

# Relationale Datenbanken

und

SQL



# Inhaltsverzeichnis

1	Relationale Datenbanken .....	6
1.1	Historie.....	6
1.2	ANSI SPARC .....	8
1.3	Historische Datenmodelle .....	11
1.3.1	Hierarchisches Datenmodell .....	11
1.3.2	Netzwerk-Datenmodell.....	11
1.3.3	Relation vs. Relationship.....	12
2	ERM (Entity Relationship Model).....	13
2.1	Entität .....	15
2.2	Attributarten.....	15
2.3	Relation und Kardinalität.....	16
2.3.1	1: n (one to many).....	16
2.3.2	n : m (many to many) .....	17
2.3.3	1 : 1 (one to one).....	18
2.3.4	Attribut einer Relation .....	18
2.3.5	Rekursive Assoziation.....	19
2.4	Aufgaben.....	20
3	ERM zu Tabellen .....	23
3.1	Tabellenmodell .....	24
3.1.1	Auflösung der zusammengesetzten Attribute in Einzelattribute .....	24
3.1.2	Auflösung der Mehrfachattribute.....	25
3.1.3	Umwandeln der Entitäten zu Tabellen .....	26
3.1.4	Auflösen der 1:n-Beziehung.....	27
3.1.5	Auflösen der n:m-Beziehung.....	28
3.1.6	Auflösen der 1:1-Beziehung.....	29
3.1.7	Fazit .....	30
3.2	Aufgaben.....	31
4	Normalformen .....	32
4.1	Normalform 1 .....	34
4.2	Normalform 2 .....	36

4.3 Normalform 3 .....	37
Aufgabe:.....	37
4.4 Referentielle Integrität .....	38
5 SQL -DDL .....	39
5.1 Anlegen/Löschen einer Datenbank .....	40
5.1.1 Zuweisen eines CharacterSets .....	40
5.1.2 Sortierreihenfolge .....	41
5.1.3 Löschen einer Datenbank .....	41
5.2 Anlegen einer Tabelle .....	42
5.2.1 Datentypen .....	44
5.2.2 Constraints .....	44
5.2.3 Ändern/Löschen von Datenstrukturen .....	49
5.2.4 Einfügen/Ändern/Löschen von Daten .....	51
6 SELECT .....	58
6.1 SELECT-Anfragen im Überblick .....	59
6.2 SELECT - FROM.....	60
6.2.1 Eingrenzen der Spalten .....	60
6.3 DISTINCT - Keine doppelten Zeilen .....	61
6.4 WHERE – Eingrenzen der Ergebnismenge .....	61
6.4.1 Einzelne Suchbedingung .....	62
6.4.2 BETWEEN AND – Werte zwischen zwei Grenzen.....	63
6.4.3 LIKE – Ähnlichkeiten.....	63
6.4.4 IS NULL – null-Werte prüfen .....	64
6.4.5 IN – genauer Vergleich mit einer Liste.....	65
6.4.6 EXISTS – schneller Vergleich mit einer Liste .....	66
6.4.7 Mehrere Bedingungen .....	66
6.4.8 Übungen.....	69
6.5 ORDER BY – Sortieren .....	72
6.6 FROM – Mehrere Tabellen verknüpfen.....	73
6.6.1 FROM und WHERE .....	73
6.6.2 JOIN ... ON .....	74
6.7 Zusammenfassung .....	74
6.8 Übungen .....	75
6.9 Lösung.....	75

---

6.10	Aggregatfunktionen.....	76
6.10.1	COUNT – Anzahl.....	76
6.10.2	SUM – Summe.....	77
6.10.3	MAX, MIN – Maximum, Minimum.....	77
6.10.4	AVG – Mittelwert.....	78
6.11	Gruppierungen .....	79
6.11.1	HAVING .....	83
6.11.2	Übungen.....	84
6.11.3	Lösungen .....	85
6.12	Joins.....	87
6.12.1	WHERE - EQUI JOIN .....	88
6.12.2	JOIN .....	95
6.12.3	Unterabfragen.....	122
7	Transaktionen .....	133
7.1	Probleme von Multi-User-Datenbanken .....	133
7.2	Eigenschaften einer Transaktion .....	135
7.2.1	Serialisierung als Garantie der Transaktion .....	136
7.2.2	Zusammenfassung Serialisierung.....	138
7.2.3	Isolation-Level .....	138
7.3	Beispiel zu Transaktionen in MySQL.....	139
7.3.1	DirtyReads in MySQL.....	140
7.3.2	PhantomRows in MySQL.....	142
7.3.3	Zusatzbeispiel IsolationLevel .....	142
7.4	Aufgabe zu Transaktionen .....	146
7.5	Weiterführende Links .....	146
8	Views.....	147
8.1	Aufgabe zu Views.....	151
8.2	Lösung zu Views.....	153
9	Trigger .....	156
9.1	Trigger erstellen.....	157
	Funktionen.....	163
10	Stored Procedures .....	164
10.1	Introduction.....	164
10.2	Block Structure .....	165

---

10.3	PL/pgSQL Block Structure.....	166
10.4	Errors and Messages .....	167
11	Indizes .....	170
12	grant, revoke - UserManagement .....	171
12.1	Aufgabe des Rechtesystems.....	171
12.2	Funktionsweise des Rechtesystems .....	171
12.3	Definitionen.....	173
12.4	Benutzerkonten mit GRANT und REVOKE erstellen.....	174
12.4.1	Beispiel für die Vergabe von Benutzerrechten .....	175
12.4.2	REVOKE - Wegnahme von Benutzerrechten.....	176
13	Aufgaben zu SQL.....	178
13.1	LUNA.....	178
13.1.1	Luna_Lösung .....	180
13.2	Nordwind.....	189
13.2.1	Lösung .....	191
13.3	Fragen zum Thema JOINS.....	197
13.3.1	SQL-DDL.....	203
14	Links .....	206
15	Literatur- und Quellenverzeichnis .....	206
16	Index .....	206

# 1 Relationale Datenbanken

## 1.1 Historie

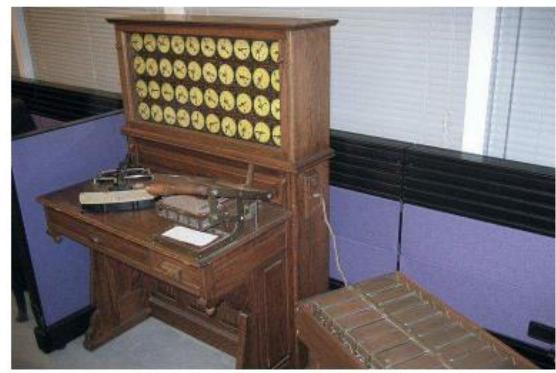
- Herman Hollerith (1860 – 1929)
  - Hollerith-Lochkartenverfahrens zur Datenverarbeitung
  - Z.B. Volkszählung USA 1890 (2 Jahre, 43 Maschinen, 500 Personen)
  - Gründete 1896 die *Tabulating Machine Company*
  - Fusion zur *Computing Tabulating Recording Corporation* (CTR)
  - 1924 umbenannt: *International Business Machines Corporation* (IBM)



<http://en.wikipedia.org/wiki/Hollerith>

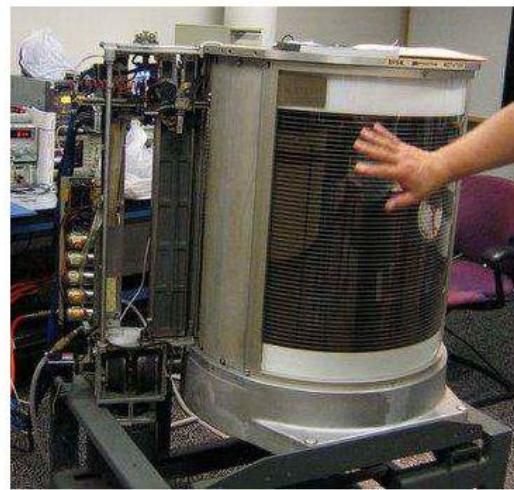
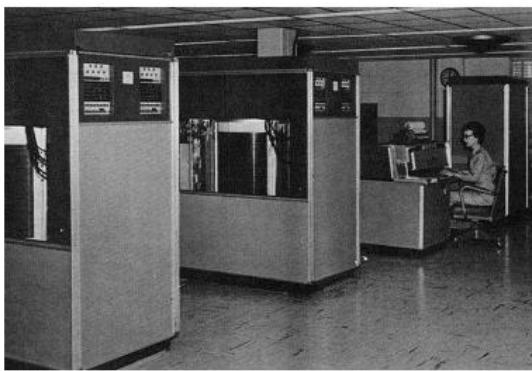
L	A	B	C	A	B	C	L	M	N	G	A	C	C	S	M	H	M	E	V	D	G	E	F	D	A
E	P	E	F	D	F	E	L	O	C	Y	S	M	S	F	V	O	C	X	T	S	D	A	A	A	
L	G	H	I	G	H	I	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
C	K	L	M	K	L	M	M	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
G	N	O	P	N	O	P	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
L	Q	R	S	Q	R	S	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
K	4	5	6	5	6	7	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
R	6	7	8	7	8	9	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
Q	8	9	10	9	10	11	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
A	10	11	12	11	12	13	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
Y	12	13	14	13	14	15	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
3994	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9

Hollerith Lochkarte



Lochkartenleser mit Zählwerken

- Erste Festplatte: IBM 350 RAMAC
  - Random Access Method of Accounting and Control
  - 3,75 MB
  - 50 Platten, 1200 U/min
  - 3.200 \$ / Monat
  - 1 Tonne



[http://en.wikipedia.org/wiki/File:IBM\\_350\\_RAMAC.jpg](http://en.wikipedia.org/wiki/File:IBM_350_RAMAC.jpg)

