

PROJEKTE\_MITARBEITER(#projektNr, #persNr, aufgabe, dauer)

### 6.3.5 Datenbankentwurf: wichtige Kleinigkeiten

Falls in dem Projekt nicht ohnehin festgelegt, lohnt es sich spätestens jetzt, folgende Dinge zu entscheiden und die Entscheidungen zu dokumentieren.

#### Festlegen der Benennung von Schlüsseln

- *Primärschlüssel* sollten nach Möglichkeit möglichst oft gleich heißen, zum Beispiel "id". Wenn die Anwendung jedoch "natürliche" Schlüssel vorgibt, wie zum Beispiel eine Matrikelnummer, so sollte der Name wegen der besseren Verständlichkeit beibehalten werden.
- *Fremdschlüssel* sollten durch einen Suffix (z.B. \_id, wird von vielen Frameworks, z.B. Ruby on Rails so gemacht) oder Präfix (z.B. FK\_) als Fremdschlüssel kenntlich gemacht werden. Das entbindet einen aber nicht, im SQL-Datenbankentwurf auch die entsprechenden FOREIGN KEY CONSTRAINTs zu definieren.

#### Attribut- und Entitätsnamen

- Wahl der *Sprache* für Attribut- und Entitätsnamen. Hier ist Englisch die erste Wahl, auch einfach deshalb, weil dort keine Umlaute und Ligaturen verwendet werden. Auch das Arbeiten in internationalen Teams erfordert dies. Allerdings muss dann eine klare Dokumentation der Übersetzungen bereitgestellt werden.
- Entscheidungen für *Groß-/Kleinschreibung*: Die meisten DBMS ignorieren die Groß-/Kleinschreibung, aber halt nicht alle. Deshalb sollte man klare Regeln haben, wie zum Beispiel diese:  
"Entitäten: Erster Buchstabe in Großschreibung der Rest klein, Attribute: alles in Kleinschreibung, SQL-Schlüsselworte komplett in Großschreibung."
- *Sonderzeichen*: Zwar akzeptieren einige Datenbanken UTF-Enkodierung von Bezeichnern, aber man sich sollte trotzdem wegen der Portabilität auf die Zeichen [A-Z], [a-z], [0-9\_] beschränken. Umlaute, Ligaturen und sonstige nationale Sonderzeichen sollte man vermeiden.

- **Teilbegriffstrenner:** Legen Sie fest, wie zusammengesetzte Begriffe geschrieben werden: Einfach zusammen (`matrikelnummer`), Kamelhöckerschreibweise (`matrikelNummer`), oder mit `"_`-Trennung (`matrikel_number`). Bindestriche (`"-`) sind meist nicht zulässig ...
- **Plural, singular:** Für die Entitätsnamen nimmt man gewöhnlich den Plural (`nationalities`), für Attribute den Singular (`street`). Festzulegen ist, wie bei Fremdschlüsseln vorgegangen wird: Hier lautet die Empfehlung, den Entitätsnamen (also im Plural) gefolgt vom Suffix `_id` zu nehmen (`nationalities_id`), da man sonst noch die englischen Regeln für die Plural-/Singularbildung einfließen lassen muss (`nationality_id`), was aber auch oft gemacht wird.
- **Präfixe und Suffixe:** neben den Suffixen für Fremdschlüssel sollte man noch festlegen, wie andere benannte Objekte der Datenbank, zum Beispiel Foreign-Key-Constraints (Beispiel: `FK_students2lectures`), Sequenzen (`SEQ_students`), durch Präfixe oder Suffixe gekennzeichnet werden. Viele Datenbanksysteme legen diese Namen automatisch fest, doch sollte man sich immer die Mühe machen, diese Dinge selbst zu benennen, so dass man zum Beispiel bei Stacktraces nach Ausnahmen sofort sehen kann, welcher Constraint verletzt worden ist.

### 6.3.6 Übungen

#### Übung PHY.1

Erstellen Sie bitte den Datenbankentwurf zu Übung ER.3 (Tierpflege) als Relationenschema.

#### Übung PHY.2

Studenten (MatrNr, Vorname, Name) schreiben Abschlussarbeiten (Thema, Startdatum) die von Professoren (PersonalNummer, Vorname, Name) betreut werden.

PHY.2.1: Entwerfen Sie bitte das entsprechende ER-Diagramm.

PHY.2.2: Erstellen Sie bitte den entsprechenden Datenbankentwurf als Relationenschema.

#### Übung PHY.3 (Hausaufgabe)

Erstellen Sie bitte den Datenbankentwurf zu Übung ER.4 (Internat) als Relationenschema.

#### Übung PHY.4 (Querbezug MMK, Hausaufgabe)

PHY.4.1: Finden Sie zwei typische GUI-Elemente, mit denen **1:1-Beziehungen** realisiert werden.

PHY.4.2: Finden Sie zwei typische GUI-Elemente, mit denen **1:n-Beziehungen** realisiert werden.

PHY.4.3: Finden Sie zwei typische GUI-Elemente, mit denen **n:m-Beziehungen** realisiert werden.

PHY.4.4: Finden Sie zwei typische GUI-Elemente, mit denen **Is-a-Beziehungen** realisiert werden.

Bitte dokumentieren Sie Ihr Ergebnis in einer PDF-Datei.

# 7 Normalisierung

## 7.1 Einführung

Die **Normalisierung** hilft beim fachgerechten Entwurf von relationalen Datenmodellen. Bei Beachtung der Normalisierungsregeln werden Redundanzen beseitigt und Anomalien und Nullwerte vermieden.

In diesem Kapitel werden die in der Praxis geläufigen Erste Normalform (1NF) bis Dritte Normalform (3NF) behandelt. Es gibt in der Literatur weitere Normalformen, die aus Zeitgründen leider nicht behandelt werden können (Boyce-Codd-Normalform (BCNF), Vierte Normalform (4NF), Fünfte Normalform (5NF)).

### 7.1.1 Ziele der Normalisierung

- Fakten in verständlicher, "logischer" Weise repräsentieren.
- Redundanzen vermeiden.
- Performance erhöhen.

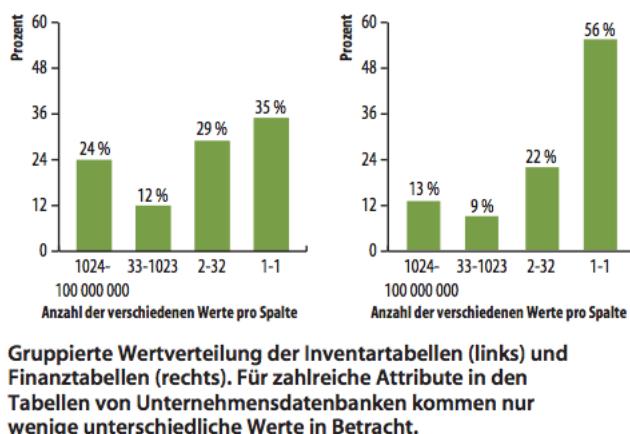


Bild 7.1: Gruppierte Wertverteilung

- Nullwerte vermeiden.
- Anomalien vermeiden.
- **Bewusster Verstoß** gegen die Normalisierung: Wenn man z.B. weiß, wo im Datenmodell Redundanzen sind, weiß man, dass diese im Programmcode gepflegt werden müssen und kann das zur Qualitätssicherung überprüfen.

### 7.1.2 Probleme der Normalisierung

- Performance kann sinken (bei sehr großen Datenbeständen, die verteilt werden müssen).
- Übersicht geht verloren.
- Man muss SQL-JOINS können.

### 7.1.3 Probleme mit Nullwerten

- Redundanz.
- Verschiedene, auslegungsbedürftige Bedeutungen von Nullwerten:
  - Das Attribut ist auf diesen Tupel nicht *anwendbar*.
  - Der Attributwert ist *unbekannt*.
  - Der Attributwert ist *noch nicht erfasst*.
  - Der Attributwert ist 0 (bei Zahlen-Attributen).

### 7.1.4 Anomalien

| PersNr | Name    | Abt Kürzel | AbtName      | ProjektNr  | ProjektName | Zeit   |
|--------|---------|------------|--------------|------------|-------------|--------|
| 1      | Mayer   | EP         | Entwicklung  | 4711, 4712 | A, B        | 20,25  |
| 2      | Schulze | EP         | Entwicklung  | 4713       | C           | 100    |
| 3      | Müller  | KT         | Konstruktion | 4714, 4713 | D, C        | 30, 15 |
| 4      | Groß    | VT         | Vertrieb     | 4715       | E           | 40     |

Bild 7.2: Personalverwaltung: Nicht-normalisierte Relation

#### Update-Anomalie:

Wenn im obigen Bild für die Abteilung "Entwicklung" das Kürzel auf "EN" geändert wird, muss dies an mehreren Zeilen geändert werden.

#### Insert-Anomalie:

Wenn in die Tabelle Personalverwaltung der neue Satz  
(5, 'Klein', 'EW', 'Entwicklung', 4712, 'E', 50)  
eingefügt wird, so haben wir ein weiteres Kürzel für die Abteilung "Entwicklung".

#### Delete-Anomalie:

Wenn in der ursprünglichen Tabelle zur Personalverwaltung der letzte Satz gelöscht wird, dann ist nicht nur Herr Groß weg aus der Firma, sondern auch das Wissen, dass "Vertrieb" mit "VT" abgekürzt wird.

## 7.2 Normalformen

### 7.2.1 Übersicht

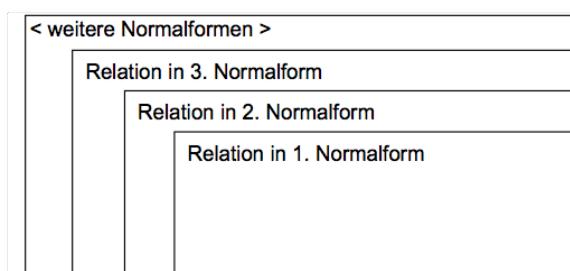


Bild 7.3: Übersicht über die Normalformen

- Jede Normalform enthält implizit die vorhergehenden Normalformen.
- So enthält die dritte Normalform die zweite und damit auch die erste Normalform.
- Um auf eine Normalform zu kommen, müssen nicht zwangsläufig die vorgehenden Normalformen durchlaufen werden. Man kann mit etwas Übung direkt auf die 3. Normalform kommen.

## 7.2.2 Erste Normalform (1NF)

### Definition:

Eine Relation befindet sich in der 1. Normalform, wenn alle zugrundeliegenden Wertebereiche der Attribute nur *atomare Werte* enthalten.

| PersNr | Name    | Abt Kürzel | AbtName      | ProjektNr | ProjektName | Zeit |
|--------|---------|------------|--------------|-----------|-------------|------|
| 1      | Mayer   | EP         | Entwicklung  | 4711      | A           | 20   |
| 1      | Mayer   | EP         | Entwicklung  | 4712      | B           | 25   |
| 2      | Schulze | EP         | Entwicklung  | 4713      | C           | 100  |
| 3      | Müller  | KT         | Konstruktion | 4714      | D           | 30   |
| 3      | Müller  | KT         | Konstruktion | 4713      | C           | 15   |
| 4      | Groß    | VT         | Vertrieb     | 4715      | E           | 40   |

Bild 7.5: Personalverwaltung: Erste Normalform

### Möglichkeiten, die erste Normalform zu erreichen:

1. Nicht-atomare Attribute in extra Tabelle auslagern und dann wie bei n:m-Beziehungen vorgehen. Bester Ansatz.
2. Wie im Bild oben: Tupel mit nicht-atomaren Attributen für jedes Atom der nicht-atomaren Attribute wiederholen. Man erhält dann einen *zusammengesetzten Schlüssel*: {PersNr, ..., ProjektNr, ProjektName, Zeit}
3. Wenn man die Maximalanzahl  $m$  der Atome in nicht-atomaren Attributen kennt, kann man  $m$  Spalten anlegen: {..., ProjektNr1, ProjektNr2, ...}. Meist ist dieses  $m$  aber nicht so fest wie man zu Anfang glaubt ...

## 7.2.3 Funktionale Abhängigkeit

### Definition *funktional abhängig*:

Sei  $R$  eine Relation und  $X$  und  $Y$  beliebige Teilmengen von Attributen von  $R$ . Dann heißt  $Y$  *funktional abhängig* von  $X$  genau dann, wenn zu jeder Wertekombination von  $X$  genau eine Wertekombination von  $Y$  gehört.

Schreibweise:  $X \rightarrow Y$

### Funktional abhängig verbal:

Funktional abhängig bedeutet, dass nicht-Primärschlüsselattribute ( $Y$  in der Definition) vom Primärschlüssel ( $X$  in der Definition) abhängen, oder anders formuliert, wenn der Primärschlüssel eindeutig die Werte der Nicht-Primärschlüsselattribute bestimmt.