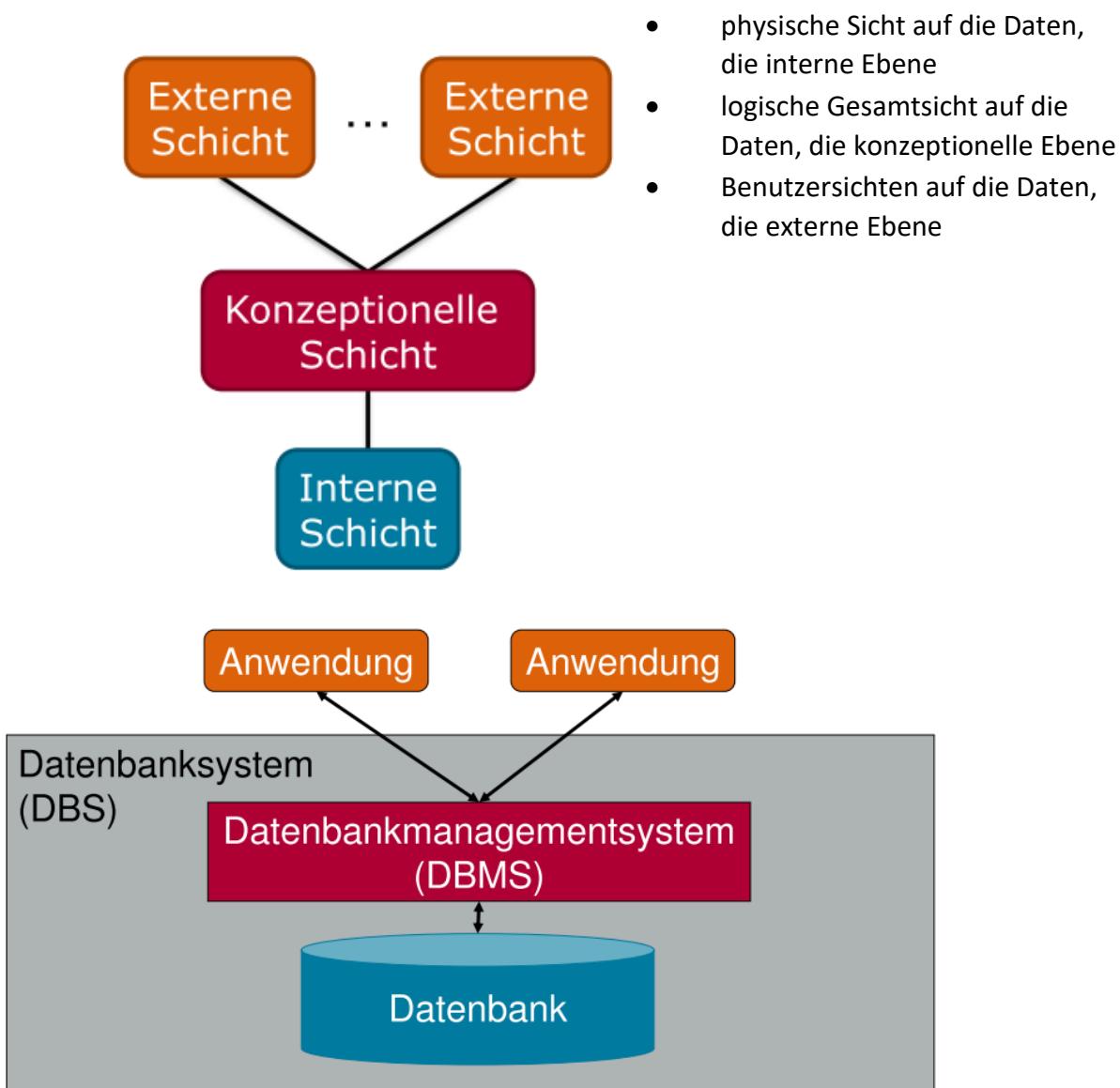
- 60er: DBS basierend auf hierarchischem Modell, Netzwerkmodell
  - Zeigerstrukturen zwischen Daten
  - Schwache Trennung physische / konzeptuelle Ebene
  - Navigierende DML
- 70er und 80er: Relationale Datenbanksysteme
  - Daten in Tabellenstrukturen
  - 3-Ebenen-Konzept zur Trennung zw. physischer / konzeptueller Ebene
  - Deklarative DML
- 80er und 90er
  - Immer größer / Immer kleiner
  - Objektorientierte DBs
- 2000er
  - Spezialisierung auf neue Daten-Arten
- 2010er
  - Web-scale, NoSQL, neue Hardware
- 70er: Relationale Datenbanksysteme
  - Geräte- und Datenunabhängigkeit, redundanzfrei, konsistent
- 1970: Ted Codd (IBM)
  - Relationenmodell als konzeptionelle Grundlage relationaler DBMS
- 1974: System R (IBM)
  - erster Prototyp eines RDBMS
  - ca. 80.000 LOC (PL/1, PL/S, Assembler), ca. 1,2 MB Codegröße
  - Anfragesprache SEQUEL ( $\neq$  SQL)
  - erste Installation 1977
- 1975: University of California at Berkeley (UCB)
  - Ingres
  - Anfragesprache QUEL
  - Vorgänger von Postgres, Sybase, . . .
  - Michael Stonebraker
- 1979: Oracle Version 2



## 1.2 ANSI SPARC

Nachdem 1969 das Gremium CODASYL ein Datenbankkonzept basierend auf zwei Ebenen entwickelt hatte, in dem es um die Unabhängigkeit von einer speziellen Programmiersprache ging, wurde 1975 vom Gremium ANSI/SPARC ein Standard verabschiedet, der drei Ebenen definiert. Diese drei Ebenen gewährleisten nun eine Unabhängigkeit von der Programmiersprache wie auch von der Hardware. Dieses ANSI-3-Ebenenmodell hat sich bis in die Gegenwart als Standard für die Architektur von Datenbanksystemen durchgesetzt.

Er besteht aus drei Abstraktionsebenen

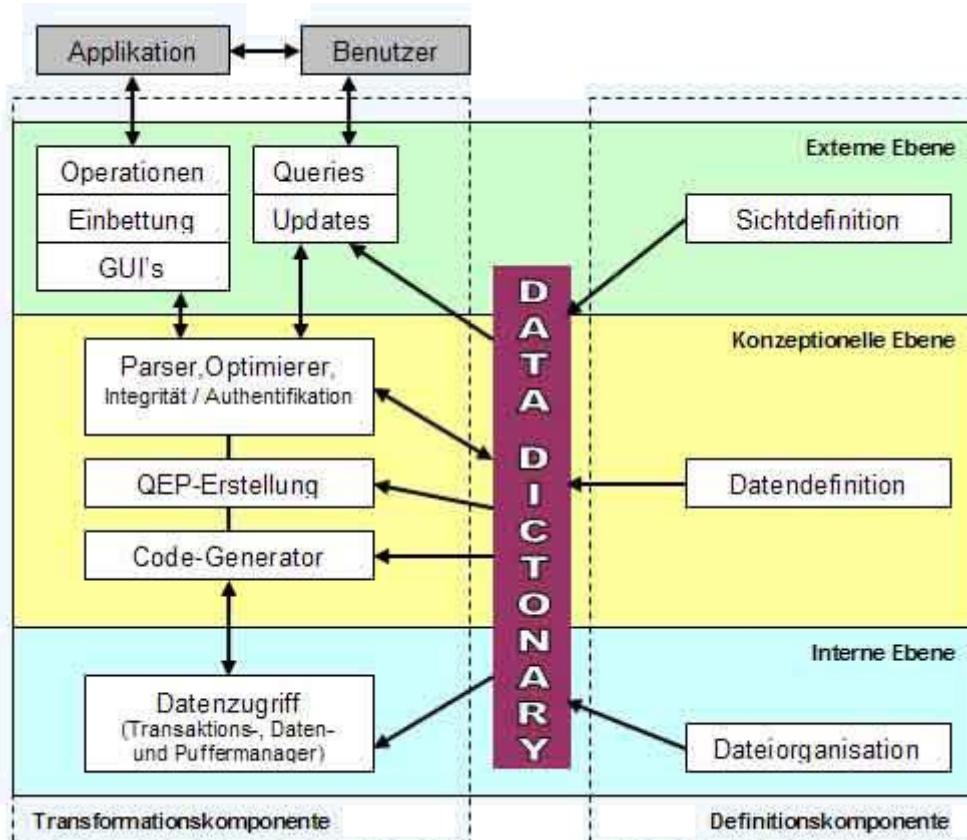


## Externe Ebene

Die externe Ebene enthält die Benutzersichten auf die Daten, bzw. den Ausschnitt aus den Gesamtdaten, den spezielle Benutzergruppen benötigen. Benutzersichten können hilfreich sein, um bestimmten Benutzern bzw. Benutzergruppen nur Ausschnitte des Gesamtdatenmodells zugänglich zu machen (Rechteverwaltung), oder für Umbenennungen von Attributen, um dem unterschiedlichen Vokabular von Fachabteilungen Rechnung zu tragen. Als Beispiel sei hier die Begriffe Lieferant/Kunde und Debitor/Kreditor genannt. Diese Ebene wird vom Datenbankentwickler verwaltet.

## Konzeptionelle / konzeptuelle Ebene

Diese Ebene hat die logische Darstellung der Gesamtsicht der Daten in einem speziellen Datenmodell, z.B. dem relationalen Datenmodell zum Inhalt. Ziel ist hier eine Unternehmensgesamtsicht zu erstellen, was ohne detaillierte Kenntnisse über relevante Informationen, Logik, Abläufe und Zusammenhänge im Unternehmen nicht gelingen kann. Diese Gesamtsicht beschreibt alle Daten, deren Beziehungen untereinander und auch die Konsistenzanforderungen in Form z.B. von Integritätsbedingungen.

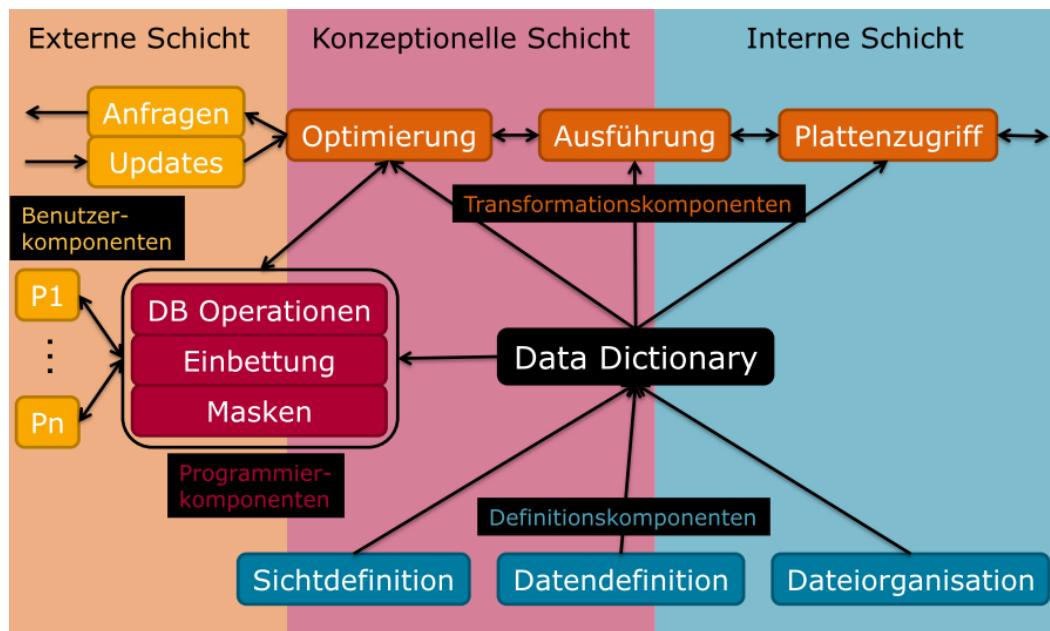


Die verschiedenen Unternehmensansichten aus der externen Ebene aus den verschiedenen Abteilungen werden zu einem Gesamtmodell zusammengeführt. Zentrales Ziel ist hier eine redundanzfreie Speicherung der Daten, z.B. durch Normalisierung. Innerhalb dieser Ebene wird in der Regel zweistufig verfahren. Zuerst wird ein konzeptionelles Schema (z.B. ER-Modell, UML, ...) erstellt, welches noch unabhängig vom konkret eingesetzten Datenmodell resp. Datenbanksystem ist. Davon wird dann in einem zweiten Schritt das logische Datenbankschema abgeleitet, welches schon auf das Modell eines bestimmten Datenbanktyps (relational, objektorientiert, ...) angepasst ist. Diese Ebene wird primär vom Datenbankentwickler verwaltet.

## Interne Ebene

Dies stellt die physikalische Implementierung des konzeptionellen Schemas in Abhängigkeit zum verwendeten DBS dar. Information über die Art und den Aufbau der Datenstrukturen auf dem physikalischen Speicher und Zugriffsmechanismen sind Bestandteil der internen Ebene. Diese Ebene wird vom Datenbankadministrator verwaltet. Zu seinen Aufgaben gehören insbesondere Maßnahmen zur Laufzeitverbesserung (Performance), Verwaltung von Benutzerrechten, Verfügbarkeit (Ausfallsicherheit), Datensicherheit und Datenschutz und vieles mehr. Trotz aller SQL-Standardisierungen unterscheiden sich die Hersteller wie IBM, Oracle, MySQL, Microsoft, ... hier doch teilweise gravierend.

Für die sogenannten klassischen DBS, wie z.B. Netzwerk- relationale bis hin zu objektorientierten Datenbanksystemen stellt dieses ANSI-3-Ebenenmodell wirklich ein Paradigma dar, das zwingend eingehalten wurde mit eben dem Vorteil der Datenunabhängigkeit.



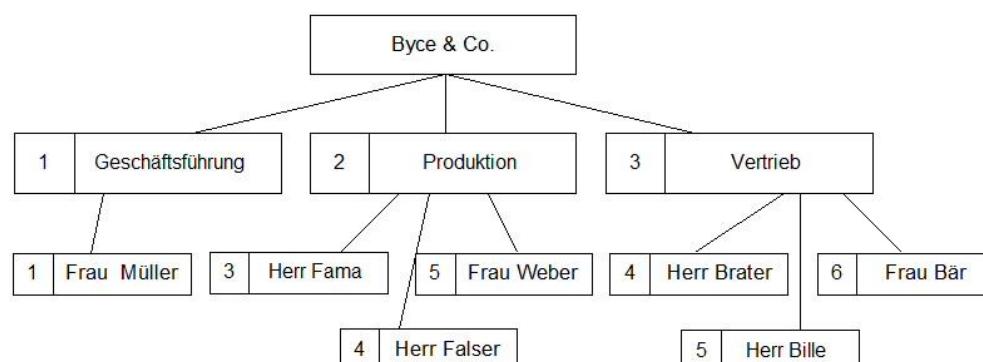
## 1.3 Historische Datenmodelle

Bereits vor dem relationalen Datenbankmodell gab es Konzepte zur Datenhaltung.