### **Definition trivial funktional abhängig:**

Falls Y Teilmenge von X ist, dann nennt man die funktionale Abhängigkeit  $X \rightarrow Y$  trivial.

Verbal: Teile eines zusammengesetzten Schlüssels sind von diesem funktional abhängig.

### **Beispiele**

Gegeben sei folgende Relation:

PERSON = {Id, Name, Vorname, Strasse, Nr, Plz, Ort, Land}.

1. Ist  $X=\{\text{Id}\}$  so gilt:

$$X \rightarrow \text{PERSON}.$$

Diese Abhängigkeit ist nicht trivial.

2. Ist  $X=\{\text{Name, Vorname}\}$  so gilt **nicht**:

$$X \rightarrow \text{PERSON}.$$

3. Ist  $X=\{\text{Name, Vorname, Nr, Ort}\}$  und  $Y=\{\text{Name, Ort}\}$  so gilt:

$$X \rightarrow Y.$$

Diese Abhängigkeit ist trivial.

4. Ist  $X=\{\text{Strasse, Nr, Ort, Land}\}$  und  $Y=\{\text{Plz}\}$  so gilt:

$$X \rightarrow Y.$$

Diese Abhängigkeit ist nicht trivial.

**Übung FA.1:** Warum ist das jeweils so?

**Übung FA.2**

Gegeben sei folgende Relation:

STUDENT = {MatrNr, Name, Vorname, Land, Fakultaet\_id, FakultaetsName, Studiengang\_id, Semester, IstImVorstudium, Login, MifareId}.

**FA.2.1:**

Gilt  $\text{STUDENT} \rightarrow \text{STUDENT}$ ? Warum?

**FA.2.2:**

Gilt  $\{\text{MatrNr}\} \rightarrow \{\text{Name, Vorname}\}$ ? Warum?

**FA.2.3:**

Gilt  $\{\text{MatrNr, Land}\} \rightarrow \{\text{Name, Vorname, Land}\}$ ? Warum?

**FA.2.4:**

Gilt  $\{\text{Studiengang_id}\} \rightarrow \{\text{FakultaetsName}\}$ ? Warum?

**FA.2.5:**

Gilt  $\{\text{Name, Vorname, Land}\} \rightarrow \{\text{Fakultaet_id}\}$ ? Warum?

**FA.2.6:**

Finden Sie alle X, Y mit  $|X| == |Y| == 1$  UND  $X \rightarrow Y$ .

## 7.2.4 Zweite Normalform (2NF)

| PersNr | Name    | Abt Kürzel | AbtName      | <u>ProjektNr</u> | ProjektName | Zeit |
|--------|---------|------------|--------------|------------------|-------------|------|
| 1      | Mayer   | EP         | Entwicklung  | 4711             | A           | 20   |
| 1      | Mayer   | EP         | Entwicklung  | 4712             | B           | 25   |
| 2      | Schulze | EP         | Entwicklung  | 4713             | C           | 100  |
| 3      | Müller  | KT         | Konstruktion | 4714             | D           | 30   |
| 3      | Müller  | KT         | Konstruktion | 4713             | C           | 15   |
| 4      | Groß    | VT         | Vertrieb     | 4715             | E           | 40   |

Anomalie beim Einfügen folgender Zeile?

|   |       |    |          |      |   |    |
|---|-------|----|----------|------|---|----|
| 5 | Klein | VT | Vertrieb | 4712 | E | 50 |
|---|-------|----|----------|------|---|----|

Bild 7.6: Anomalie durch Einfügen in 1NF-Relation?

| PersNr | Name    | Abt Kürzel | AbtName      | <u>ProjektNr</u> | ProjektName | Zeit |
|--------|---------|------------|--------------|------------------|-------------|------|
| 1      | Mayer   | EP         | Entwicklung  | 4711             | A           | 20   |
| 1      | Mayer   | EP         | Entwicklung  | 4712             | B           | 25   |
| 2      | Schulze | EP         | Entwicklung  | 4713             | C           | 100  |
| 3      | Müller  | KT         | Konstruktion | 4714             | D           | 30   |
| 3      | Müller  | KT         | Konstruktion | 4713             | C           | 15   |
| 4      | Groß    | VT         | Vertrieb     | 4715             | E           | 40   |

Bild 7.7: Anomalie durch Einfügen in 1NF-Relation

### Definition:

Eine Relation ist in 2NF, wenn jedes nicht dem Schlüssel angehörende Attribut funktional abhängig ist vom Gesamtschlüssel, nicht aber von einzelnen Schlüsselteilen.

### Folgerung:

Eine 1NF-Relation mit einem einteiligen Primärschlüssel ist immer in der 2 NF.

| Mitarbeiter |         |            |              | Projekte  |             | Kapazitäten |           |      |
|-------------|---------|------------|--------------|-----------|-------------|-------------|-----------|------|
| PersNr      | Name    | Abt Kürzel | AbtName      | ProjektNr | ProjektName | PersNr      | ProjektNr | Zeit |
| 1           | Mayer   | EP         | Entwicklung  | 4711      | A           | 1           | 4711      | 20   |
| 2           | Schulze | EP         | Entwicklung  | 4712      | B           | 1           | 4712      | 25   |
| 3           | Müller  | KT         | Konstruktion | 4713      | C           | 2           | 4713      | 100  |
| 4           | Groß    | VT         | Vertrieb     | 4714      | D           | 3           | 4714      | 30   |
|             |         |            |              | 4715      | E           | 3           | 4713      | 15   |
|             |         |            |              |           |             | 4           | 4715      | 40   |

Bild 7.8: 2NF-Relation

### Möglichkeiten, die zweite Normalform zu erreichen:

- 1. Kein Weg:** Durch künstliche, zusätzliche, einstellige Schlüssel die Definition von 2NF austricksen.

## 2. Ansatz:

- Attribute, die nur von einem Teilschlüssel abhängen, zusammen mit diesem in eine neue Relation auslagern (Projekte im Bild oben).
- Attribute wie Zeit, die sich sowohl auf den verbleibenden Rest {Persnr, Name, AbtKürzel, Abtname} als auch die neue Relation Projekte beziehen, werden als Verbindungs-Attribute zu einer dritten Link-Relation zur Modellierung der n:m-Beziehung zwischen Mitarbeiter und Projekte hinzugefügt.

Anomalie beim Einfügen folgender Zeile in die Relation Mitarbeiter?

|   |       |    |             |
|---|-------|----|-------------|
| 5 | Klein | VT | Entwicklung |
|---|-------|----|-------------|

Bild 7.10: Anomalien bei 2NF-Relation?

| Mitarbeiter |         |            |              |
|-------------|---------|------------|--------------|
| PersNr      | Name    | Abt Kürzel | AbtName      |
| 1           | Mayer   | EP         | Entwicklung  |
| 2           | Schulze | EP         | Entwicklung  |
| 3           | Müller  | KT         | Konstruktion |
| 4           | Groß    | VT         | Vertrieb     |
| 5           | Klein   | VT         | Entwicklung  |

Fkt. Abhängigkeit von  
Nicht-Schlüssel-Attribut

Bild 7.11: Anomalien bei 2NF-Relation

## 7.2.5 Dritte Normalform (3NF)

### Definition:

Eine Relation ist in 3NF, wenn sie in 2NF ist und jedes Attribut direkt vom Schlüssel abhängig ist.

Man sagt auch: Nicht-Schlüsselattribute dürfen nur von Schlüsseln, nicht aber von anderen Nicht-Schlüsselattributen abhängen.

| Mitarbeiter |         |            | Projekte  |              | Kapazitäten |           |      |
|-------------|---------|------------|-----------|--------------|-------------|-----------|------|
| PersNr      | Name    | Abt Kürzel | ProjektNr | Projekt Name | PersNr      | ProjektNr | Zeit |
| 1           | Mayer   | EP         | 4711      | A            | 1           | 4711      | 20   |
| 2           | Schulze | EP         | 4712      | B            | 1           | 4712      | 25   |
| 3           | Müller  | KT         | 4713      | C            | 2           | 4713      | 100  |
| 4           | Groß    | VT         | 4714      | D            | 3           | 4714      | 30   |
|             |         |            | 4715      | E            | 3           | 4713      | 15   |
|             |         |            |           |              | 4           | 4715      | 40   |

| Abteilungen |              |
|-------------|--------------|
| Abt Kürzel  | AbtName      |
| EP          | Entwicklung  |
| KT          | Konstruktion |
| VT          | Vertrieb     |

Bild 7.12: 3NF-Relation

Möglichkeit, die dritte Normalform zu erreichen:

- Attribute, die von einem Nicht-Schlüssel-Attribut abhängen, zusammen mit diesem in eine neue Relation auslagern (Abteilungen im Bild oben).
- Das Nicht-Schlüssel-Attribut (AbtKürzel im Bild) wird zum Primärschlüssel der neuen Relation und zum Fremdschlüssel in der alten Relation (Mitarbeiter), aus der die Attribute entfernt wurden.

## 7.3 Zusammenfassung und Beispiele

### 7.3.1 Übersicht Normalformen (1NF ... 3NF)

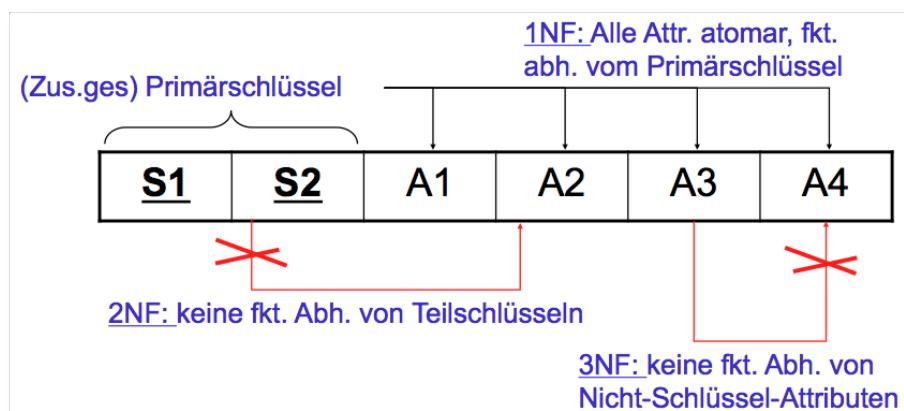


Bild 7.13: Übersicht

## 7.3.2 Übungen

### Übung NRM.1

| Fehlerakte |           |               |            |                    |                   |              |               |            |             |  |
|------------|-----------|---------------|------------|--------------------|-------------------|--------------|---------------|------------|-------------|--|
| Motornr    | Motor typ | Montage datum | Stationsnr | Stations typ       | Prüf datum        | Merk malcode | Merk maltext  | Grenz wert | Ist wert    |  |
| 98180      | M111      | 5.10.13       | L10; L12   | Lecktest; Lecktest | 7.10.13; 14.10.13 | P_O          | Öldruck       | 0.83       | 0.85; 0.86  |  |
| 98186      | M129      | 7.10.13       | L12        | Lecktest           | 14.10.13          | P_H          | Wasserdruck   | 2.7        | 2.9         |  |
| 98189      | M111      | 6.10.13       | HT1        | Heißtest           | 13.10.13          | P_S          | Saugrohrdruck | 0.148      | 0.165       |  |
| 98191      | M161      | 14.10.13      | L12        | Lecktest           | 15.10.13          | P_O          | Öldruck       | 0.81       | 0.82        |  |
| 98113      | M111      | 24.9.13       | HT1; KT2   | Heißtest; Kalttest | 25.9.13; 30.9.13  | P_S          | Saugrohrdruck | 0.148      | 0.161; 1.59 |  |
| 98199      | M111      | 23.10.13      | L12        | Lecktest           | 30.10.13          | P_O          | Öldruck       | 0.83       | 0.85        |  |

Bild 7.14: Nicht-normalisierte Relation Fehlerakte

### NRM.1.1

Erstellen Sie bitte ein Relationenschema für die Fehlerakte in 3NF.

### Übung NRM.3 (Hausaufgabe)

Überführen Sie bitte Ihr Relationenschema aus Aufgabe PHY.2 (Tierpflege) in die 3NF.  
Begründen Sie für jede Relation Ihres Relationenschemas, dass 1NF ... 3NF erfüllt sind.

### Übung NRM.3 (Hausaufgabe)

Überführen Sie bitte Ihr Relationenschema aus Aufgabe PHY.3 (Internat) in die 3NF.  
Begründen Sie für jede Relation Ihres Relationenschemas, dass 1NF ... 3NF erfüllt sind.