### 1.3.1 Hierarchisches Datenmodell

Ein hierarchisches Datenmodell ist ein Datenmodell in der Art eines Baumes. Es ist historisch gesehen das älteste Datenmodell und stammt aus den 1960er Jahren. Die Adressverknüpfungen werden gemeinsam mit den Daten gespeichert. Ein Datensatz kann höchstens mit einem übergeordneten, sowie mehreren untergeordneten Datensätzen in Beziehung stehen. Dieser Datenbanktyp wurde von IBM mit dem System IMS/DB implementiert und ist sogar heutzutage noch im Einsatz, auch wenn das Datenmodell selber eher von historischem Interesse ist. Eine gewisse Renaissance erlebt die hierarchische Datenspeicherung mit XML.

Aufgrund der baumstrukturierten Adressverweise zwischen den Daten, die ja bereits bei der Speicherung bestimmt werden müssen, sind einerseits lesende Zugriffe äußerst schnell, andererseits lesende Verknüpfungen sehr unflexibel, denn die Adresspfade müssen ja schon beim Einfügen der Daten festgelegt werden.



### 1.3.2 Netzwerk-Datenmodell

Auszug aus: <http://de.wikipedia.org/wiki/Netzwerkdatenbankmodell>

“Nach der Vorstellung beider Datenbankmodelle (Relational vs. Netzwerk) anfang der 1970er Jahre gab es schon zwanzig Jahre lang hocheffiziente Netzwerkdatenbanksysteme auf mittleren und großen Mainframes für höchste Transaktionsraten, bis relationale Datenbanksysteme bezüglich der Performance einigermaßen gleichziehen konnten. Nicht ohne Grund ist das hierarchische Datenbanksystem von IBM von Ende der 1960er noch heute bei vielen IBM-Kunden im Einsatz. Auch Abfragesprachen für Ad-hoc-Anfragen standen auf Netzwerksystemen zur Verfügung, beispielsweise QLP/1100 von Sperry Rand. Heute wird das Netzwerkdatenbankmodell hauptsächlich auf Großrechnern eingesetzt.

Bekannte Vertreter des Netzwerkdatenbankmodells sind UDS (Universal Datenbank System) von Siemens, DMS (Database Management System) von Sperry Univac. Mischformen zwischen Relationalen Datenbanken und Netzwerkdatenbanken wurden entwickelt – z. B. von Sperry Univac (RDBMS Relational Database Management System) und Siemens

(UDS/SQL), mit der Absicht, die Vorteile beider Modelle zu verbinden.

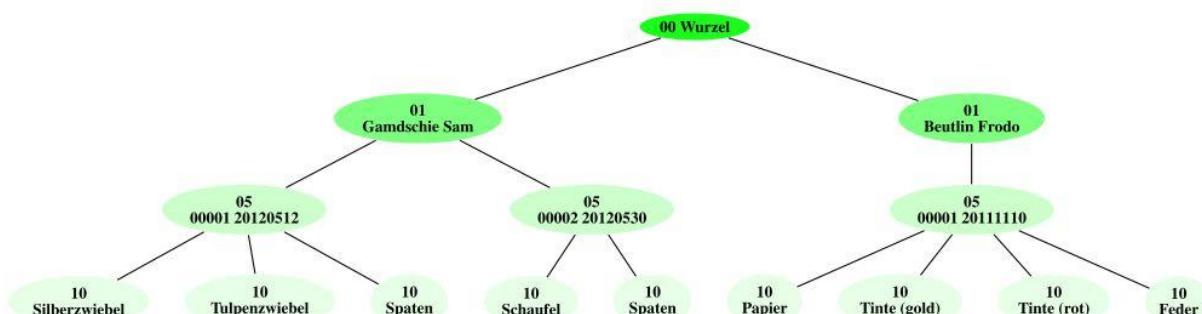
Seit den 1990er Jahren wird das Netzwerkdatenbankmodell vom relationalen Datenbankmodell mehr und mehr verdrängt. Mit der Idee des semantischen Webs gewinnt das Netzwerkdatenbankmodell wieder mehr an Bedeutung. Mittlerweile existieren eine Reihe von Graphdatenbanken, die man als moderne Nachfolger der Netzwerkdatenbanken bezeichnen könnte. Ein wesentlicher Unterschied ist allerdings die Möglichkeit der dynamischen Schemaänderung.“

### 1.3.3 Relation vs. Relationship

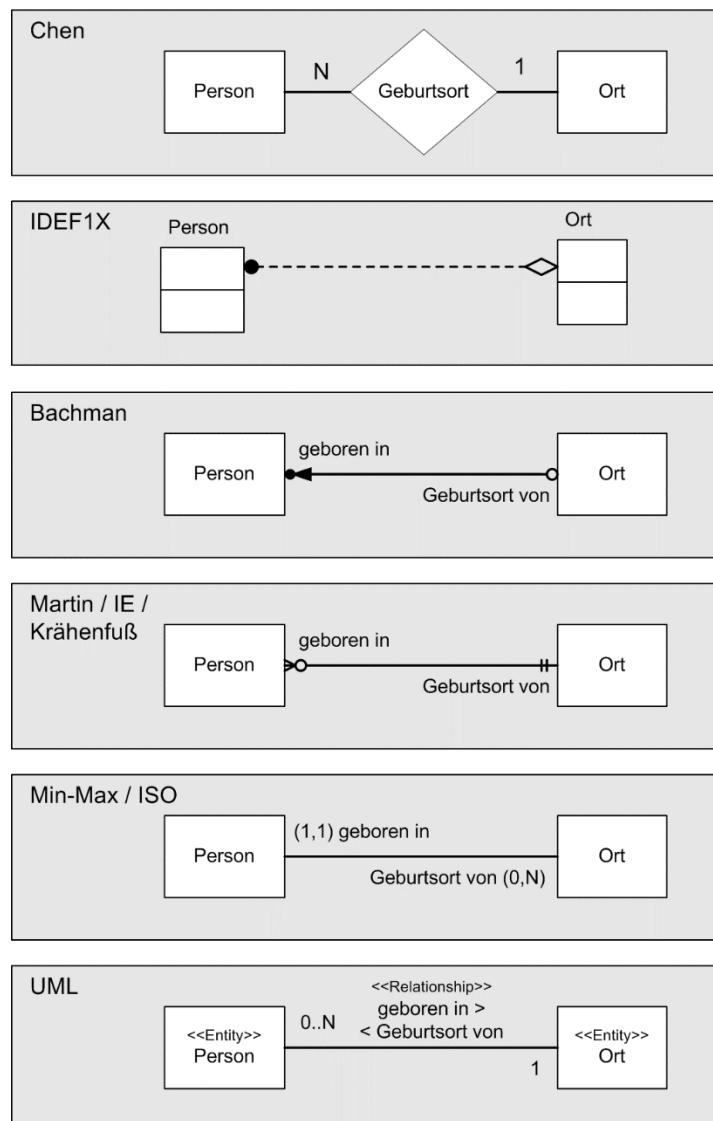
Der Begriff relationale Datenbank selbst ist häufig missverständlich beschrieben. „Zwischen den Daten bestehen Beziehungen (Relationen)“, daher der Name. **Nun ist aber der englische Begriff für Beziehung relationship und nicht relation.** Relation ist ein wohldefinierter Fachbegriff und kann in etwa mit „Tabelle“ übersetzt werden. Relationale Datenbanken sind also in Tabellen organisiert. Die Daten werden in Spalten und Zeilen organisiert. Das hört sich so selbstverständlich an, dass man sich klar machen muss, dass es auch Alternativen gibt. Als Beispiel sei hier die Ablage von Bestelldaten eines Kunden im Format von COBOL Data Division angegeben:

```
01 12345 Gamdschie Samweis Beutelhaldenweg 5 67676 Hobbingen Privatkunde
05 00001 20120512 12:30:00 bezahlt
10 7856 Silberzwiebel 15 6.15 €
10 7863 Tulpenzwiebel 10 32.90 €
10 9015 Spaten 1 19.90 €
05 00002 20120530 17:15:00 offen
10 9010 Schaufel 1 15.00 $
10 9015 Spaten 1 19.90 €
01 12346 Beutlin Frodo Beutelhaldenweg 1 67676 Hobbingen Privatkunde
05 00001 20111110 11:15:00 bezahlt
10 3001 Papier (100) 10 23.00 €
10 3005 Tinte (gold) 1 55.70 €
10 3006 Tinte (rot) 1 6.20 €
10 3010 Feder 5 25.00 €
```

Die Informationen sind hier hierarchisch organisiert. Die Zugehörigkeit einer Information (beispielsweise eines bestellten Artikels) ergibt sich aus der Position, wo die Information steht, oder besser: unter welcher Information diese steht. In der grafischen Darstellung wird dies deutlicher.



## 2 ERM (Entity Relationship Model)



In der Informatik ist es üblich, dass man EDV-Systeme als Modell darstellt. Vielleicht kennen Sie schon solche Modelle aus der Programmierung: Programmablaufplan nach [ISO85] (ISO/IEC 5807, DIN 66001), Struktogramm nach [Nor85] (DIN 66261), Datenflussdiagramm nach [ISO85] (ISO 5807), UML-Klassendiagramme usw.

Die Modellierungstechnik für relationale Datenbanken ist das Entity Relationship Model oder auch ER-Modell. Dieses gibt es in verschiedenen Notationen.

**Chen:** Hier werden die Tabellen als Rechtecke dargestellt. Die Spalten der Tabellen werden als Blasen um die Tabelle herum notiert. In die Blase schreibt man den Spaltenamen. Der Name des Primärschlüssels wird dabei unterstrichen.

**IDEF1X:** Die Tabellen werden auch hier als Rechtecke (ggf. mit abgerundeten Ecken) dargestellt. Die Spalten werden aber innerhalb des Rechtecks notiert. Die Primärschlüssel Spalten werden dabei durch eine Linie von den anderen Spalten abgetrennt. Diese Notation ist der Quasi-standard US-amerikanischer Behörden.

**Krähenfuß (Martin):** Die Tabellen werden in Rechtecken dargestellt. Diese enthalten die Spaltennamen. Vor dem Spaltennamen ist Platz für eine weitere Spezifikation der Spalte (z.B. als Fremdschlüssel).

## Ausgangssituation

Die Datenbank FIRMA verwaltet die Angestellten, Abteilungen und Projekte einer Firma. Die Phase der Erfassung und Analyse der Anforderungen ist abgeschlossen und die Designer haben folgende Beschreibung der Firma erstellt.

Die Firma ist in Abteilungen organisiert. Jede Abteilung hat eine eindeutige Bezeichnung, eine eindeutige Nummer und einen bestimmten Angestellten, der die Abteilung leitet. Wir verfolgen das Anfangsdatum, ab dem dieser Angestellte die Leitung der Firma übernommen hat. Eine Abteilung verfügt über mehrere Standorte

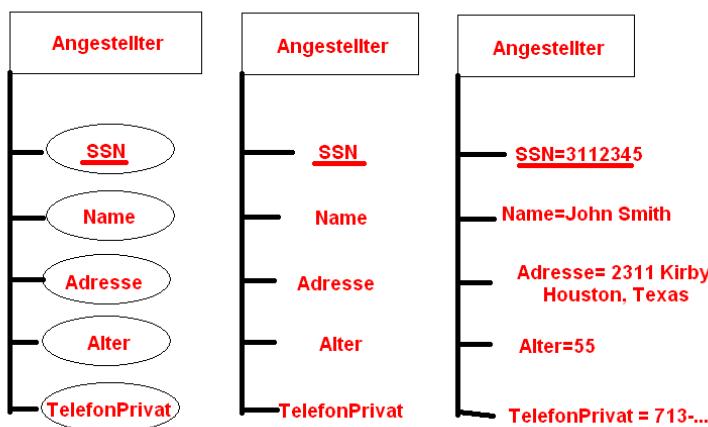
Eine Abteilung kontrolliert eine Reihe von Projekten, die jeweils einen eindeutigen Namen, eine eindeutige Nummer und einen einzigen Standort haben. Die Abteilung kennt die Anzahl ihrer Mitarbeiter.

Wir speichern zu jedem Angestellten den Namen, die Sozialversicherungsnummer, die Adresse, das Gehalt, das Geschlecht und das Geburtsdatum. Ein Angestellter wird einer Abteilung zugewiesen, kann aber an mehreren Projekten arbeiten, die nicht unbedingt alle von der gleichen Abteilung kontrolliert werden. Wir verfolgen die Stundenzahl pro Woche, die ein Angestellter an jedem Projekt arbeiten, und den unmittelbaren Vorgesetzten jedes Angestellten.

Zu Versicherungszwecken möchten wir die Familienangehörigen jedes Mitarbeiters verfolgen. Wir führen also jeden Angehörigen mit Vornamen, Geschlecht, geburtsdatum und Verwandtschaftsgrad zum jeweiligen Angestellten.

## 2.1 Entität

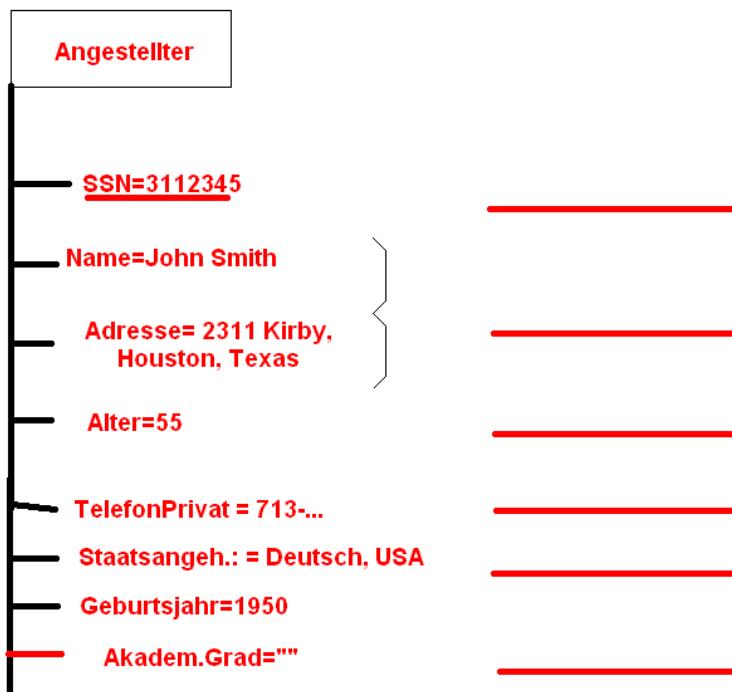
Das vom ER-Modell dargestellte Basisobjekt ist die sog. Entität . Sie kann real existieren (z.B. Auto) oder lediglich konzeptionell (z.B. eine Firma) sein. Jede Entität hat Attribute, d.h. bestimmte Eigenschaften, die sie beschreiben. Ein Attribut einer Entität wird mit sog. Attributwerten belegt.



Nebenstehende Abbildung zeigt mehrere Darstellungsformen einer Entität.

## 2.2 Attributarten

Im ER-Modell kommen mehrere Attributarten vor:

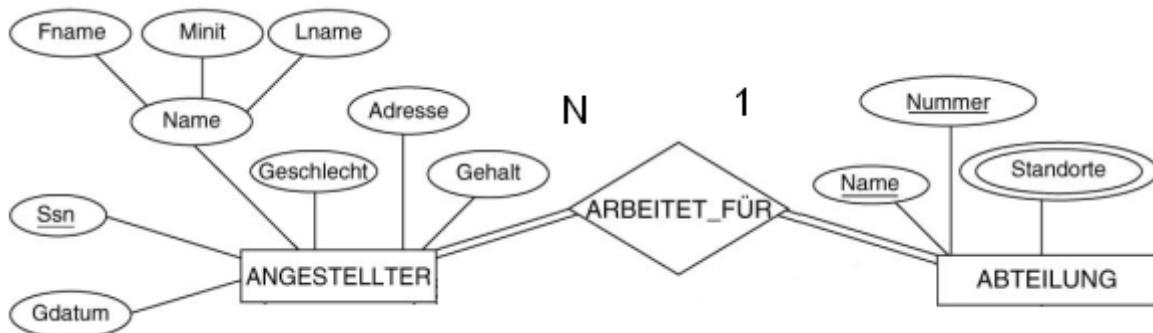


## 2.3 Relation und Kardinalität

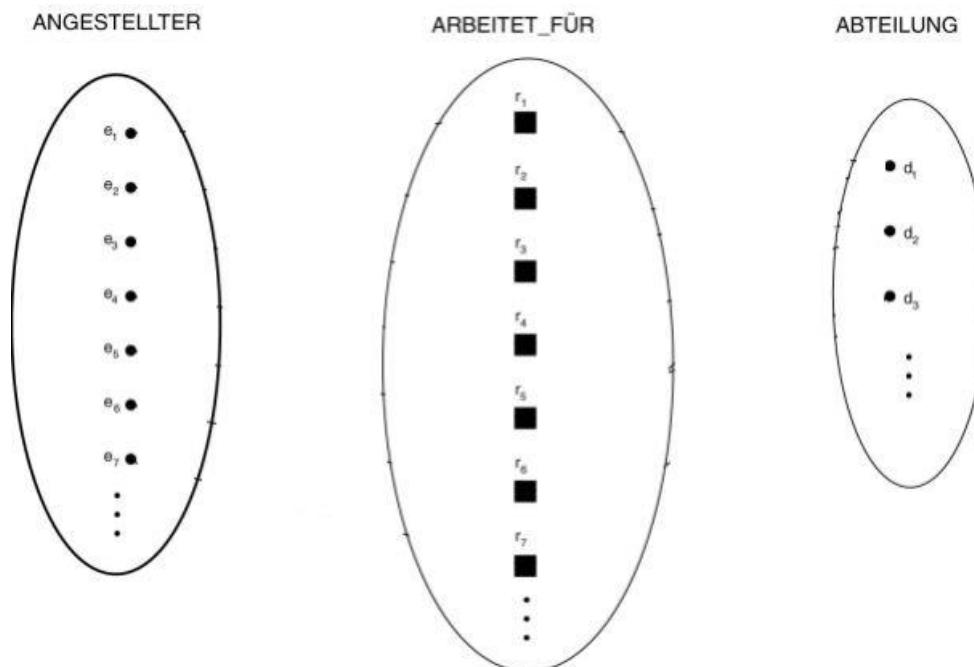
Eine Entität besitzt neben ihren eigentlichen Attributen auch Beziehungen zu anderen Entitäten. Diese werden durch Linien zwischen den Entitäten (Relation) zum Ausdruck gebracht. Neben dieser Linie werden durch sog. **Kardinalitäten** eine Aussage über die Anzahl der Beziehungsobjekte der jeweils anderen Seite zu sich selbst getroffen.

Dies mündet in die drei grundlegenden Beziehungsmengen:

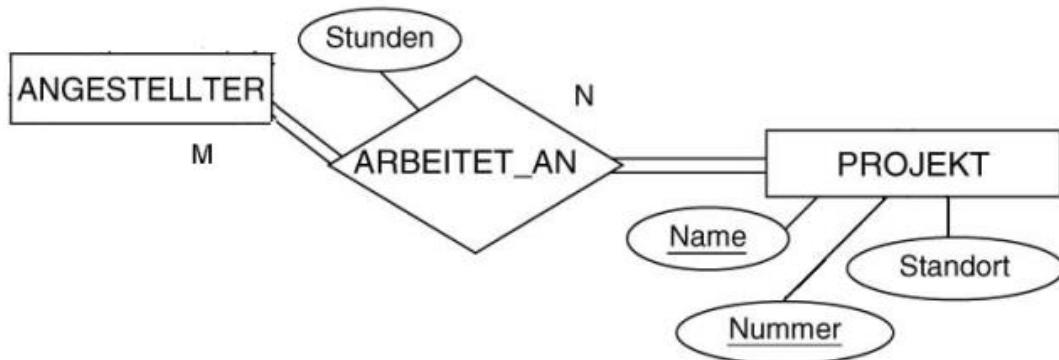
### 2.3.1 1: n (one to many)



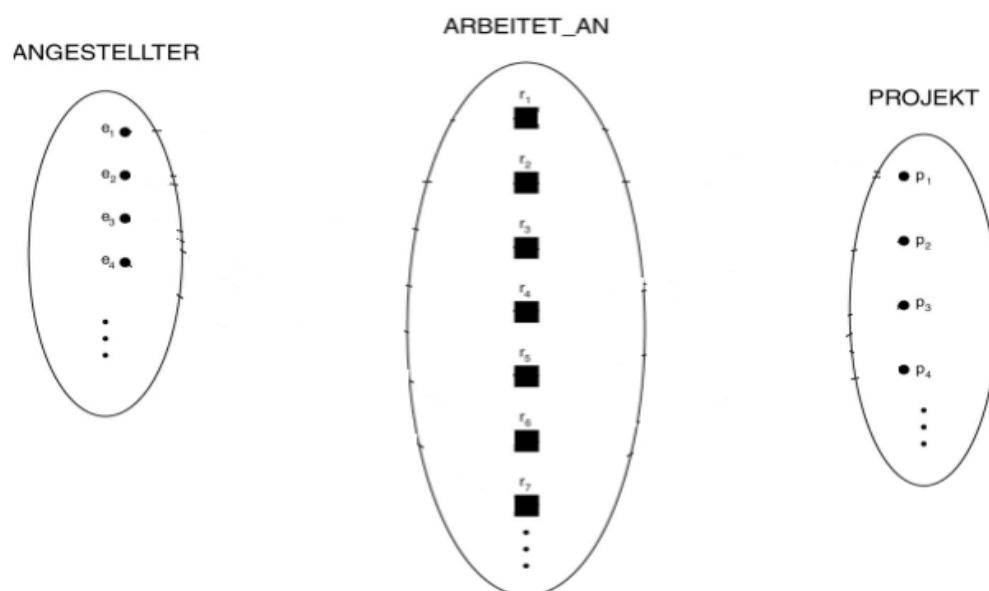
Hier hält eine Entität jeweils mehrere Beziehungen zu Entitäten der anderen Seite, die andere Seite jedoch nur eine Beziehung. So hat eine Abteilung, z.B. Einkauf, zwar viele Angestellte, ein Angestellter arbeitet aber nur in einer Abteilung



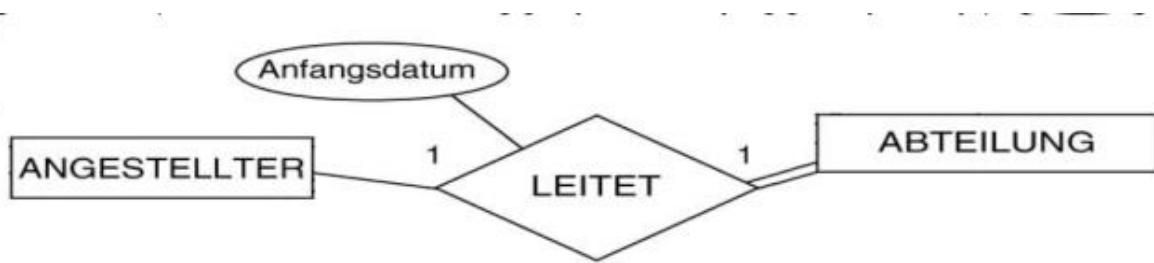
### 2.3.2 n : m (many to many)



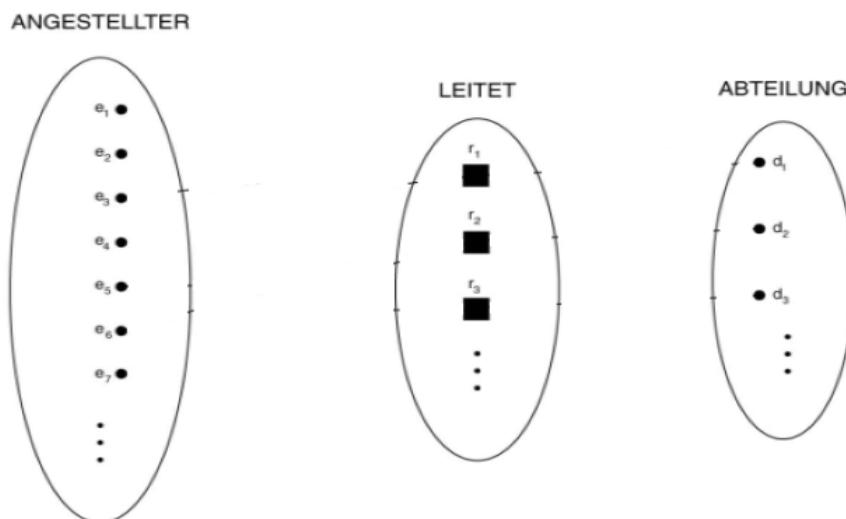
Hier hat jede Entität jeweils mehrere Beziehungen zu Entitäten der anderen Seite. So arbeitet ein Angestellter in vielen Projekten, an einem Projekt können aber auch viele Angestellte arbeiten.