# 8 Null-Werte

## 8.1 Einleitung

### 8.1.1 NULL-Werte

The screenshot shows a web page from arbeitsagentur.de. At the top, there's a logo and the text "arbeitsagentur.de" and "Ein Service der Bundesagentur für Arbeit". Below this is a red rectangular area. To the right is a photograph of a person's face. Underneath the photo is the text "SCHNELLÜBERSICHTEN | LABOUR" and a "Zurück" link. A sidebar on the left says "► Bundesrepublik Deutschland". The main content area contains a table with the following data:

| Berichtsmonat            | Januar 2005                |
|--------------------------|----------------------------|
| Region                   | Bundesrepublik Deutschland |
| <b>Arbeitslose</b>       | NULL                       |
| ± zum Vorjahresmonat     | NULL                       |
| <b>Arbeitslosenquote</b> | NULL                       |
| Vorjahresmonat           | NULL                       |
| <b>Gemeldete Stellen</b> | NULL                       |
| ± zum Vorjahresmonat     | NULL                       |

Bild 8.1: NULL-Werte sollten dem Endanwender nicht präsentiert werden, da sie leicht missverstanden werden können...

### 8.1.2 Einführung

- NULL-Werte sind entgegen ihrem Namen eben keine Werte.
- NULL-Werte zeigen an, dass ein Wert *fehlt*, zum Beispiel weil zur Zeit der Datenerfassung der Wert unbekannt war.
- NULL-Werte führen zu einer dreiwertigen Logik (wahr, falsch, unbekannt).
- Diese dreiwertige Logik führt zu:
  - Missverständnissen
  - und bei unvorsichtigem Arbeiten z.B. mit SELECTs auch zu "falschen" Ergebnissen.
- Daher sollen die NULL-Werte in diesem Kapitel genauer untersucht werden. Das Kapitel dient auch der Wiederholung wichtiger SQL-Anweisungen.
- Viele Beispiele und Ideen des Kapitels sind dem Artikel "Nulls: Nothing to Worry About" von Lex de Haan und Jonathan Gennick im Oracle Magazine, Juli/August 2005 entnommen.

### 8.1.3 Beispieldatensätze

Tabelle: DEPT

| DEPTNO | DNAME         | LOC         |
|--------|---------------|-------------|
| 10     | HQ            | Utrecht     |
| 20     | Sales         | Munising    |
| 30     | Manufacturing | Novosibirsk |

Tabelle: EMP

| EMPNO | ENAME    | JOB       | MGR | SAL  | COMM | DEPTNO |
|-------|----------|-----------|-----|------|------|--------|
| 100   | Norgaard | President |     | 5000 |      | 10     |
| 122   | Lewis    | Salesrep  | 120 | 1100 |      |        |
| 199   | Gennick  |           |     | 2200 |      | 10     |
| 111   | De Haan  | Clerk     | 110 | 2000 |      |        |
| 112   | Millsap  | Salesrep  | 110 | 1250 | 1400 | 20     |
| 110   | Adams    | Manager   | 100 |      | 1700 | 20     |
| 120   | Kolk     | Manager   | 100 | 2450 |      | 10     |
| 113   | Mcdonald | Salesrep  | 110 | 1500 |      | 20     |
| 121   | Wood     | Clerk     | 120 | 1300 |      | 10     |
| 130   | Morle    | Clerk     | 100 |      |      | 10     |

[SQL-Skript für diese Tabellen \(nulls.sql\) unter ILIAS ...](#)

## 8.2 Null-Werte und skalare Ausdrücke

### 8.2.1 Null-Werte und skalare Ausdrücke

Frage/Anforderung: Wie sieht es aus, wenn jeder Angestellte 1000 EUR mehr bekommt?

SQL:

```
SELECT
 EMPNO, ENAME, SAL,
 SAL + 1000 AS Plus
FROM EMP
```

Resultat:

| EMPNO | ENAME    | SAL  | Plus |
|-------|----------|------|------|
| 100   | Norgaard | 5000 | 6000 |
| 110   | Adams    | NULL | NULL |
| 111   | Dehaan   | 2000 | 3000 |
| ...   |          |      |      |
| 122   | Lewis    | 1100 | 2100 |
| 130   | Morle    | NULL | NULL |
| 199   | Gennick  | 2200 | 3200 |

- Skalare Ausdrücke, die eine NULL umfassen, sind ihrerseits NULL.
- Das führt im Beispiel zuvor dazu, dass Adams und Morle weiterhin ein unbekanntes Gehalt, also NULL erhalten.
- Für den Menschen, der die Bedeutung der Tabelle kennt, ist die Frage, ob es sich bei einem NULL-Gehalt um z.B. folgende Fälle handelt, klar:
  - Gehalt gibt es grundsätzlich nicht für diese Person oder
  - die Person hat ein Gehalt von 0,00 EUR oder
  - die Person hat ein unbekanntes Gehalt, das aber auch um 1000,00 EUR zu erhöhen ist.

Der DB-Server kann dies nicht wissen.

- Wenn man also solche Abfragen schreiben will, muss man wissen, welche geschäftliche Bedeutung der NULL-Wert beim Design hatte.

Frage/Anforderung: Wenn man z.B. weiß, dass NULL für ein Gehalt von 0,00 EUR steht, kann man mit der Funktion COALESCE dem NULL-Wert eine Bedeutung geben:

SQL:

```
SELECT EMPNO, ENAME, SAL,
 COALESCE(SAL, 0) + 1000
FROM EMP
```

Resultat:

| EMPNO | ENAME    | SAL  |      |
|-------|----------|------|------|
| 100   | Norgaard | 5000 | 6000 |
| 110   | Adams    | NULL | 1000 |
| 111   | Dehaan   | 2000 | 3000 |
| ...   |          |      |      |
| 122   | Lewis    | 1100 | 2100 |
| 130   | Morle    | NULL | 1000 |
| 199   | Gennick  | 2200 | 3200 |

COALESCE wählt aus seiner Parameterliste (die auch mehr als 2 Argumente haben kann) von links den ersten Wert aus, der nicht NULL ist.

(Engl. to coalesce: vereinigen, zusammenfügen, zusammenwachsen, sich verbinden).

Eine Art Umkehrfunktion zu COALESCE ist NULLIF(a, b), die NULL zurückgibt, wenn a==b ist. Damit kann man zum Beispiel die Division durch die Zahl 0 vermeiden, was sonst zu einer Ausnahme (Programmabbruch) führen würde:

Frage/Anforderung: 100 / (status-10) für alle Hersteller berechnen.

SQL:

```
SELECT 100/(status-10) FROM SUPPLIERS;
```

Resultat:

```
ERROR: division by zero
***** Fehler *****
ERROR: division by zero
SQL Status:22012
```

Frage/Anforderung: Bessere Variante:

SQL:

```
SELECT 100/NNULLIF(status-10, 0) FROM SUPPLIERS;
```

Resultat:

```
10
NULL
5
10
5
```

## 8.3 Null-Werte und die dreiwertige Logik

### 8.3.1 Einleitung



Bild 8.2: Dreiwertige Logik

- NULL-Werte sind insbesondere bei Booleschen Ausdrücken etwa bei
  - WHERE-,
  - HAVING- und
  - ON-Bedingungentückisch, da sie einen dritten Wert neben WAHR und FALSCH darstellen:  
**UNBEKANNT**.
- Dies führt zu einer dreiwertigen Logik.

### 8.3.2 Verknüpfungstabellen der dreiwertigen Logik

Logisches NICHT:

|   |  |        |
|---|--|--------|
| x |  | NOT(x) |
|---|--|--------|

|             |  |         |  |
|-------------|--|---------|--|
| -----+----- |  |         |  |
| FALSE       |  | TRUE    |  |
| TRUE        |  | FALSE   |  |
| UNKNOWN     |  | UNKNOWN |  |

Man beachte, dass NULL != UNKNOWN ist:

- SAL+NULL ergibt NULL (siehe vorherige Beispiele) während
- SAL<NULL den Wert UNKNOWN ergibt.

Logisches UND:

|                         |  |         |  |       |  |         |  |
|-------------------------|--|---------|--|-------|--|---------|--|
| x\y                     |  | TRUE    |  | FALSE |  | UNKNOWN |  |
| -----+-----+-----+----- |  |         |  |       |  |         |  |
| TRUE                    |  | TRUE    |  | FALSE |  | UNKNOWN |  |
| FALSE                   |  | FALSE   |  | FALSE |  | FALSE   |  |
| UNKNOWN                 |  | UNKNOWN |  | FALSE |  | UNKNOWN |  |

Logisches ODER:

|                         |  |      |  |         |  |         |  |
|-------------------------|--|------|--|---------|--|---------|--|
| x\y                     |  | TRUE |  | FALSE   |  | UNKNOWN |  |
| -----+-----+-----+----- |  |      |  |         |  |         |  |
| TRUE                    |  | TRUE |  | TRUE    |  | TRUE    |  |
| FALSE                   |  | TRUE |  | FALSE   |  | UNKNOWN |  |
| UNKNOWN                 |  | TRUE |  | UNKNOWN |  | UNKNOWN |  |

### 8.3.3 Dreiwertige Logik und die WHERE-Restriktion

Jeder Vergleich mit NULL führt zum Wahrheitswert UNKNOWN, und somit ergibt der folgende SELECT ein auf den ersten Blick seltsames Ergebnis: SQL:

```
SELECT ENAME, COMM
FROM EMP
WHERE COMM = COMM
```

Resultat:

|                     |      |
|---------------------|------|
| ENAME               | COMM |
| -----+-----         |      |
| Adams               | 1700 |
| Millsap             | 1400 |
| (2 row(s) affected) |      |

Grund ist, dass die WHERE-Bedingung die Zeilen herauspicks, bei denen TRUE herauskommt, die mit FALSE, aber auch die mit UNKNOWN, werden herausgefiltert! Auch der folgende SELECT führt zu einem "seltsamen" Ergebnis:

Frage/Anforderung: Alle Angestellten, die weniger als 1500 EUR Kommission (Spalte COMM) erhalten.

SQL:

```
SELECT ENAME, COMM
FROM EMP
WHERE COMM < 1500
```

Resultat:

| ENAME   | COMM |
|---------|------|
| Millsap | 1400 |

(1 row(s) affected)

Ist von der Anwendung her sicher bekannt, dass NULL für "der Angestellte bekommt 0,00 EUR Provision" steht, kann man mit dem Prädikat IS NULL in der WHERE-Restriktion arbeiten:

Frage/Anforderung: Alle Angestellten, die weniger als 1500 EUR Kommission (Spalte COMM) erhalten.

SQL:

```
SELECT ENAME, COMM
FROM EMP
WHERE COMM < 1500
OR (COMM IS NULL)
```

Resultat:

| ENAME    | COMM |
|----------|------|
| Norgaard | NULL |
| Dehaan   | NULL |
| Millsap  | 1400 |
| Mcdonald | NULL |
| ...      |      |

(9 row(s) affected)

**Warnung:** Es ist eine anwendungsabhängige Interpretationssache, ob NULL für UNBEKANNT oder für 0,00 EUR steht. Einfach in jeder WHERE-Restriktion noch die NULL-Werte mit IS NULL mitzunehmen, kann auch falsch sein.

Die Entscheidung, ob IS NULL angebracht ist oder nicht, ist eine Geschäftsentscheidung, keine technische Entscheidung!

### 8.3.4 Dreiwertige Logik und CHECK-Constraints

- WHERE-Restriktionen filtern auch Zeilen heraus, bei denen die Booleschen Verknüpfungen UNKNOWN ergeben.
- Demgegenüber lassen CHECK-Constraints Zeilen mit dem Ergebnis UNKNOWN durch!
  - Beispiel:

```
CHECK (DEPTNO IN(10, 20, 30))
```
- Das liegt daran, dass CHECK-Constraints nur die Zeilen nicht akzeptieren, deren CHECK-Bedingung FALSE ergibt.

## 8.4 Null-Werte und JOINS

### 8.4.1 OUTER-Joins können NULLEN erzeugen

Frage/Anforderung: Welcher Mitarbeiter sitzt in welcher Abteilung?

SQL:

```
SELECT E.ENAME,
 E.DEPTNO, D.DNAME
 FROM EMP E
RIGHT OUTER JOIN DEPT D
 ON E.DEPTNO = D.DEPTNO
```

Resultat:

| ENAME               | DEPTNO | DNAME         |
|---------------------|--------|---------------|
| Norgaard            | 10     | HQ            |
| Kolk                | 10     | HQ            |
| ...                 |        |               |
| Mcdonald            | 20     | Sales         |
| NULL                | NULL   | Manufacturing |
| (9 row(s) affected) |        |               |

Hier werden wegen dem Schlüsselwort OUTER automatisch NULLEN an den Stellen eingefügt, die keinen Partner haben.