### 2.3.3 1 : 1 (one to one)

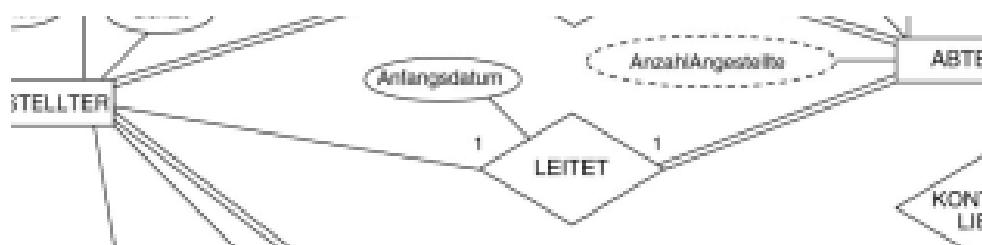


Hier hat jede Entität maximal eine Beziehung zur anderen Entität und umgekehrt.

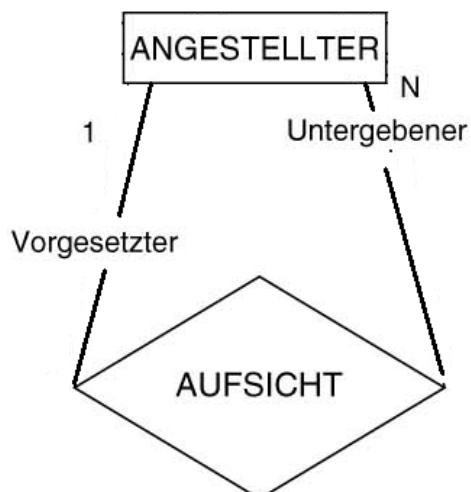


### 2.3.4 Attribut einer Relation

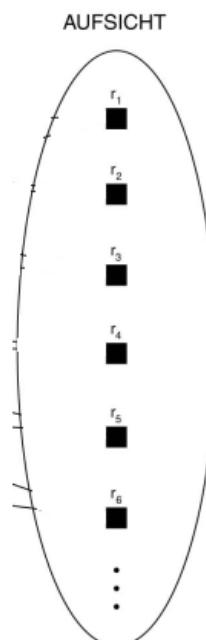
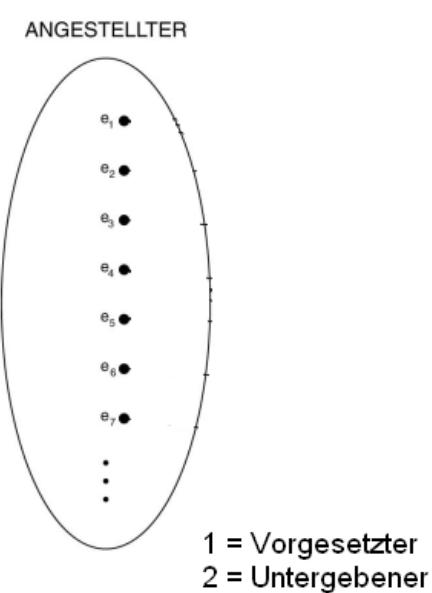
In manchen Fällen können auch die Relationen Träger von Attributen sein. So ist z.B. der Beginn der Leitung einer Abteilung ein Attribut der Relation und nicht eines der beteiligten Entitäten.



### 2.3.5 Rekursive Assoziation



In manchen Fällen kann eine Assoziation mit beiden Enden auf die gleiche Entität verweisen. So ist der Vorgesetzte eines Angestellten ebenfalls ein Angestellter. Um die beiden Teile der Assoziation besser auseinanderhalten zu können, schreibt man auf die jeweilige Seite die sog. Rolle, die eine Entität im Bezug auf seine Beziehungsseite spielt.



## 2.4 Aufgaben

### 1. Bibliothek

Folgender Sachverhalt beschreibt die Verhältnisse in einer Bibliothek

In einer Bibliothek befinden sich zahlreiche Bücher, die jeweils von einem oder mehreren Autoren verfasst wurden . Von einigen Autoren existieren bereits verschiedene Bücher. Jedes Buch stammt von einem bestimmten Verlag (wobei die Verlage natürlich mehrere Bücher veröffentlicht haben) und befindet sich an einem definierten Standort. Ein (Regal-)Standort umfasst allerdings mehrere Bücher. Entliehen werden die Bücher von Kunden, die mehrere Bücher ausleihen können.

Lösung

### 2. Pizza-Service

ZurZeit finanzieren Sie Ihr kostspieliges Privatleben als EDV-Berater bei einem Pizzaservice. Aufgrund seiner guten Pizzen fallen immer mehr Bestellungen an, so dass die wachsende Zahl an Zetteln zu einem Chaos in der Küche geführt hat. Eine Lösung verspricht sich der Inhaber des Pizzaservices durch die Verwendung einer Datenbank, mit der alle Bestellungen und Lieferungen verwaltet werden können. Dazu liegt Ihnen folgender zu modellierender Wirklichkeitsausschnitt vor:

Der Pizzaservice bietet verschiedene Pizzen an (gekennzeichnet durch eine Nummer). Jede Pizza besteht aus Teig, Tomaten, Käse und diversen Zutaten (Salami, Schinken, Pilze usw.). Die Zutaten zu einer Pizza sind fest vorgegeben; eine maximale Anzahl an (möglichen) Zutaten ist jedoch nicht vorgegeben. Es gibt eine Kundendatei, in der die Kunden anhand ihrer Telefonnummern eindeutig identifiziert werden. Jede Bestellung kommt von genau einem Kunden, umfasst beliebig viele Pizzen (aber mindestens eine) und wird von einem Fahrer ausgeliefert. Die Bestellungen werden durch eine fortlaufende Bestellnummer identifiziert. Auf einer Tour kann ein Fahrer mehrere Bestellungen ausliefern.

### 3. Bus

Sie werden beauftragt, für ein Busreiseunternehmen eine Datenbank zu entwickeln. Gehen Sie hierbei von folgendem Wirklichkeitsausschnitt aus, der abgebildet werden soll. Das Unternehmen bietet ausschließlich Städtereisen an. Für verschiedene europäische Städte existieren bestimmte Reiseangebote (jeweils genau eines), die Eigenschaften wie den Namen der Stadt, den (konstanten) Preis sowie die Reisedauer (Anzahl der Übernachtungen) besitzen. Für jede Stadt bzw. das entsprechende Reiseangebot existieren verschiedene terminliche Ausprägungen. Eine solche konkrete Reise wird jeweils von genau einem Busfahrer durchgeführt. Die Busfahrer des Unternehmens besitzen jeweils nur Kenntnisse für bestimmte Städte; dementsprechend besitzen sie nur die Fähigkeit zur Durchführung bestimmter Reisen. Die Kunden des Unternehmens sind ebenfalls in dem System abzubilden. Hierbei ist auch abzubilden, welche Kunden welche konkreten Reisen gebucht haben bzw. hatten.

### 4. Olympia

Sie arbeiten in einer EDV-Beratungsfirma. Diese Beratungsfirma erhält den Auftrag, für das IOC (International Olympic Committee) eine Datenbank für die nächsten Olympischen Spiele zu erstellen, die folgenden Wirklichkeitsausschnitt enthalten soll.

Die einzelnen Wettkämpfe der Olympischen Spiele sind durch den Namen der Sportart, den Termin und die Sportstätte gekennzeichnet. An jedem Wettkampf nehmen mehrere Sportler teil, die durch eine Startnummer identifiziert werden und außerdem natürlich, wie jede Person, einen Namen besitzen. Jeder Wettkampf wird von einem Schiedsrichter geleitet, dem für diese Spiele eine eindeutige Personalnummer zugeordnet wurde. Die Schiedsrichter werden bei einem Wettkampf von verschiedenen Helfern unterstützt, die ebenfalls eine eindeutige Personalnummer erhalten haben. Die Sportler und Schiedsrichter gehören jeweils einer Nation an, zu der der Name des Mannschaftsleiters und eine Telefonnummer für Rückfragen abgespeichert werden. Dies gilt zwar ebenfalls für die Helfer, soll jedoch hier nicht berücksichtigt werden. Erstellen Sie ein ER-Modell mit den wesentlichen Informationen bzgl Attributen, Beziehungen und Kardinalitäten.

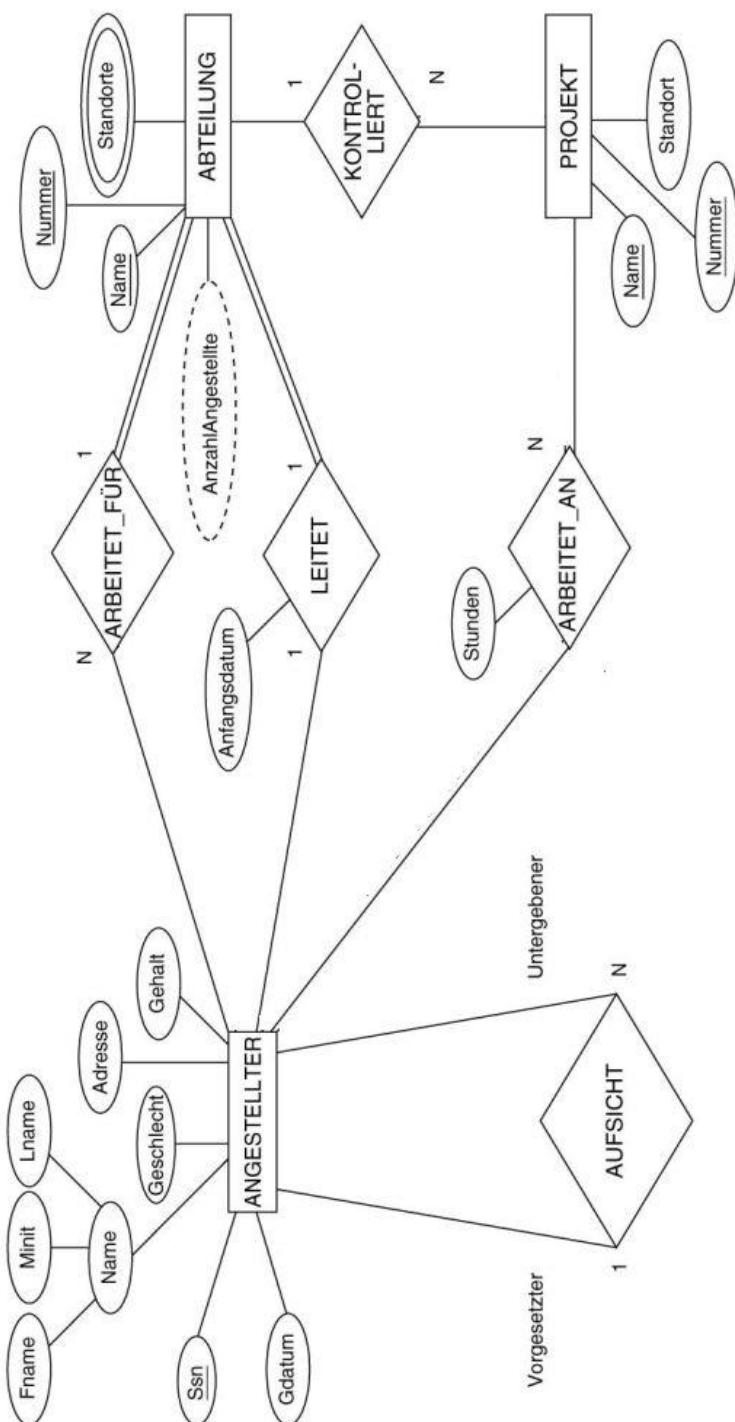
## 5. Hotel

Im Rahmen eines Hotel-Softwareprojektes werden Sie gebeten, die Datenhaltungsschicht zu programmieren. Erstellen Sie ein ER-Modell nach Auszug aus dem Lastenheft. Die Anwendung soll folgende Bereiche abdecken:

1. Kundenverwaltung Zu jedem Kunden werden seine persönlichen Daten sowie Rechnungsanschriften erfasst. Falls Kunden einer bestimmten Firma angehören, sollen diese einer Firma zugeordnet werden können, um spätere Rabatte berechnen zu können.
2. Raumreservierung Für jeden Raum wird seine Bezeichnung, Ausstattung und Tarifgruppe erfasst. Ebenso sollen pro Raum mehrere Reservierungszeiträume erfasst werden können. Diese Zeiträume sollen einem Kunden zugeordnet werden.
3. Rechnungsstellung Für jeden Kunden soll anhand der Belegzeiten und sonstiger Leistungen (Mini-Bar, Sauna etc.) eine Rechnung zusammengestellt werden können.
6. Schwache Entität Suchen Sie im Internet nach diesem Begriff und erläutern Sie seine Bedeutung

### 3 ERM zu Tabellen

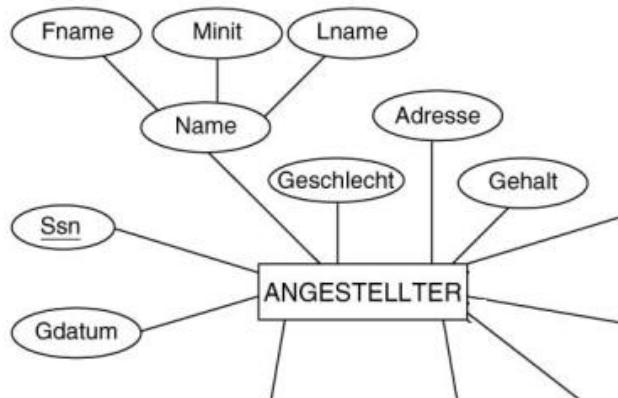
Das ERM diente als Annäherung an das zu programmierende Datenmodell; es muss nun durch weitere Schritte verfeinert werden. Das ERM wird in Tabellen aufgelöst, die Attribute der Entitäten werden zu Spalten der Tabellen. Als Ausgangssituation dient folgendes ERM:



### 3.1 Tabellenmodell

In einem ersten Schritt müssen die Entitäten und ihre Beziehungen in ein Tabellenmodell umgewandelt werden. Dies erfolgt unter Zugrundelegen folgender Regeln.

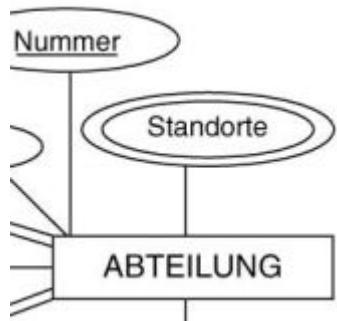
### 3.1.1 Auflösung der zusammengesetzten Attribute in Einzelattribute



### 3.1.2 Auflösung der Mehrfachattribute

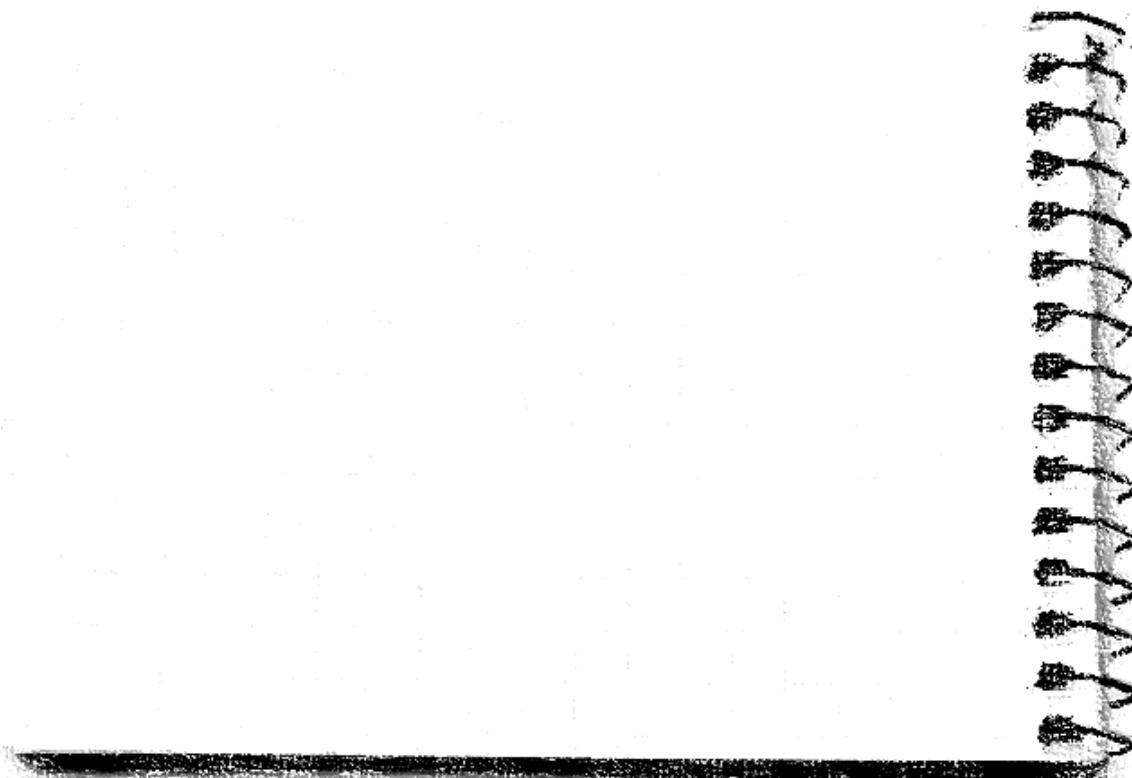
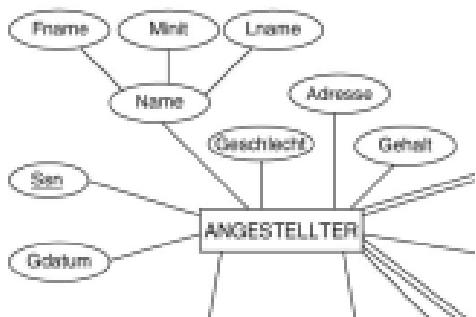
Mehrfachattribute müssen zu eigenen Entitäten und einer 1:n-Beziehung umgesetzt werden.

#### Lösung



### 3.1.3 Umwandeln der Entitäten zu Tabellen

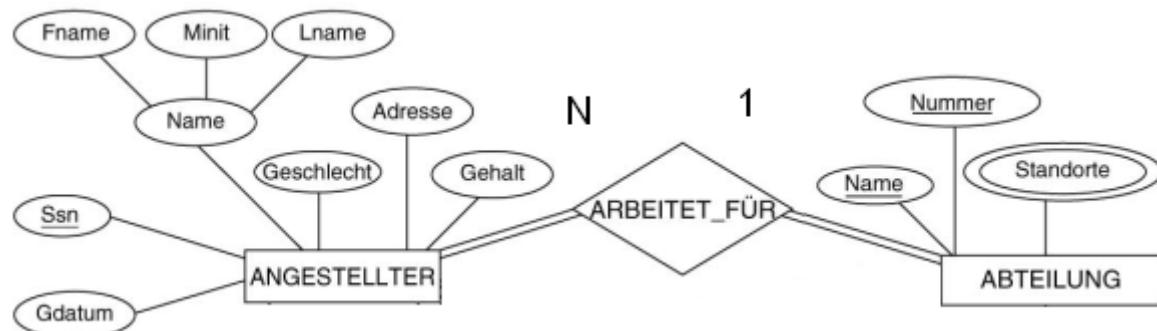
Jede Entität wird zu einer eigenen Tabelle. Die eindeutigen Attribute sind Schlüsselkandidaten und können als Primärschlüssel verwendet werden. Eigene künstliche Schlüssel sind ebenfalls möglich und häufig vorzuziehen.



### 3.1.4 Auflösen der 1 : n-Beziehung

Bei Beziehungen zwischen Entitäten muss sichergestellt sein, dass die beteiligten Ausgangsentitäten nach der Umwandlung noch über das Wissen der jeweils anderen Seite verfügen. Dies erfolgt über das Einfügen von weiteren Spalten in der Tabelle, den. sog. Fremdschlüsseeln (Foreign key).

Um die Forderung nach Redundanzfreiheit, Atomarität, etc. nicht zu verletzen macht es allerdings nur Sinn, dass die n-Seite das neue Attribut aufnimmt. Es wird zu einem Fremdschlüssel und verweist dazu auf den Primärschlüssel der 1-Seite



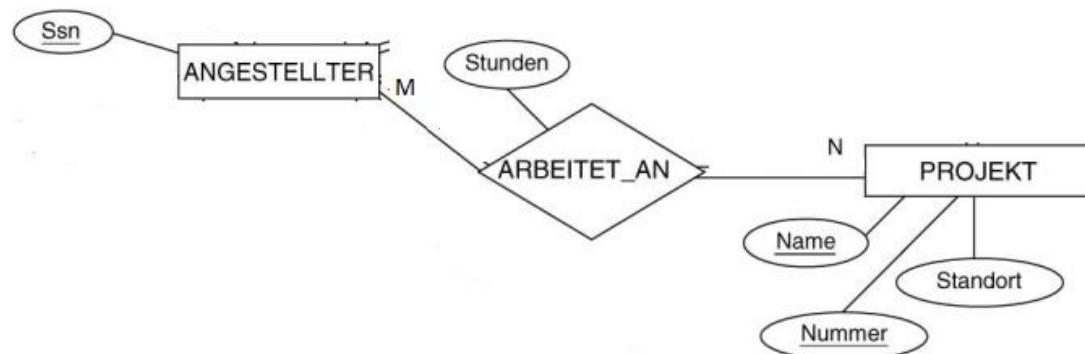
Die Beziehung ARBEITET\_FÜR ist eine 1:n-Beziehung, da ein Angestellter mehrere Abteilungen bearbeiten kann, während eine Abteilung von einem Angestellten bearbeitet wird. Um diese Beziehung in zwei 1:1-Beziehungen aufzulösen, werden neue Attribute hinzugefügt. Die neue Entität ANGESTELLTER\_NUR\_ABTEILUNG enthält die Attributgruppe (Fname, Minit, Lname) unter dem Attribut Name, das Attribut Geschlecht, die Adresse und den Gehalt. Der Primärschlüssel ist die Kombination aus Ssn und Gdatum. Der Fremdschlüssel der ABTEILUNG ist die Kombination aus Nummer und Standorte. Die Beziehung ARBEITET\_FÜR wird nun als 1:1-Beziehung zwischen ANGESTELLTER\_NUR\_ABTEILUNG und ABTEILUNG realisiert. Die Beziehung ist als Doppelpfeil dargestellt, wobei die Kardinalität 1 auf beiden Seiten angegeben ist. Der Primärschlüssel des ANGESTELLTER\_NUR\_ABTEILUNG ist die Kombination aus Ssn und Gdatum. Der Fremdschlüssel der ABTEILUNG ist die Kombination aus Nummer und Standorte.

### 3.1.5 Auflösen der n : m-Beziehung

Da bei der n:m-Beziehung alle Entitäten mehrere Beziehungen zu anderen Entitäten aufweisen können, ist es nicht möglich, innerhalb einer bestehenden Entität durch Fremdschlüsselattribute eine Beziehung herzustellen, da es evtl. einer unbekannten Zahl von Attributen bedürfen könnte.

Deshalb wird eine eigene Zwischentabelle gebildet, deren Primärschlüssel sich aus den Primärschlüsseln der beteiligten Entitäten bildet.