Interessant ist auch, dass in der letzten Zeile DEPTNO NULL ist, während DNAME nicht NULL ist.

Frage/Anforderung: Welcher Mitarbeiter hat welchen Boss und wer ist dessen Boss?

SQL:

???????????

Resultat:

| MA       | Boss     | BossBoss |
|----------|----------|----------|
| Adams    | Norgaard | NULL     |
| Dehaan   | Adams    | Norgaard |
| Gennick  | NULL     | NULL     |
| Kolk     | Norgaard | NULL     |
| Lewis    | Kolk     | Norgaard |
| Mcdonald | Adams    | Norgaard |
| Millsap  | Adams    | Norgaard |
| Morle    | Norgaard | NULL     |
| Norgaard | NULL     | NULL     |
| Wood     | Kolk     | Norgaard |

## 8.5 Null-Werte und Aggregatfunktionen

- Skalare Ausdrücke ergeben den Wert NULL, wenn irgendeine NULL am Ausdruck beteiligt ist.
- Demgegenüber werden NULL-Werte bei Aggregatfunktionen ignoriert.

### 8.5.1 $\text{SUM}(A) + \text{SUM}(B) \neq \text{SUM}(A+B)$

SQL:

```
SELECT SUM(SAL+COMM)
 AS 'SUM(SAL+COMM)',
 SUM(SAL)+SUM(COMM)
 AS 'SUM(SAL)+SUM(COMM)'
FROM EMP
```

Resultat:

```
SUM(SAL+COMM) SUM(SAL)+SUM(COMM)
----- -----
2650 19900
(1 row(s) affected)
```

- Hier schlägt die Tatsache zu, dass bei Ausdrücken  $\text{NULL} + X = \text{NULL}$  ist, während bei  $\text{SUM}(X_1\dots X_n)$  die NULL-Werte lediglich ignoriert werden.
- Man muss daher, wenn immer man Ausdrücke oder Aggregatfunktionen ansetzt, sich Gedanken machen, was passiert, wenn ein oder mehrere Werte des selektierten Attributs NULL sind.
- Dazu sollte man vor der eigentlichen Berechnung mit  
`SELECT ... WHERE ... IS NULL` prüfen, ob NULL-Werte mitspielen.

- Auch in diesem Beispiel kann man nicht eindeutig sagen, welcher Ansatz der richtige ist. Das hängt wieder von der Bedeutung der NULL ab. Die kann sogar von Spalte zu Spalte verschieden sein!

### 8.5.2 Was ist die Summe von 0 Zeilen?

Fast schon philosophisch ist folgender SELECT: SQL:

```
SELECT COUNT(EMPNO),
 AVG(EMPNO),
 SUM(EMPNO),
 MAX(EMPNO),
 MIN(EMPNO)
 FROM EMP
 WHERE 1 = 2
```

Resultat:

```

0 NULL NULL NULL NULL
(1 row(s) affected)
```

- Dass der COUNT von 0 Zeilen 0 ergibt, ist klar.
- Dass es bei 0 Zeilen nur ein UNBEKANNTes Maximum und Minimum gibt, ist klar.
- Aber dass die Summe von 0 Zeilen UNBEKANNT und nicht 0 ist, ist zwar nicht klar, steht aber so im SQL-Standard.

## 8.6 Null-Werte und Unterabfragen

- Auch bei Unterabfragen sind NULL-Werte tückisch, wenn z.B. die Unterabfrage auch einen NULL-Wert erzeugt.
- Dann kann ein NOT IN leicht zum Ergebnis UNBEKANNT führen, was wiederum dazu führt, dass gar keine Zeile der Hauptabfrage die WHERE-Restriktion passieren kann, wie im folgenden Beispiel gezeigt.

### 8.6.1 Null-Werte und NOT IN

Frage/Anforderung: Alle Angestellten, die keine Untergebenen haben.

SQL:

```
SELECT E1.ENAME
 FROM EMP E1
 WHERE E1.EMPNO
 NOT IN
 (SELECT E2.MGR
 FROM EMP E2);
```

Resultat:

```
ENAME

(0 row(s) affected)
```

- Warum ist diese Antwort des DBMS richtig?

Frage/Anforderung: Alle Angestellten, die keine Untergebenen haben

SQL:

```
SELECT E1.ENAME
FROM EMP E1
WHERE E1.EMPNO
 NOT IN
 (SELECT E2.MGR
 FROM EMP E2
 WHERE E2.MGR
 IS NOT NULL);
```

Resultat:

```
ENAME

Dehaan
Millsap
McDonald
Wood
Lewis
Morle
Gennick

(7 row(s) affected)
```

## 8.6.2 Unterabfrage mit EXISTS

Frage/Anforderung: Alle Angestellten, die keine Untergebenen haben

SQL:

```
SELECT E1.ENAME
FROM EMP E1
WHERE NOT EXISTS
 (SELECT *
 FROM EMP E2
 WHERE
 E2.MGR = E1.EMPNO)
;
```

Resultat:

```
ENAME

Dehaan
```

```
Millsap
McDonald
Wood
Lewis
Morle
Gennick

(7 row(s) affected)
```

Hier "funktioniert" die Unterabfrage, da E2.MGR = E1.EMPNO für die E2.MGR-NULL-Werte jeweils UNKNOWN ergeben, EXISTS aber nach TRUE sucht.

### 8.6.3 Unterabfragen, die die leere Menge liefern

Frage/Anforderung: Alle Angestellten, die mehr als irgendein Verkäufer (engl. Salesrep) in Abteilung 10 verdienen.

SQL:

```
SELECT E1.ENAME
FROM EMP E1
WHERE E1.SAL >
(SELECT MAX(E2.SAL)
 FROM EMP E2
 WHERE E2.DEPTNO = 10
 AND E2.JOB='SALESREP')
;
```

Resultat:

```
ENAME

(0 row(s) affected)
```

Frage/Anforderung: Alle Angestellten, die mehr als irgendein Verkäufer (engl. Salesrep) in Abteilung 10 verdienen.

SQL:

```
SELECT E1.ENAME
FROM EMP E1
WHERE E1.SAL > ALL
(SELECT E2.SAL
 FROM EMP E2
 WHERE E2.DEPTNO = 10
 AND E2.JOB = 'SALESREP')
;
```

Resultat:

```
ENAME

Norgaard
```

```
Adams
Dehaan
Millsap
...
Morle
Gennick
(10 row(s) affected)
```

Warum dieser Unterschied?

# 9 Übungen

## 9.1 Allgemeines

Ziele, Inhalt, Umfang:

Siehe [Lehrveranstaltung Datenbanken 1 Labor \(IB 332\) im Modulkatalog.](#)

### 9.1.1 Organisation

Das Praktikum ist in 4 Aufgaben und 5 Termine aufgeteilt. Für die ersten 2 Aufgaben ist jeweils ein Termin vorgesehen. Diese bestehen aus "einfachen" SQL Anweisungen. Nacheinander werden einfache bis komplexere Abfragen und Joins erstellt.

Für die 3. Aufgabe sind 2 Termine vorgesehen. Hier geht es darum, den Umgang von Datenbanken in der Programmierung, am Beispiel von JDBC, zu lernen. Die Aufgabe ist wiederum in Unteraufgaben aufgeteilt und zeigt, wie man von Grund auf eine Datenbank zum Persistieren von Daten in ein Programm einbaut.

Für die 4. Aufgabe, ER-Modellierung geübt wird, ist der letzte Termin vorgesehen.

Weitere Informationen entnehmen Sie bitte aus dem ILIAS:

<https://ilias.hs-karlsruhe.de>

Dort finden Sie die Datei `SQL-Skripte.zip`, die SQL-Skripte zur Installation der Datenbank für Oracle, MS-SQL, Postgres und MySQL umfasst.

Weiter hinten in diesem Skript finden Sie eine genauere Beschreibung der Datenbank ("Anhang: Datenbankbeschreibung").

Ihre Lösungen zu den Aufgaben hinterlegen Sie bitte in Ihrem Gruppenordner im entsprechenden Verzeichnis im ILIAS.

Viel Erfolg!

### 9.1.2 Arbeiten mit der Hochschuldatenbank

Für den Zugriff auf die Datenbank steht die Anwendung **SQuirreL** im Poolraum zur Verfügung:

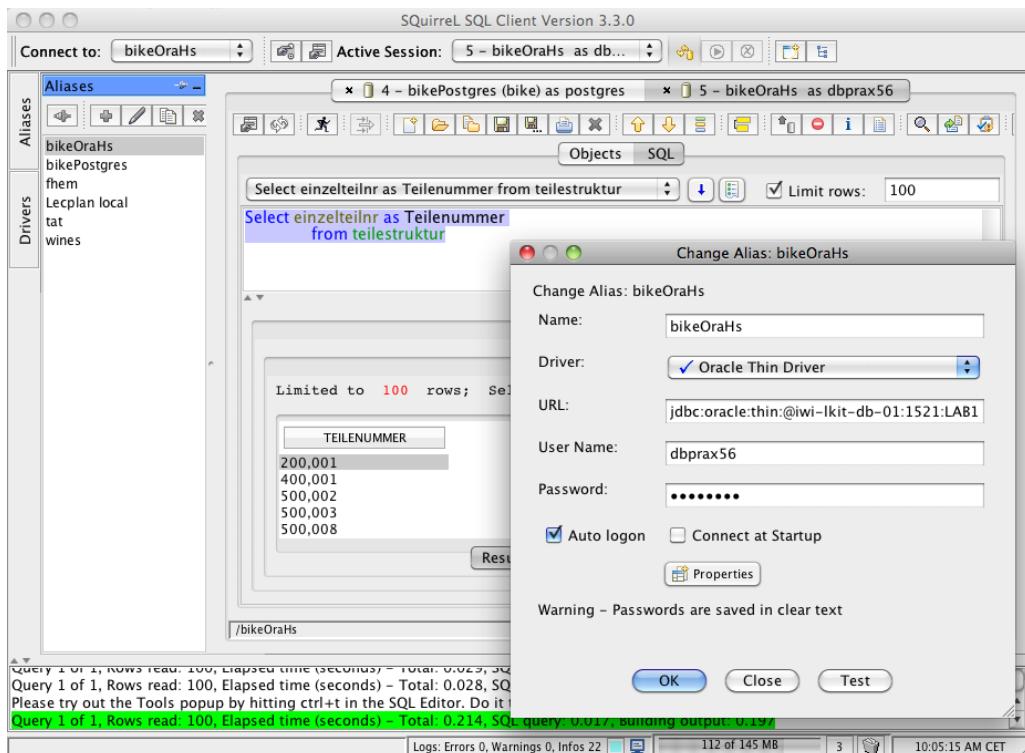


Bild 9.1: SQuirreL Client

- SQuirreL SQL Client ist im Linux-Menü auf den Laborrechnern zu finden
- Im Fenster **Aliases LAB1** mit Doppelklick auswählen. Es öffnet sich das Fenster **Connect to: LAB1**
- Unter URL steht `jdbc:oracle:thin:@iwi-lkit-db-01:1521:LAB1`
- Die Benutzerkennung und das Passwort für den Zugriff auf die Datenbank lauten gleich: `dbprax <zweistellige Gruppennummer>`

Das Benutzerpasswort für den Datenbankzugriff kann (muss aber nicht) nur vom Benutzer selbst mit folgendem SQL-Befehl geändert werden:

```
ALTER USER user IDENTIFIED BY myNewPassword;
```

### 9.1.3 Arbeiten mit der Hochschuldatenbank von extern

Es besteht auch die Möglichkeit, von zu Hause auf die Datenbank zuzugreifen.

Trotzdem entbindet Sie das nicht, zum Datenbanklabor zu kommen, da Anwesenheitspflicht besteht.

Aber so können Sie die verschiedenen SQL-Befehle üben und das Labor vorbereiten oder die Datenbank zur Klausurvorbereitung zu nutzen.

#### 9.1.3.1 Verwendung von VPN

1. Wenn Sie es nicht schon haben, dann benötigen Sie ein VPN-Client, um sich mit der Hochschule Karlsruhe zu verbinden.
2. Laden Sie den [SQuirreL Client](#) herunter und installieren Sie diesen.
3. Speichern Sie den Oracle Thin Driver (`oracle-ojdbc6.jar`) in dem installierten SQuirreL SQL Client im Ordner `lib` ab. Diese Jar-Datei, aber auch weitere für

andere Datenbanken, finden Sie in der Zip-Datei DBLab - JDBC-Projekt.zip unter ILIAS.

4. SQuirreL SQL Client starten.
5. Im Fenster Aliases das "+" anklicken.
6. Im Fenster Add Alias den Button New auswählen.
7. Im Fenster Add Driver in die Ansicht Extra Class Path wechseln und Add anklicken.
8. Jetzt in den Pfad gehen, wo der Oracle Thin Driver (oracle-ojdbc6.jar) abgelegt wurde. Auswählen und öffnen.
9. Das Fenster Add Driver schließen.
10. Im Fenster Add Alias bei Driver den Oracle Thin Driver auswählen.
11. Bei Name LAB1 eintragen.
12. Bei URL muss jdbc:oracle:thin:@iwi-lkit-db-01:1521:LAB1 eingetragen werden.
13. Bei User Name dbprax+Gruppennummer (z.B. dbprax04) hinzufügen. Das Passwort ist das gleiche wie der Username.
14. Mit OK das Fenster abschließen.

### 9.1.3.2 Verwendung von SSH in einer Shell

1. Unter Linux und Mac OS ist ein SSH-Client schon installiert.
2. Unter Windows können Sie entweder ssh.exe mit *Cygwin* <https://cygwin.com/> benutzen oder eine graphische Oberfläche namens Putty einsetzen (siehe nächstes Unterkapitel).
3. Am Hauptserver der Hochschule, der login.hs-karlsruhe.de heißt und zum Oracle-Server tunnelt, melden Sie sich wie im folgenden Bild gezeigt an.



```
brul0001-UbsSurface :/cygdrive/c/Users/Ulrich/Downloads/fmz2late/HTML-585|brul0001@UbsSurface|16:16|Downloads|ssh -L 1521:iwi-lkit-db-01:1521 -L 1 brul0001@login.hs-karlsruhe.deLast unsuccessful login: wed Feb 12 23:53:33 2014 on ssh from hsi-kbw-46-223-235-101.hsi.kabel-badenwuerttemberg.deLast login: Fri Sep 16 16:15:00 2016 on /dev/pts/9 from iwi-ibrul0001c.ads.hs-karlsruhe.de*****
* HOCHSCHULE KARLSRUHE *
* TECHNIK UND WIRTSCHAFT *
* (Informationsszentrum) *
* IZ - Benutzerberatung *
* Montag bis Freitag 8:00-14:00 Uhr Telefon: 0721 / 925 - 2305 *
* Aktuelle Ankündigungen und Betriebshinweise/Wartungstermine *
* siehe unter http://www.iz.hs-karlsruhe.de/ *
* Welcome to AIX on IBM System p *
*****%
```

Bild 9.2: SSH unter Cygwin/Windows.

Roter Rahmen in erster Zeile: Tunnel, grüner Rahmen: IZ-Anmeldename. Im Gegensatz zum Bild müssen Sie noch Ihr IZ-Passwort angeben. Wenn Sie das automatisieren wollen, siehe bitte: [SSH with Keys in a console window](#).

Hier zum Kopieren noch mal das Kommando:

```
ssh -L 1521:iwi-lkit-db-01:1521 -l <IZ-Name> login.hs-karlsruhe.de
```

4. Die JDBC-URL für Squirrel sieht dann so aus, da sie eben einen Tunnel auf localhost, Port 1521 eingerichtet haben:

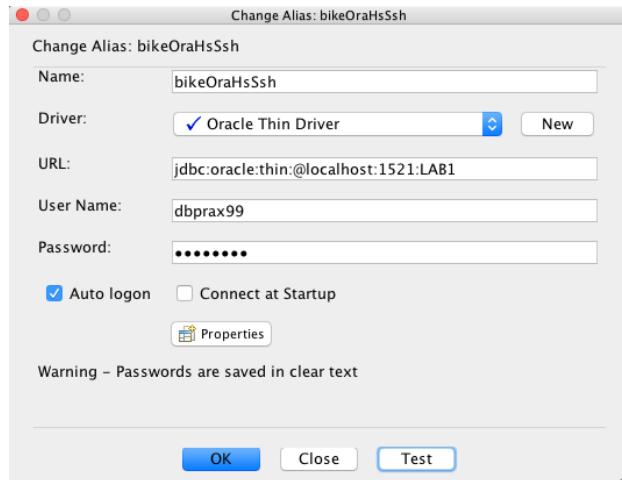


Bild 9.3: Einsatz des SSH-Tunnels unter Squirrel.

#### 9.1.3.3 Verwendung von SSH mit Putty unter Windows

1. Dazu brauchen müssen Sie den Client von <http://www.putty.org/> herunterladen und installieren.
2. Die nötigen Einstellungen entnehmen Sie bitte den folgenden Abbildungen:

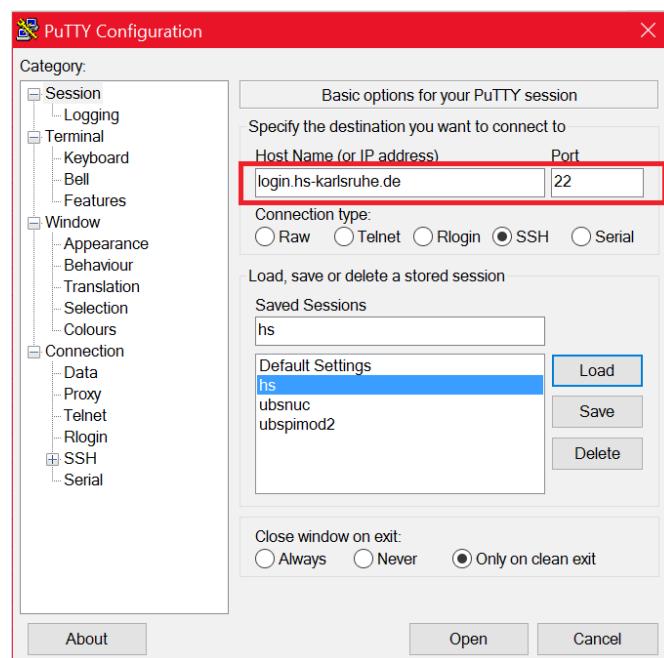


Bild 9.4: Tunnel-Server der HS eintragen.

^

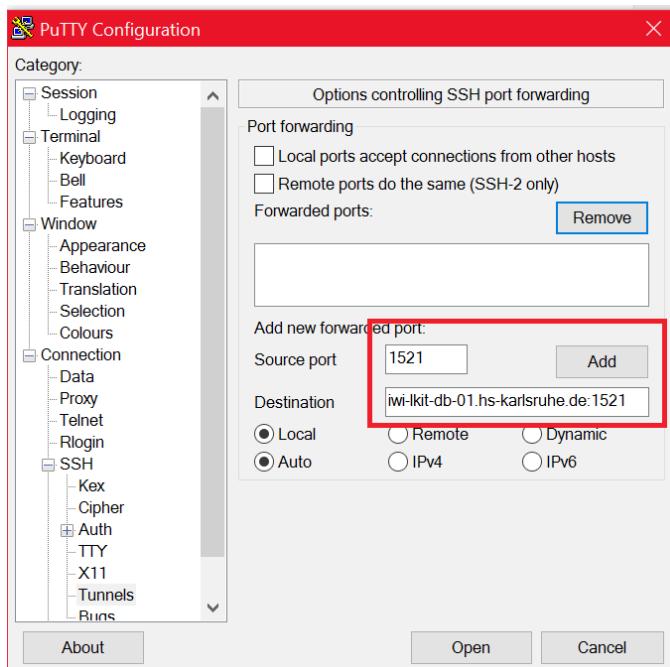


Bild 9.5: Tunnel-Daten eintragen.

Danach öffnen Sie mit Putty die eben eingerichtete Sitzung (Session auf `login.hs-karlsruhe.de`), der Tunnel zu `iwi-lkit-db-01:1521` wird dann automatisch aufgebaut und kann wie im Squirrel-Bild oben verwendet werden.

#### 9.1.4 Arbeiten mit anderen Datenbanken

Wie schon erwähnt, finden Sie im ILIAS in der Datei **SQL-Skripte.zip** die Skripte zur Installation der Datenbank für Oracle, MS-SQL, Postgres und MySQL. Damit können Sie auch zu Hause auf Ihren Privatrechnern üben.

Dies empfiehlt sich auf jeden Fall. Es empfiehlt sich sogar, mehr als eine Variante zu wählen, zum Beispiel eine freie Datenbank (Postgres) und eine industriell weit verbreitete Datenbank (MS-SQL) zu installieren und mit diesen zu üben, um die unterschiedlichen Merkmale dieser Produkte und die jeweiligen Inkompatibilitäten zum Beispiel kennen zu lernen.

#### 9.1.5 Datenbankschema installieren

Die entsprechenden SQL-Kommandos zum Erstellen der Tabellen für die Bike Datenbank befinden sich in der Datei `hska_<Datenbanktyp>_bike.sql` (die sich in der Zip-Datei **SQL-Skripte.zip** unter ILIAS befinden).

Im Labor wird also für die Hochschuldatenbank - die unter Oracle betrieben wird - das Installationsskript `hska_oracle_bike.sql` benötigt.

Sie können das Installationsskript ausführen, indem Sie den Inhalt der Datei in einen SQL Editor (z.B. SQuirreL SQL) laden und ablaufen lassen. Beim SQuirreL SQL Client muss der

Tab "SQL" ausgewählt sein, um die Datei zu öffnen. Die Befehle können (wenn sie markiert sind) mit der Tastenkombination `ctrl + return` ausgeführt werden.

Falls Sie zu Hause zum Beispiel eine Postgres Datenbank installieren und diese mit `hska_pgsql_bike.sql` "betanken", lesen Sie bitte unbedingt vorher etwaige Kommentare in den Köpfen der jeweiligen Installationsskripte.

Es gibt noch ein Erweiterungsskript, z.B. `hska_pgsql_bike2.sql`, das der Datenbank noch etliche weitere Daten (Zeilen) hinzufügt. Am Anfang des Labors wird dieses nicht gebraucht.

## 9.2 SQL-Aufgaben

### 9.2.1 Aufgabe 1 - Abfragen

Lesen Sie bitte obiges Kapitel "Organisation" durch. Arbeiten Sie die Datenbankbeschreibung im Anhang unten komplett durch.

- A.1.1 Installieren Sie die Beispieldatenbank. Schauen Sie sich die Relationen Kunde, Personal und Auftrag an. Notieren Sie den Namen des Kunden sowie den Namen des zuständigen Mitarbeiters zu Auftrag 2, welchen Namen hat der Vorgesetzte dieses Mitarbeiters? (Diese Aufgabe ist "von Hand" zu lösen, keine komplizierten JOINs etc. nötig.)
- A.1.2 Geben Sie alle Teile aus, die sich mindestens einmal im Lager befinden, aufsteigend sortiert nach der Anzahl der Teile.
- A.1.3 Finden Sie die Nummer aller Teile, die geliefert wurden, absteigend sortiert, keine Mehrfachnennung.
- A.1.4 Zeigen Sie die Teilenummer, Bezeichnung und (Brutto-)Preis für alle Teile, die mehr als 30 Euro kosten. Geben Sie jeder Spalte der Ausgabe eine selbsterklärende Überschrift.
- A.1.5 Welche Einzelteile des Teils 300001 werden mehr als 100 mal benötigt?  
Beispielausgabe:

|             |
|-------------|
| TEILENUMMER |
| -----       |
| 500001      |
| 500011      |

- A.1.6 Zeigen Sie den (Brutto-)Preis in Dollar und die Bezeichnung für alle blauen Teile, geben Sie der Spalte eine selbsterklärende Überschrift. Beispielausgabe:

| BEZEICHNUNG                 | PREIS_IN_DOLLAR |
|-----------------------------|-----------------|
| -----                       | -----           |
| Herren-City-Rad             | 1006,327        |
| Herren-City-Rahmen lackiert | 575,044         |

- A.1.7 Geben Sie (mit Hilfe von JOIN) die Bezeichnung aller von Kunden beauftragten Teile an, die auf Lager sind und aus einem oder mehreren Einzelteilen zusammengesetzt sind. (Es sind also nur *Oberteile* gefragt.) Keine Mehrfachnennung bitte. Beispielausgabe:

| BEZEICHNUNG                |
|----------------------------|
| Damen-City-Rad             |
| Damen-City-Rahmen lackiert |
| Herren-City-Rad            |

## 9.2.2 Aufgabe 2 - Joins, Views

- A.2.1 Erstellen Sie eine View mit dem Namen v500009, welche die Bezeichnung und den Lieferanten mit dem niedrigsten Nettopreis für das Teil 500009 ausgibt. Rufen Sie die View auf. Beispielausgabe:

| BEZEICHNUNG   | LIEFERANT              |
|---------------|------------------------|
| Sattelstuetze | Firma Gerti Schmidtner |

- A.2.2 Zum Zweck der Rückverfolgung möchte die Geschäftsführung eine View zu allen unvollständigen Auftragsposten haben. Auftragsposten sind unvollständig, wenn deren bestellte Teile nicht in ausreichender Anzahl auf Lager sind. Die View hat den Namen fehlendT und soll zu allen unvollständigen Auftragsposten die Bezeichnung zu dem Teil und wie viele Teile jeweils fehlen, um alle Auftragsposten zu dem Teil zu vervollständigen, anzeigen. Reservierte Teile werden hier nicht beachtet.