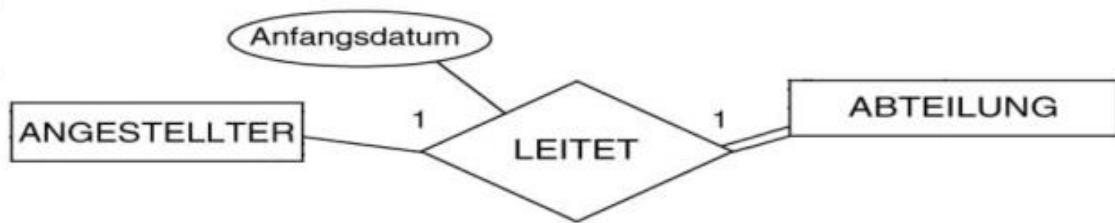
#### Lösung



### 3.1.6 Auflösen der 1 : 1-Beziehung

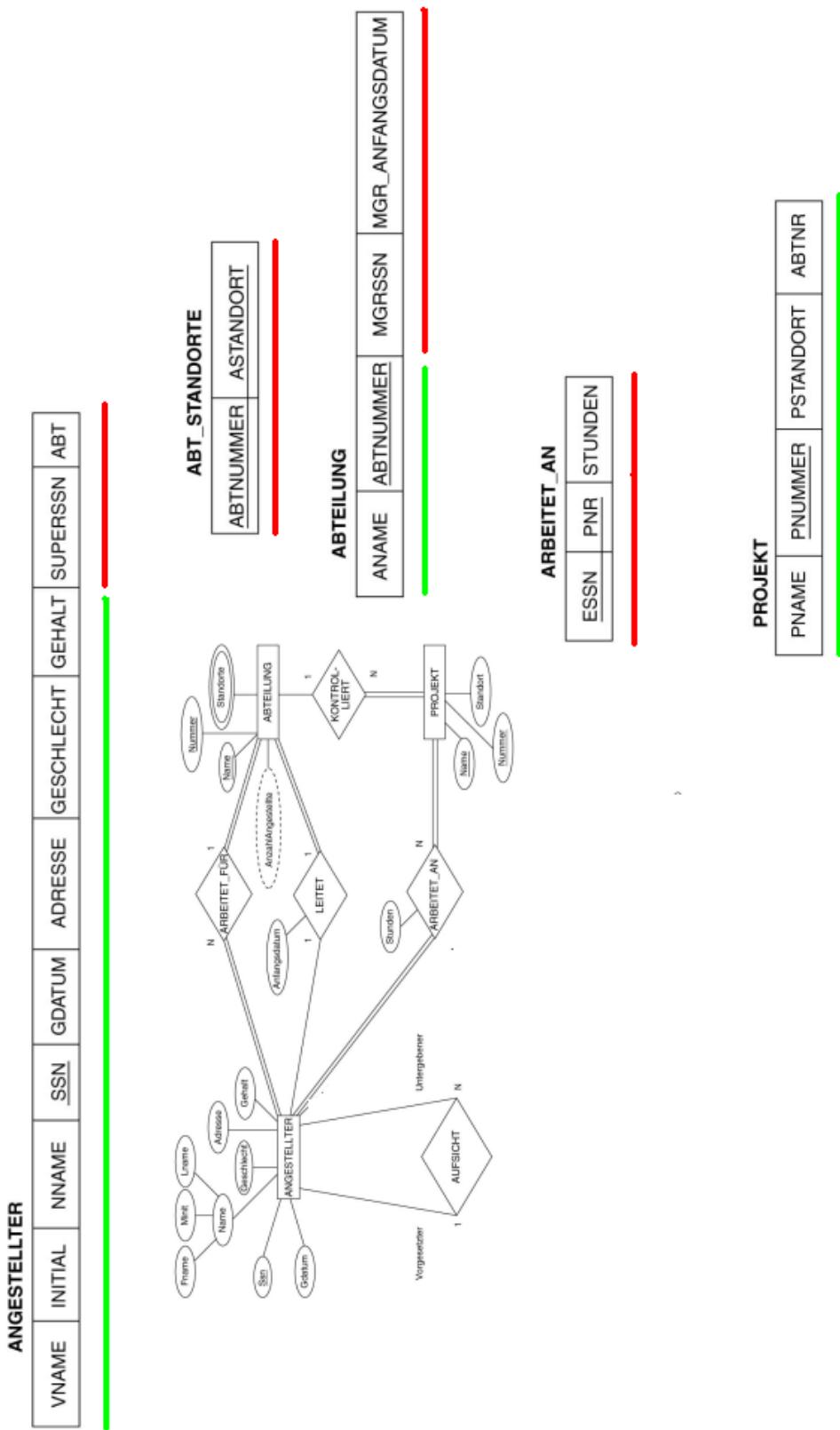
Hier hält jede Entität maximal eine Beziehung zu einer anderen Entität. Deshalb kann eine der beiden Entitäten einen Fremdschlüssel zur anderen Seite führen. Auf die Anzahl möglicher Null-Werte ist dabei zu achten. So hat z.B. **jede Abteilung einen Abteilungsleiter, aber nicht jeder Angestellte ist Abteilungsleiter**. Somit ist es besser, die Information in der Tabelle Abteilung zu führen.

#### Lösung



### 3.1.7 Fazit

Als Ergebnis der Umwandlung ist dann folgendes Tabellenmodell entstanden.



## 3.2 Aufgaben

Wandeln Sie die ER-Modelle aus dem vorhergehenden Kapitel in Tabellenmodelle um.



## 4 Normalformen

Wenn Datenbanken über einen längeren Zeitraum verändert und angepasst werden, kann man erkennen, dass die Konsistenz der Daten, d.h. die "Stimmigkeit der Daten" abnimmt. Ebenso kommt es oft vor, dass Daten redundant gehalten und somit an vielen Stellen gepflegt werden müssen.

Dies versucht man durch das Konzept der **Normalisierung** zu verhindern. Sie soll

- redundante Daten
- Anomalien unter den Daten (Einfüge-, Änderungs-, Löschanomalie)

verhindern.

Empno(PK)	Ename	Hiredate	Manager	Deptno	Dname
1	Smith	10.12.02	34 100	1	Verkauf
2	Meyer	10.04.88	34.?	2	Marketing
3	Jones	31.01.03	34.?	2	Marketing
34	King	24.02.01	100	4	Management
<NULL>	<NULL>	<NULL>	<NULL>	5	Controlling

### Einfüge-Anomalie

Eine Einfüge-Anomalie tritt auf, wenn ein Datensatz gespeichert werden soll und dieser keine oder keine eindeutigen Primärschlüsselwerte aufweist. Das Einfügen in eine Tabelle ist somit nicht möglich. Informationen können nicht gespeichert werden und gehen womöglich verloren. Das kann zum Beispiel der Fall sein, wenn für die Speicherung der Kundendaten zu Verifizierungszwecken die Personalausweisnummer als Primärschlüssel verwendet wird, diese aber leider vom Sachbearbeiter nicht erfasst werden konnte. Der Datensatz des Kunden kann nicht gespeichert werden.

## Änderungs-Anomalie

Man spricht von einer Änderungs-Anomalie, wenn eine Entität redundant in einer oder sogar in mehreren Tabellen enthalten ist und bei einer Aktualisierung nicht alle berücksichtigt werden. Dadurch kommt es zur Inkonsistenz im Datenbestand. Es kann möglicherweise nicht mehr nachvollzogen werden welcher Wert der gültige ist. Dieser Sachverhalt lässt sich gut an einer Auftragstabelle darstellen. Diese speichert neben der Auftragsnummer auch den Namen eines Kunden und dessen Bestellung. Ein Kunde kann mehrere Bestellungen aufgegeben haben, wobei jede Bestellung in einem Datensatz erfasst wird. Wird nun aufgrund eines Schreibfehlers nachträglich der Name des Kunden „Reiher“ in „Reier“ bei einem Datensatz geändert, führt dies zu einem inkonsistenten Datenbestand. Nach der Änderung liegen demnach Aufträge für scheinbar zwei verschiedene Kunden vor und zwar für einen Kunden „Reiher“ und einen Kunden „Reier“.

## Lösch-Anomalie

Enthalten die Datensätze einer Tabelle mehrere unabhängige Informationen, so kann es leicht zu Lösch-Anomalien kommen. Da sich die Daten in einem nicht-normalisierten Zustand befinden, kann durch Löschen eines Datensatzes ein Informationsverlust entstehen. Die Ursache liegt darin, dass in einer Tabelle unterschiedliche Sachverhalte gespeichert werden. Am Beispiel einer nicht-normalisierten Mitarbeitertabelle soll dies kurz skizziert werden. In der Mitarbeitertabelle werden die Personalnummer, der Name und die Abteilung gespeichert. Der Mitarbeiter „Krause“, der als einziger in der Abteilung „Lager“ war, ist aus dem Unternehmen ausgetreten und wird daher aus der Datenbank gelöscht. Da die Abteilung in der gleichen Tabelle gespeichert wird, verschwindet das „Lager“ aus der Datenbank, da „Herr Krause“ ja als einziger dieser Abteilung zugeordnet war.

## Datenbank-Anomalien auflösen

Die beschriebenen Anomalien treten durch ein schlechtes Datenbank-Design auf. Daraus ergibt sich auch die redundante Datenhaltung. Um diese zu vermeiden, müssen die Tabellen einer Datenbank normalisiert werden. Die Normalisierung umfasst in der Praxis drei Stufen und sorgt für eine redundanzfreie und nach Entitäts-Typ getrennte Datenhaltung.

## 4.1 Normalform 1

- **keine Wiederholungsgruppen**
- **Atomarität**

Die 1. NF versucht, das Problem von Wiederholungsgruppen zu verhindern und fordert die Atomarität von Spaltenwerten. Folgendes Beispiel soll dies verdeutlichen.

In einer Tabelle soll der Warenkorb eines Shop-Besuchs abgelegt werden. Jeder Warenkorb wird durch seinen Primärschlüssel identifiziert. Wir gehen hier davon aus, dass der Warenkorb nur von angemeldeten Kunden gefüllt wird. Im Wesentlichen wird hier die Information gespeichert, welcher Artikel wie oft im Warenkorb abgelegt wurde. Das Ergebnis einer Überlegungen sieht wie folgt aus.

Tabelle: artikel			
artikel_id	bezeichnung	warengruppe	[...]
7856	Silberzwiebel	pflanzen	[...]
7863	Tulpenzwiebel	pflanzen	[...]
9010	Schaufel	garten	[...]
9015	Spaten	garten	[...]
3001	Papier (100)	büro	[...]
3005	Tinte (gold)	büro	[...]
3006	Tinte (rot)	büro	[...]
3010	Feder	büro	[...]

Tabelle: warenkorb		
warenkorb_id	artikel	kunde_id
1	7856 30;7863 50;9015 1	12345
2	3006 1;3010 4	12346

### Frage

Betrachten Sie die Inhalte der Tabelle **warenkorb** und diskutieren Sie die möglichen Nachteile. Gibt es auch Vorteile?

Die Werte der Spalte **artikel** in der Tabelle **warenkorb** sind im Grunde Listen, und die Listenelemente bestehen aus zwei Informationen: Artikelnummer und Anzahl. Dies sind zu vermeidende Wiederholungsgruppen: Die Listen sollen aufgelöst und die Teileinformationen in jeweils eigene Spalten überführt werden.

Um die Wiederholungsgruppenfreiheit herzustellen, brauchen wir eine neue Tabelle, die ähnlich der Tabelle **position** die Artikel aufnimmt.

The diagram illustrates the normalization process. It starts with the 'warenkorb' table, which contains two rows (warenkorb\_id 1 and 2). Red arrows point from the 'artikel' column of each row to the 'korbposition' table. This table has five rows: (warenkorb\_id 1, positionsnr 1, artikel 7856 30), (warenkorb\_id 1, positionsnr 2, artikel 7863 50), (warenkorb\_id 1, positionsnr 3, artikel 9015 1), (warenkorb\_id 2, positionsnr 1, artikel 3006 1), and (warenkorb\_id 2, positionsnr 2, artikel 3010 4). Red arrows point from each row in the 'korbposition' table to the 'artikel' table. The 'artikel' table lists individual articles with their descriptions and groups: Silberzwiebel (pflanzen), Tulpenzwiebel (pflanzen), Schaufel (garten), Spaten (garten), Papier (100) (büro), Tinte (gold) (büro), Tinte (rot) (büro), and Feder (büro).