Hinweis: Eine zweite View, auf die Ihre eigentliche View zugreift, könnte sich als nützlich erweisen. Beispielausgabe:

| BEZEICHNUNG                 | FEHLEND |
|-----------------------------|---------|
| Herren-City-Rad             | 1       |
| Herren-City-Rahmen lackiert | 1       |

- A.2.3 Finden Sie die Namen aller Mitarbeiter, welche Vorgesetzte mindestens eines anderen Mitarbeiters sind. Lösen Sie diese Aufgabe bitte vier mal, jeweils mit:

1. EXISTS
2. JOIN
3. IN
4. ANY (Theta-Abfrage)

Beispielausgabe:

VORGESETZTER

Maria Forster  
Marianne Lambert

#### A.2.4 Rabattaktion!

Der Kunde, dessen Aufträge in dem Zeitraum vom 01.10.08 bis zum 01.11.08 den höchsten Umsatz gebracht haben, erhält 10% Rabatt auf seine Aufträge in dieser Zeit. Finden Sie diesen Kunden! Wer ist der zuständige Kundenbetreuer und wer ist dessen Chef(in)? Was kostet das? Beispielausgabe:

| KdNr | KdName          | Kundenbetreuer | Kundenbetreuerchef | Preis | Rabatt |
|------|-----------------|----------------|--------------------|-------|--------|
| 4    | Rafa - Seger KG | Anna Kraus     | Maria Forster      | 3.286 | 328,6  |

## 9.3 Java-Aufgaben

### 9.3.1 Aufgabe 3 - JDBC

In dieser Aufgabe des Datenbankpraktikums geht es um Teile einer Implementierung eines Backends für die Warenwirtschaft des Fahrradhändlers. Das mitgelieferte Java Projekt beinhaltet nur die Klassen aus der SQL-Schicht der Anwendung.

Im ILIAS finden Sie das Paket für das Javaprojekt (**DBLab-JDBC-Projekt.zip**). In dem Paket des Javaprojekts sind neben allen benötigten Klassen auch die jeweiligen Datenbanktreiber enthalten, diese müssen Sie aber noch in Ihr Projekt einbinden, um sie benutzen zu können. Tipps, wie Sie das Projekt installieren und die Treiber einbinden können, finden Sie im Abschnitt "Tipps" der im Anhang folgenden "Datenbankbeschreibung".

Die Aufgaben selbst befinden sich innerhalb der Klassen. Diese sind nicht fertig implementiert, in den Javadoc Kommentaren ist beschrieben, welche Funktionalität innerhalb der

```
//TODO
...
//END TODO
```

Blöcke ergänzt werden soll. Weiter gibt es innerhalb der Javadoc Kommentare noch Fragen, die bei der Abgabe der Aufgabe beantwortet sein müssen.

Die Klassen sollten bitte in dieser Reihenfolge bearbeitet werden:

| Aufg.-Nr. | Klassenname               | Name d. Termins im Terminplan |
|-----------|---------------------------|-------------------------------|
| 3.a       | SQLConnector              | A3.1 Java                     |
| 3.b       | Output                    | A3.1 Java                     |
| 3.c       | CustomerSupplierRelations | A3.1 Java                     |
| 3.d       | SQLUpdateManager          | A3.2 Java                     |

Diese Klassen befinden sich in Ihrem Projekt im default-Package im Verzeichnis ./src.

Weiter finden Sie dort die Klasse LabUtilities, mit der Sie die Datenbank einfach neu aufsetzen können beim Testen. Siehe die dortige main() -Methode.

Auch jede andere Klasse verfügt über eine main() -Methode, die dem Testen - und der Abgabe der Teilaufgabe - dient.

## 9.4 Datenbankmodellierung

### 9.4.1 Aufgabe 4.1 - ERM



Bild 9.7: Rennenten

Sie erhalten von einem Kunden den Auftrag ein Informationssystem zu erstellen. Leiten Sie aus der folgenden Beschreibung des Kunden ein Datenmodell ab, das die verwendeten Geschäftsregeln wiedergibt:

- Bei Richmond Arms findet jährlich ein großes Ereignis statt. Richmond Arms ist der Name einer Kneipe direkt neben dem Mühlengraben in West Ashling, ein Dorf in der Grafschaft Sussex. Der Kneipenwirt veranstaltet jedes Jahr ein Entenrennen zugunsten wohltätiger Hilfsorganisationen.
- Schon beim ersten Rennen wurde dem Wirt bewusst, dass lebende Enten für ein organisiertes Rennen nicht ruhig genug sind. Daher kam er auf die Idee, Enten aus

Holz zu benutzen. Die Enten werden in den Mühlengraben gesetzt und schwimmen mit der Strömung vom Start bis zur Southbrook-Brücke.

- Gäste der Kneipe, die mit einer Ente am Rennen teilnehmen wollen, der Wirt nennt sie Sponsoren, müssen einen Geldbetrag von mindestens 100 Pfund zu Gunsten einer Hilfsorganisation seiner Wahl setzen, bevor sie eine Ente bekommen. Die Ente muss in ordentlicher aufrechter Position schwimmen. Veränderungen zum besseren Schwimmverhalten sind erwünscht, mit Ausnahme jeglicher Form von Antrieb.
- Die Sponsoren schmücken mit großem Aufwand ihre Enten. Dabei muss die offizielle Nummer an der vorgeschriebenen Stelle gut sichtbar bleiben. Der Wirt teilt die Enten in verschiedene Dekorationsgruppen ein. Die örtlichen Unternehmen spenden Preise für diese Gruppen. Diese Preise verleiht der Wirt an den Sponsor der originellsten Ente jeder Gruppe.
- Der Sponsor darf die Ente während des Rennens nicht anfassen. Der Wirt hat diese Regel wohlweislich eingeführt, nachdem mehrere übereifrige Sponsoren ins Wasser gesprungen waren, um ihre Enten zu unterstützen. Sie, die Sponsoren, wurden nicht ganz unversehrt stromabwärts in der Gegend von Oakwood Weir an Land gezogen. Die Enten werden immer noch vermisst.
- Jede Ente erhält eine Startnummer und eine Startposition. Die Startpositionen der Enten werden durch das Los bestimmt. Eine Seite der Absperrung wird beim Start geöffnet. Enten mit niedrigen Positionsnummern sind dabei im Vorteil.
- Ein Sponsor kann mehrere Enten auswählen, wenn er unsicher ist, welche Bauweise die Strömung am besten ausnutzt. Um seine Gewinnchancen zu erhöhen, wählt er mehrere Enten aus und verändert jeweils deren Schwimmverhalten im Rahmen der zulässigen Regeln.
- Der Wirt von Richmond Arms will nicht mehr Personen als nötig nachrennen, um deren Einsätze einzusammeln. Er hat auch keine Lust, mit mehreren Sponsoren je Ente zu verhandeln. Deswegen können nicht mehrere Sponsoren gemeinsam auf dieselbe Ente setzen.
- Der Wirt notiert Datum, Zeit und Namen eines Zeugen, wenn einer seiner Gäste eine Ente sponsieren möchte. Er tut dies, damit er die Sponsoren an ihre Verpflichtung erinnern kann, falls sie vergessen sollten, ihre Ente am Tag des Rennens abzuholen.
- Die Summe der Einsätze beim Entenrennen wird der Hilfsorganisation übergeben, deren Ente das Rennen gewinnt. Der glückliche Sponsor der Gewinnerente bekommt offiziell nichts für den Sieg seiner Ente, aber der Wirt spendiert ihm das eine oder andere Freibier.

Aufgabe 4.1.1: Dokumentieren Sie Ihr Ergebnis durch ein ERM-Diagramm in Chen-Notation. Nur die zum Verständnis wirklich nötigen Attribute (z.B. Beziehungsattribute) müssen mit aufgenommen werden.

Aufgabe 4.1.2: Erstellen Sie dazu das Relationenschema, dort bitte alle Attribute aufführen.

Aufgabe 4.1.3: Erstellen Sie ein physikalisches Datenbankschema in SQL, einschließlich der Foreign-Key-Beziehungen.

Surrogatschlüssel verwenden Sie dabei bitte spärlich, falls doch, nennen Sie diese bitte "id", darauf zeigende Fremdschlüssel bitte "xxx\_id".

## 9.4.2 Aufgabe 4.2 - ERM

Wiederholen Sie Aufgabe 4.1, aber dieses mal für Ihre MMK-Entwurfsaufgabe!

Aufgabe 4.2.1: Dokumentieren Sie Ihr Ergebnis durch ein ERM-Diagramm in Chen-Notation.

Aufgabe 4.2.2: Erstellen Sie dazu das Relationenschema, dort bitte alle Attribute aufführen.

Aufgabe 4.2.3: Erstellen Sie ein physikalisches Datenbankschema in SQL, einschließlich der Foreign-Key-Beziehungen.

Surrogatschlüssel verwenden Sie dabei bitte spärlich, falls doch, nennen Sie diese bitte "id", darauf zeigende Fremdschlüssel bitte "xxx\_id".

## 9.4.3 Aufgabe 4.3 - Normalisierung

Gegeben sei folgende Relation zur Auftragsverwaltung:

| MitarbeiterNr | Abteilung | ChefNr | Kunde                  | Ware                           | Anzahl     |
|---------------|-----------|--------|------------------------|--------------------------------|------------|
| 11            | VT        | 5      | Miele, Bosch,<br>Miele | Dichtung, TFT,<br>Drehschalter | 15, 51, 55 |
| 21            | VT        | 5      | DB, Miele              | Trafo, Dichtung                | 58, 55     |
| 28            | EN        | 8      | SEW                    | Wicklung                       | 70         |

MitarbeiterNr und ChefNr seien dabei Fremdschlüssel in eine Personaltabelle, die nicht weiter betrachtet werden muss.

### A.4.3.1

Erstellen Sie bitte ein Relationenschema für die Auftragsverwaltung in 1NF.

### A.4.3.2

Erstellen Sie daraus bitte ein Relationenschema für die Auftragsverwaltung in 3NF.

## 9.4.4 Aufgabe 4.4 - Normalisierung

Überführen Sie Ihr Relationenschema zur MMK-Entwurfsaufgabe aus Aufgabe 4.2.2 in die 3NF. Begründen Sie für jede Relation Ihres gegebenenfalls neuen Relationenschemas, dass 1NF ... 3NF erfüllt sind.

## 9.5 Anhang: Datenbankbeschreibung für die SQL- und JDBC-Aufgaben

### 9.5.1 Überblick

Die Aufgaben des Datenbanklabors basieren auf der Bike Datenbank eines Fahrradhändlers von E. Schicker [Schicker00]. Diese Datenbank ist der Praxis entnommen und voll funktionsfähig, mit sowohl einfachen als auch komplexen Relationen. Durch diesen Aufbau ist es möglich, in den Übungen einen Eindruck über die Leistungsfähigkeit von relationalen Datenbanken zu bekommen.

An der Hochschule steht ihnen für die Übungen der Zugang zu einer Oracle Datenbank zur Verfügung, das Übungspaket enthält aber auch SQL Skripte der Bike Datenbank für MS-SQL, MySQL sowie Postgres.

Entstanden ist die Bike Datenbank zur Verwaltung der Warenwirtschaft von Fahrradhändlern. Folgende Überlegungen liegen dabei zugrunde:

- Es existiert eine bestimmte, aber mit der Zeit variable Anzahl von Teilen welche Einzelteile, Zwischenteile sowie Endprodukte umfassen.
- Zwischenteile und Endprodukte bestehen aus einfacheren Teilen, die Datenbank soll diese Struktur wiedergeben.
- Der aktuelle Lagerbestand muss festgehalten werden. Für geplante Arbeiten benötigte Teile werden reserviert.
- Für die Bestellung neuer Teile werden Lieferanten benötigt. Es muss auch erkennbar sein, welcher Lieferant welches Teil liefern kann.
- Um dem Geschäft nachgehen zu können, müssen Aufträge verwaltet werden, welche wiederum von Kunden kommen. Es soll auch vermerkt werden, welche Arbeiten zu einem Auftrag anfallen und welcher Mitarbeiter für diese zuständig ist. Da hierfür Teile reserviert werden, besteht auch ein Zusammenhang zum Lager.
- Die Mitarbeiter werden damit auch in der Datenbank geführt, für Aufträge soll erkennbar sein, welcher Mitarbeiter welchen Auftrag entgegen nahm.
- Um die Datenbank einfach zu halten, wird auf ein Rechnungs- und Mahnwesen verzichtet.

## 9.5.2 Relationales Datenmodell im Bild

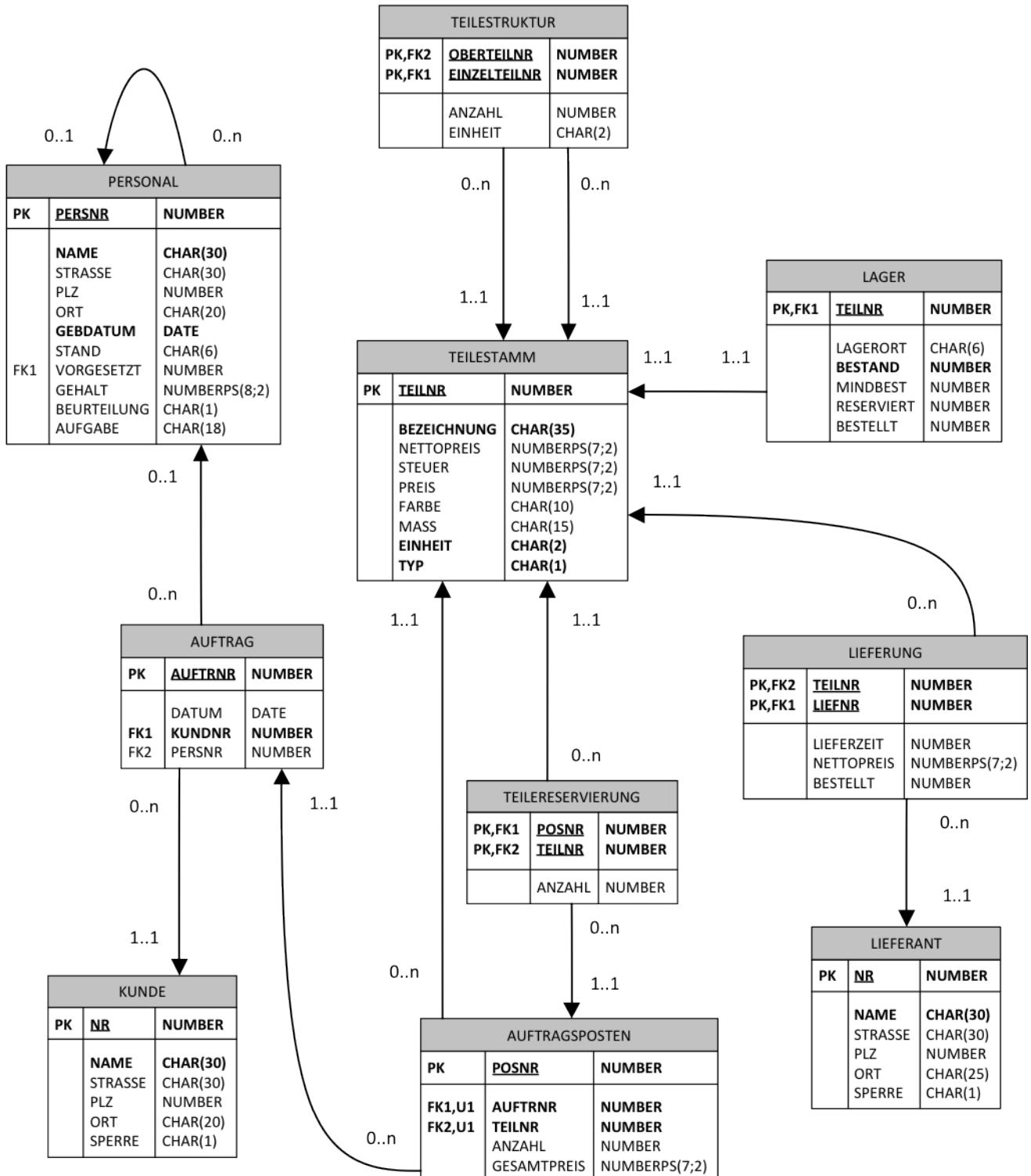


Bild 9.9

Aufgrund der Vorüberlegungen kann man schnell die Teileverwaltung und das Auftragswesen als zentrale Stellen erkennen. Dazu gibt es Lieferanten, Kunden sowie Personal.

Zur Verwaltung der Teile gibt es die Entität **Teilestamm**. Hier finden sich alle Teile, mit denen der Händler in Berührung kommt, vom Einzelteil bis zum fertigen Fahrrad. Dazu wird

eine Entität **Lager** benötigt, welche den aktuellen Bestand speichert. Diese ist schwach bezüglich Teilestamm, da Teile, die es nicht gibt, auch nicht gelagert werden können. Der Aufbau komplexer Teile wird über eine Beziehungsentität, die **Teilestruktur**, realisiert.

Solche Beziehungsrelationen, die Entitäten intern über zwei Verbindungen verknüpfen, nennt man Stücklisten. Diese kommen in der Praxis immer wieder vor. Für die Stückliste Teilestruktur, bilden zwei Fremdschlüssel den Primärschlüssel. Der erste Fremdschlüssel (Oberteilnr) zeigt auf das komplexe Teil, der zweite Fremdschlüssel (Einzelteilnr) auf ein Einzelteil des komplexen Teils. Wird ein Teil beispielsweise aus 10 Einzelteilen zusammengesetzt, gibt es in Teilestruktur 10 Einträge zu diesen Teilen. Jeder Eintrag steht für eines der Tupel aus komplexem Teil und Einzelteil, dazu gibt es ein Attribut für die benötigte Anzahl an Einzelteilen.

Für das Auftragswesen werden alle Aufträge in der Entität **Auftrag** hinterlegt. Diese hat Beziehungen zu **Kunden** als auch **Personal**. Einzelheiten zu den Aufträgen findet man in **Auftragsposten**. Diese Entität besitzt außerdem 2 Beziehungen zum **Teilestamm**, einmal um zu wissen, welche Teile in Auftrag gegeben wurden und zum anderen muss erkennbar sein, welche Teile reserviert werden müssen. Da es sich hier um eine m:n Beziehung handelt, wird eine eigene Beziehungsentität, die **Teilereservierung**, benötigt. Die 10 Entitäten aus dem Modell der Datenbank, davon 4 Beziehungsentitäten, werden eins zu eins in Relationen umgesetzt. Die Beziehungsentitäten lösen alle m:n Beziehungen auf, diese werden in relationalen Datenbanken als eigenständige Relation realisiert. Jede Relation besitzt einen Primärschlüssel und jeder Pfeil repräsentiert einen Fremdschlüssel.

Der Fremdschlüssel ist immer in der Relation mit der 0..n Beziehung, bei der einzigen 1:1 Beziehung ist der Fremdschlüssel in der schwachen Entität **Lager**.

Die Relationen sind schon mit den Beispieldaten aus dem SQL Skript gefüllt.

### 9.5.3 Relation: Teilestamm

Diese Relation enthält detaillierte Informationen über jedes Teil. Falls vorhanden, gibt Mass die Abmessung für ein Teil und Einheit die Maßeinheit (z.B. ST für Stück) an. Das Attribut Typ zeigt, ob es sich um ein Endprodukt (E), ein zusammengesetztes Teil (Z) oder ein Fremdteil (F) handelt. Fremdteile sind Einzelteil von Lieferanten.

| TEILNR | BEZEICHNUNG     | NETTOPREIS | STEUER | PREIS | FARBE | MASS    | EINHEIT | Typ |
|--------|-----------------|------------|--------|-------|-------|---------|---------|-----|
| 100001 | Herren-City-Rad | 588,24     | 111,76 | 700   | blau  | 26 Zoll | ST      | E   |
| 100002 | Damen-City-Rad  | 546,22     | 103,78 | 650   | rot   | 26 Zoll | ST      | E   |
| 200001 | Herren-City...  | 336,13     | 63,87  | 400   | blau  | NULL    | ST      | Z   |
| 200002 | Damen-City...   | 336,13     | 63,87  | 400   | rot   | NULL    | ST      | Z   |
| 300001 | Herren-City...  | 310,92     | 59,08  | 370   | NULL  | NULL    | ST      | Z   |
| 300002 | Damen-City...   | 310,92     | 59,08  | 370   | NULL  | NULL    | ST      | Z   |
| 400001 | Rad             | 58,82      | 11,18  | 70    | NULL  | 26 Zoll | ST      | Z   |
| 500001 | Rohr 25CrMo4... | 6,3        | 1,2    | 7,5   | NULL  | 9mm     | CM      | F   |
| 500002 | Sattel          | 42,02      | 7,98   | 50    | NULL  | NULL    | ST      | F   |
| 500003 | Gruppe Deore LX | 5,88       | 1,12   | 7     | NULL  | LX      | ST      | F   |
| 500004 | Gruppe Deore XT | 5,04       | 0,96   | 6     | NULL  | XT      | ST      | F   |
| 500005 | Gruppe XC-LTD   | 6,72       | 1,28   | 8     | NULL  | XC-LTD  | ST      | F   |
| 500006 | Felgensatz      | 33,61      | 6,39   | 40    | NULL  | 26 Zoll | ST      | F   |
| ...    |                 |            |        |       |       |         |         |     |

Bild 9.10

## 9.5.4 Relation: Teilestruktur

Wie bereits beim relationalen Modell beschrieben, enthält diese Relation die Information, welche zusammengesetzten Teile aus welchen Einzelteilen bestehen. Für jedes Tupel einer Beziehung von komplexem Teil zu einem Einzelteil existiert ein Eintrag mit der benötigten Anzahl für das jeweilige Einzelteil und der entsprechenden Einheit.

| OBERTEILNR | EINZELTEILNR | ANZAHL | EINHEIT |
|------------|--------------|--------|---------|
| 100001     | 200001       | 1      | ST      |
| 100001     | 500002       | 1      | ST      |
| 100001     | 500003       | 1      | ST      |
| 100001     | 400001       | 1      | ST      |
| 100001     | 500008       | 1      | ST      |
| 100001     | 500009       | 1      | ST      |
| 100001     | 500010       | 1      | ST      |
| 100002     | 200002       | 1      | ST      |
| 100002     | 500002       | 1      | ST      |
| ...        |              |        |         |

Bild 9.11

## 9.5.5 Relation: Lager

Neben dem aktuellen Bestand findet sich hier auch der Lagerort, der Mindestbestand, die Anzahl der reservierten als auch die Anzahl der bestellten Teile. Dazu muss ein Teil aus dem Teilstamm hier nicht aufgeführt sein, wenn es nicht auf Lager, reserviert oder bestellt ist.

| TEILNR | LAGERORT | BESTAND | MINDBEST | RESERVIERT | BESTELLT |
|--------|----------|---------|----------|------------|----------|
| 100001 | 001002   | 3       | 0        | 2          | 0        |
| 100002 | 001001   | 6       | 0        | 3          | 0        |
| 200001 |          | 0       | 0        | 0          | 0        |
| 200002 | 004004   | 2       | 0        | 0          | 0        |
| 300001 |          | 0       | 0        | 0          | 0        |
| 300002 | 002001   | 7       | 0        | 2          | 0        |
| 500001 | 003005   | 8050    | 6000     | 184        | 0        |
| 500002 | 002002   | 19      | 20       | 2          | 10       |
| 500003 | 001003   | 15      | 10       | 0          | 0        |
| 500004 | 004001   | 18      | 10       | 0          | 0        |
| 500005 | 003002   | 2       | 0        | 0          | 0        |
| ...    |          |         |          |            |          |

Bild 9.12

## 9.5.6 Relation: Lieferung

Die Relation Lieferung gibt an, welche Teile von welchem Lieferanten in welcher Zeit geliefert werden. Außerdem werden der Preis und die aktuellen Bestellungen vermerkt.

| TEILNR | LIEFNR | LIEFERZEIT | NETTOPREIS | BESTELLT |
|--------|--------|------------|------------|----------|
| 500001 | 5      | 1          | 5,5        | 0        |
| 500002 | 2      | 4          | 36,6       | 10       |
| 500002 | 1      | 5          | 35,1       | 0        |
| 500003 | 3      | 6          | 5,6        | 0        |
| 500003 | 4      | 5          | 5,45       | 0        |
| 500004 | 3      | 2          | 4,7        | 0        |
| 500004 | 4      | 3          | 4,5        | 0        |
| 500005 | 4      | 5          | 5,7        | 0        |
| 500006 | 1      | 1          | 26         | 0        |
| 500007 | 1      | 2          | 15,5       | 0        |
| 500008 | 1      | 4          | 73         | 0        |
| ...    |        |            |            |          |

Bild 9.13

## 9.5.7 Relation: Lieferant

Enthält alle wichtigen Attribute der Lieferanten. Sperre gibt an, ob z.B. aufgrund schlechter Erfahrungen hier keine Bestellungen mehr getätigt werden sollen. In der Praxis werden oft noch weitere Attribute für Statistiken oder weitere Informationen gespeichert.

| NR | NAME                   | STRASSE             | PLZ   | ORT        | SPERRE |
|----|------------------------|---------------------|-------|------------|--------|
| 1  | Firma Gerti Schmidtner | Dr. Gesslerstr. 59  | 93051 | Regensburg | 0      |
| 2  | Rauch GmbH             | Burgallee 23        | 90403 | Nürnberg   | 0      |
| 3  | Shimano GmbH           | Rosengasse 122      | 51143 | Köln       | 0      |
| 4  | Suntour LTD            | Meltonstreet 65     |       | London     | 0      |
| 5  | MSM GmbH               | St-Rotteneckstr. 13 | 93047 | Regensburg | 0      |

Bild 9.14

## 9.5.8 Relation: Auftrag

Da kein Rechnungswesen existiert, ist diese Relation sehr einfach gehalten. Zu jeden Auftrag wird hier nur das Datum, die Kundennummer und die Vertreternummer gespeichert. Einzelheiten dazu befinden sich in der Relation Auftragsposten.

| AUFRNR | DATUM      | KUNDNR | PERSNR |
|--------|------------|--------|--------|
| 1      | 04.08.2008 | 1      | 2      |
| 2      | 06.09.2008 | 3      | 5      |
| 3      | 07.10.2008 | 4      | 2      |
| 4      | 18.10.2008 | 6      | 5      |
| 5      | 03.11.2008 | 1      | 2      |

Bild 9.15

## 9.5.9 Relation: Auftragsposten

Diese Relation nimmt mit 2 Beziehungen zum Teilestamm und einer Beziehung zum Auftrag eine zentrale Rolle ein. Eine der Beziehungen zum Teilestamm wird über die Beziehungsrelation Teilereservierung realisiert. Über die direkte Beziehung wird das in Auftrag gegebene Teil gemerkt, über die Beziehungsrelation die dafür zu reservierenden Einzelteile.

| POSNR | AUFRNR | TEILNR | ANZAHL | GESAMTPREIS |
|-------|--------|--------|--------|-------------|
| 11    | 1      | 200002 | 2      | 800         |
| 21    | 2      | 100002 | 3      | 1950        |
| 22    | 2      | 200001 | 1      | 400         |
| 31    | 3      | 100001 | 1      | 700         |
| 32    | 3      | 500002 | 2      | 100         |
| 41    | 4      | 100001 | 1      | 700         |
| 42    | 4      | 500001 | 4      | 30          |
| 43    | 4      | 500008 | 1      | 94          |
| 51    | 5      | 500010 | 1      | 40          |
| 52    | 5      | 500013 | 1      | 30          |

Bild 9.16

## 9.5.10 Relation: Teilereservierung

Gibt Auskunft, welche Teile für welchen Auftragsposten reserviert wurden. Es handelt sich um eine Beziehungsrelation, die beiden Fremdschlüsselelemente bilden zusammen den Primärschlüssel, dazu gibt es noch die Anzahl der reservierten Teile.

| POSNR | TEILNR | ANZAHL |
|-------|--------|--------|
| 11    | 300002 | 2      |
| 21    | 100002 | 3      |
| 22    | 500001 | 180    |
| 22    | 500011 | 161    |
| 22    | 500012 | 20     |
| 22    | 500013 | 1      |
| 22    | 500014 | 1      |
| 31    | 100001 | 1      |
| 32    | 500002 | 2      |
| 41    | 100001 | 1      |
| 42    | 500001 | 4      |
| 43    | 500008 | 1      |
| 51    | 500010 | 1      |
| 52    | 500013 | 1      |

Bild 9.17

### 9.5.11 Relation: Kunde

In Kunde werden alle wichtigen Attribute für Kunden gespeichert. Das Attribut Sperre gibt an, ob von einem Kunden z.B. aufgrund mangelnder Liquidität kein Auftrag mehr angenommen werden darf.

| NR | NAME                   | STRASSE             | PLZ   | ORT        | SPERRE |
|----|------------------------|---------------------|-------|------------|--------|
| 1  | Fahrrad Shop           | Obere Regenstr. 4   | 93059 | Regensburg | 0      |
| 2  | Zweirad-Center Staller | Kirschweg 20        | 44267 | Dortmund   | 0      |
| 3  | Maier Ingrid           | Universitätsstr. 33 | 93055 | Regensburg | 1      |
| 4  | Rafa - Seger KG        | Liebigstr. 10       | 10247 | Berlin     | 0      |
| 5  | Biker Ecke             | Lessingstr. 37      | 22087 | Hamburg    | 0      |
| 6  | Fahrräder Hammerl      | Schindlerplatz 7    | 81739 | München    | 0      |

Bild 9.18

### 9.5.12 Relation: Personal

Die Relation Personal ist ähnlich der Relation Kunde aufgebaut. Neben den Standardattributen befindet sich hier aber noch das Geburtsdatum, der Familienstand, die/der Vorgesetzte (rekursiv!), das Gehalt, eine persönliche Beurteilung und die Aufgabe in der Firma. In der Praxis werden häufig noch viele weiter Daten über Mitarbeiter gespeichert.

| PERSNR | NAME             | STRASSE          | PLZ   | ORT        | GEBDATUM   |
|--------|------------------|------------------|-------|------------|------------|
| 1      | Maria Forster    | Ebertstr. 28     | 93051 | Regensburg | 05.07.1979 |
| 2      | Anna Kraus       | Kramgasse 5      | 93047 | Regensburg | 09.07.1975 |
| 4      | Heinz Rolle      | In der Au 5      | 90455 | Nürnberg   | 12.10.1957 |
| 5      | Johanna Köster   | Wachtelstr. 7    | 90427 | Nürnberg   | 07.02.1984 |
| 6      | Marianne Lambert | Fraunhofer Str 3 | 92224 | Landshut   | 22.05.1974 |
| 3      | Ursula Rank      | Dreieichstr. 12  | 60594 | Frankfurt  | 04.09.1967 |
| 7      | Thomas Noster    | Mahlergasse 10   | 93047 | Regensburg | 17.09.1972 |
| 8      | Renate Wolters   | Lessingstr. 9    | 86159 | Augsburg   | 14.07.1979 |
| 9      | Ernst Pach       | Olgastr. 99      | 70180 | Stuttgart  | 29.03.1992 |

Bild 9.19

| PERSNR | STAND | VORGESETZT | GEHALT | BEURTEILUNG | AUFGABE          |
|--------|-------|------------|--------|-------------|------------------|
| 1      | verh  |            | 4800   | 2           | Manager          |
| 2      | led   | 1          | 2300   | 3           | Vertreter        |
| 4      | led   | 1          | 3300   | 3           | Sekretär         |
| 5      | ges   | 1          | 2100   | 5           | Vertreter        |
| 6      | verh  |            | 4100   | 1           | Meister          |
| 3      | verh  | 6          | 2700   | 1           | Facharbeiterin   |
| 7      | verh  | 6          | 2500   | 5           | Arbeiter         |
| 8      | led   | 1          | 3300   | 4           | Sachbearbeiterin |
| 9      | led   | 6          | 800    |             | Azubi            |

Bild 9.20: Weitere Attribute an der Relation Personal

### 9.5.13 Tipps

1. Es kann vorkommen, dass in Eclipse die **Umlaute** in den Source Files für die JDBC Aufgaben nicht richtig angezeigt werden. Um dies zu beheben, muss unter Window → Preferences → General → Workspace unten im Fenster das **Text file encoding** von Default auf UTF-8 eingestellt werden.
2. Zum **Erstellen des Projekts** für die JDBC Aufgabe laden Sie sich bitte die Projektdateien (**DBLab - JDBC-Projekt.zip**) aus dem ILIAS herunter und entpacken Sie die Dateien. Erstellen Sie in Eclipse ein neues Java Projekt, dieses enthält für gewöhnlich schon einen Ordner mit der Bezeichnung **src**. Sollte der Ordner fehlen, können Sie ihn mit Rechtsklick auf das Projekt → New → Source Folder erstellen.

Um fortzufahren, ziehen Sie den **dblab** Ordner, dieser befindet sich in den vorher entpackten Dateien in dem Ordner **src**, per Drag & Drop in den **src** Ordner Ihres eben erstellten Projekts.

Alternativ können Sie die Dateien auch importieren, indem Sie per Rechtsklick im Package Explorer

Import → Existing Projects into Workspace → Next wählen. In dem jetzt offenen Fenster, wählen Sie per Browse . . . das **DBLab - JDBC-Projekt**-Verzeichnis in den vorher entpackten Dateien aus und bestätigen Sie mit Finish.

Wurden die Ordner nicht als Paket sondern als einzelne Ordner in das **src** Verzeichnis eingefügt, war der erstellte Ordner kein Source Folder. Um dies zu ändern, wählen Sie per Rechtsklick auf das **src** Verzeichnis → Build Path → Use as Source Folder.

3. Nun müssen Sie noch den **Datenbanktreiber** für die Datenbank, die Sie benutzen möchten, in das Projekt einbinden. Die Treiber befinden sich im Verzeichnis **Treiber** in der Datei **DBLab - JDBC - Projekt.zip** aus dem ILIAS. Im Labor an der Hochschule ist das die dortige Datei **oracle-ojdbc6.jar** für das dort eingesetzte Oracle. Wählen Sie dazu per Rechtsklick auf Ihr Projekt → Build Path → Add External Archives. Im folgenden Fenster suchen Sie den Treiber in den Dateien, die Sie entpackt haben, unter dem Verzeichnis **Treiber** und wählen Sie Open.

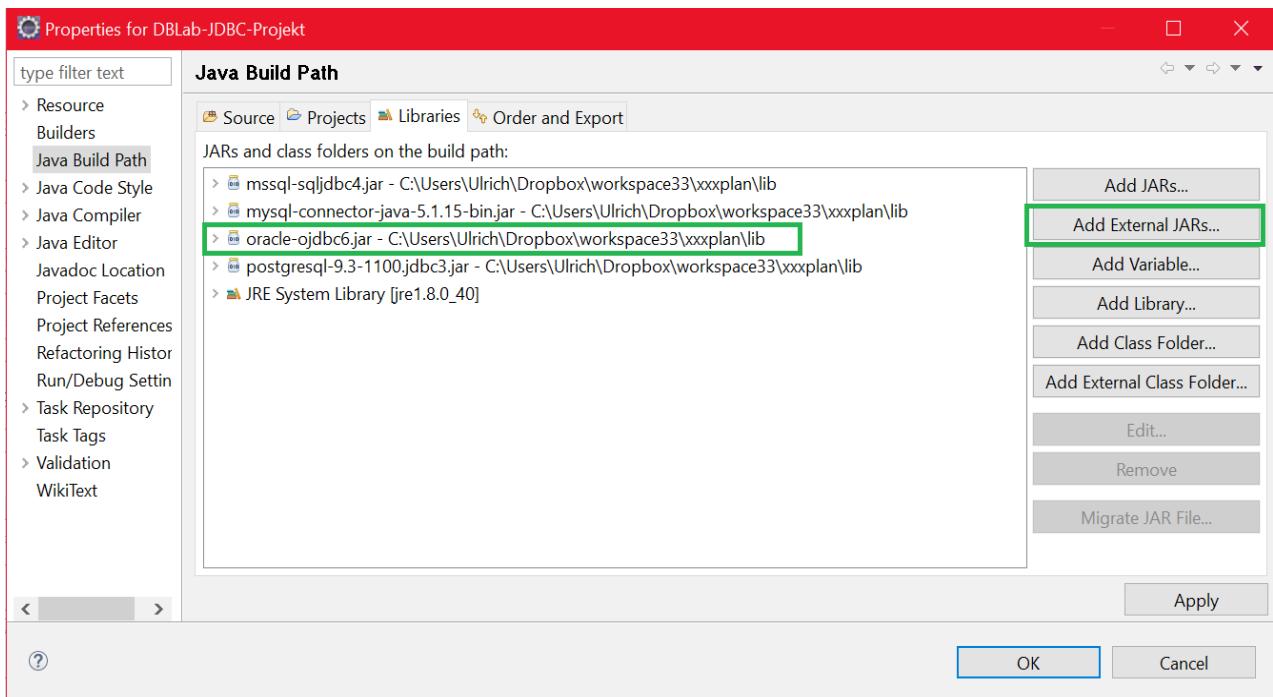


Bild 9.21: Eclipse Project Properties zum Beispiel für den Einsatz von Postgres zur Laborvorbereitung