Tabelle: warenkorb			
warenkorb_id	kunde_id		
1	12345		
2	12346		
Tabelle: korbposition			
warenkorb_id	positionsnr	artikel	
1	1	7856 30	
1	2	7863 50	
1	3	9015 1	
2	1	3006 1	
2	2	3010 4	
artikel_id	bezeichnung	warengruppe	[...]
artikel_id	bezeichnung	warengruppe	[...]
7856	Silberzwiebel	pflanzen	[...]
7863	Tulpenzwiebel	pflanzen	[...]
9010	Schaufel	garten	[...]
9015	Spaten	garten	[...]
3001	Papier (100)	büro	[...]
3005	Tinte (gold)	büro	[...]
3006	Tinte (rot)	büro	[...]
3010	Feder	büro	[...]

## Frage

Warum ist die Spalte artikel in der Tabelle **korbposition** keine Wiederholungsgruppe?

Was danach noch störend ins Auge fällt, ist die Spalte artikel in der Tabelle korbposition. Die Artikelnummer und die Anzahl sind in der gleichen Spalte abgelegt, was eine Auswertung und Veränderung der Daten erheblich erschwert. Die Spalte ist somit nicht atomar und sie kann noch weiter zerlegt werden. Zuletzt haben die Tabellen folgenden Aufbau.

Tabelle: warenkorb

warenkorb_id	kunde_id
1	12345
2	12346

Tabelle: korbposition

warenkorb_id	positionsnr	menge	artikel
1	1	30	7856
1	2	50	7863
1	3	1	9015
2	1	1	3006
2	2	4	3010

Tabelle: artikel

artikel_id	bezeichnung	warengruppe	[...]
7856	Silberzwiebel	pflanzen	[...]
7863	Tulpenzwiebel	pflanzen	[...]
9010	Schaufel	garten	[...]
9015	Spaten	garten	[...]
3001	Papier (100)	büro	[...]
3005	Tinte (gold)	büro	[...]
3006	Tinte (rot)	büro	[...]
3010	Feder	büro	[...]

## Erste Normalform

Eine Tabelle ist dann in der 1. Normalform, wenn sie atomar und wiederholungsgruppenfrei ist. Eine Datenbank ist dann in der 1. Normalform, wenn alle Tabellen in der 1. Normalform sind.

## 4.2 Normalform 2

### Volle funktionale Abhangigkeit der Nichtschlsselattribute

Wir betrachten eine Tabelle `tblPositionen`, die die Bestellpositionen eines Auftrags zusammenfasst. Ein Auftrag kann mehrere Bestellpositionen besitzen; pro Bestellposition wird ein Artikel definiert. Der Primarschlssel ist zusammengesetzt aus den Spalten `AuftragNr` und `TeileNr`.

TblPositionen					
A_Nr	T_Nr	T_Bezeichnung	A_Menge	P_Preis	
99-8419	12088	P400 komplett	3	999,00	
99-8420	11000	PC-Tisch	2	399,00	
99-8420	11002	Monitor 19``	1	599,00	
99-8420	12089	P500 komplett	10	1199,00	
99-8421	12089	P500 komplett	5	1199,00	
99-8421	23044	Tischfax	2	489,00	
99-8422	33002	Toner HP	25	189,00	
99-8422	11000	PC-Tisch	10	399,00	
99-8422	11002	Monitor 19"	5	599,00	

Die 2. NF fordert, dass die Werte alle Nichtschlsselspalten voll funktional vom Gesamt-schlssel abhangig sind. Dies ist in der obigen Tabelle nicht immer der Fall, da der Artikel-name lediglich abhangig von der TeileNr ist. Diese Spalten sind aus der Tabelle herauszulosen und in eigenen Tabellen zu speichern.

TblPositionen				tblArtikel		
A_Nr	T_Nr	A_Menge		T_Nr	T_Bezeichnung	P_Preis
99-8419	12088	3		12088	P400 komplett	999,00
99-8420	11000	2		11000	PC-Tisch	399,00
99-8420	11002	1		11002	Monitor 19"	599,00
99-8420	12089	10		12089	P500 komplett	1199,00
99-8421	12089	5		12089	P500 komplett	1199,00
99-8421	23044	2		23044	Tischfax	489,00
99-8422	33002	25		33002	Toner HP	189,00
99-8422	11000	10				
99-8422	11002	5				

## 4.3 Normalform 3

### Keine transitive Abhangigkeit zwischen Nichtschlusselpalten

Ermitteln Sie in untenstehender Grafik den Zusammenhang zwischen Bankleitzahl (blz) und dem Banknamen sowie ebenso zwischen der Bankleitzahl, der Kontonummer und der IBAN.

Tabelle: bankverbindung

kunde_id	bankverbindung_nr	blz	kontonr	bankname	iban
12345	1	50041597	1234506789	Sparkasse Aulenland	DEXX500415971234506789
12345	2	50287667	5432109876	Volksbank Eriador	DEXX502876675432109876
12346	1	50287667	5432109880	Volksbank Eriador	DEXX502876675432109890

Wenn Nichtschlusselpalten aus anderen Nichtschlusselpalten herleitbar sind, bedeutet dies in der Regel, dass Informationen redundant in der Tabelle gehalten werden. Hier werden beispielsweise die Banknamen mehrfach genannt. Dies verbraucht nicht nur Speicherplatz, sondern macht eine nderung der Banknamen teuer, da diese in vielen Zeilen durchgefrt werden mssen.

Tabelle: bankverbindung

kunde_id	bankverbindung_nr	kontonr	iban	blz
12345	1	1234506789	[...]	50041597
12345	2	5432109876	[...]	50287667
12346	1	5432109880	[...]	50287667

Tabelle: bank

blz	bankname	lkz
50041597	Sparkasse Aulenland	DE
50287667	Volksbank Eriador	DE

Transitive Informationen begegnen Ihnen relativ oft. Der Kontoinhaber ergibt sich aus der Kontonummer, der Ortsname aus der Postleitzahl, der Rabatt aus der Kundenart etc.

### Aufgabe:

- Finden Sie Beispiele fr Einfe-, nderungs- und Loschanomalien
- Erltern Sie den Begriff Wiederholungsgruppe.
- Was bedeutet die Forderung nach Atomaritt.
- Diskutieren Sie die Thematik Zusammengesetzter Schlussel vs . Knstlicher Schlussel

## 4.4 Referentielle Integrität

Im Bereich der relationalen Datenbanken wird die referentielle Integrität dazu verwendet die Konsistenz und die Integrität der Daten sicherzustellen. Dazu werden Regeln aufgestellt, wie und unter welchen Bedingungen ein Datensatz in die Datenbank eingetragen wird.

Bei der referentiellen Integrität können Datensätze die einen Fremdschlüssel aufweisen nur dann gespeichert werden, wenn der Wert des Fremdschlüssels einmalig in der referenzierten Tabelle existiert. Im Falle, dass ein referenziertes Wert nicht vorhanden ist, kann der Datensatz nicht gespeichert werden.

### Warum wird die Referentielle Integrität benötigt?

Eine Datenbank kann schnell in einen inkonsistenten Zustand geraten. Im ungünstigsten Fall liegt eine nicht-normalisierte Datenbank vor, die starke Redundanzen aufweist. Dabei können Anomalien im Datenbestand auftreten, die verschiedene Formen annehmen. Man spricht hier von Einfüge-, Lösch- und Änderungsanomalien. Tritt eine oder mehrere dieser Anomalien auf, kann das zur Verfälschung oder Löschung von Informationen führen.

### Weitergabe

Während die RI grundsätzlich vor inkonsistenten Datenaktionen schützt, bieten viele Datenbanksysteme Zusatzfunktionen an, die bei Updates von Master-Datensätzen nützlich sein können:

#### Änderungsweitergabe (ÄW)

Wenn der eindeutige Schlüssel eines Datensatzes geändert wird, kann das DBMS die Fremdschlüssel in allen abhängigen Datensätzen anpassen – anstatt die Änderung abzulehnen. Änderungsweitergabe wird insbesondere dann benutzt, wenn natürliche Schlüssel (die sich ändern können; Familienname bei Heirat) verwendet werden; denn künstliche Schlüssel sind i. d. R. unveränderlich und eine Änderungsweitergabe nicht erforderlich.

#### Löschweitergabe (LW)

In bestimmten Fällen ergibt es einen Sinn, abhängige Datensätze bei Löschung des Masterdatensatzes mitzulöschen.

```
create table ...
    ...
foreign key (id) references tabelle.pk
    on delete cascade
    on update restrict
```

Diese Funktionen können in der RI-Spezifikation optional gesetzt und (je nach DBMS) durch zusätzliche Bedingungen erweitert/präzisiert werden.

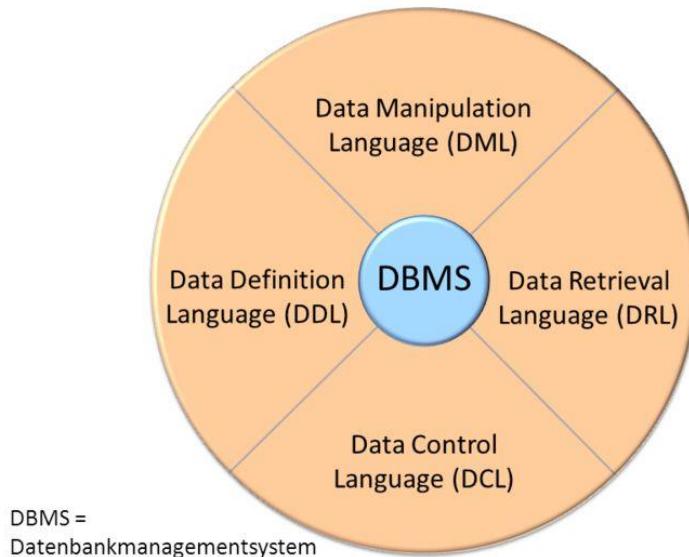
# 5 SQL -DDL

SQL ist eine Sprache zur Bearbeitung und Auswertung von relationalen Datenbanken. Sie umfasst drei Bereiche:

**SQL Standard.**

**Teilsprachen.**

## Structured Query Language (SQL)



- Data Definition Language (DDL): Befehlssatz zum Anlegen, Ändern und Löschen von Datenbanken, Tabellen usw. und ihren Strukturen.
- Data Manipulation Language (DML): Befehlssatz zum Einfügen, Ändern, Löschen und Auslesen von Daten aus den Tabellen.
- Data Control Language (DCL): Befehlssatz zur Administration von Datenbanken

Anders als bei imperativen Programmiersprachen wie C#, C++, Java oder Pascal wird durch die Befehle nicht die Art und Weise bestimmt, wie man ein Ergebnis erhält; es wird kein Algorithmus implementiert. Vielmehr sagt man, was man haben möchte, und der Datenbanksystem ermittelt das Ergebnis. Solche Arten von Programmiersprachen nennt man *deklarativ*.

Obwohl es viele SQL-Dialekte gibt, ist der offizielle SQL-Standard in vielen Systemen implementiert und garantiert eine Wiederverwendbarkeit oder Übertragbarkeit der Befehle.

1986 wurde der erste SQL-Standard vom ANSI verabschiedet, der 1987 von der ISO ratifiziert wurde. 1992 wurde der Standard überarbeitet und als SQL-92 (oder auch SQL2) veröffentlicht. Alle aktuellen Datenbanksysteme halten sich im Wesentlichen an diese Standardversion. Die Version SQL:1999 (ISO/IEC 9075:1999, auch SQL3 genannt) ist noch nicht in allen Datenbanksystemen implementiert. Das gilt auch für die nächste Version SQL:2003. Der aktuelle Standard wurde 2008 unter SQL:2008 verabschiedet.

## 5.1 Anlegen/Löschen einer Datenbank

Wir gehen davon aus, dass die Datenbank komplett neu erstellt werden soll. Der Befehl zum Anlegen einer Datenbank ist CREATE SCHEMA. Damit werden allerdings noch keine Tabellen angelegt, sondern nur die diese umfassende Datenbank.

Bibliothek

```
--SQL 92:  
CREATE SCHEMA datenbankname  
    [ DEFAULT CHARACTER SET zeichensatz]  
  
--Mysql:  
CREATE { DATABASE | SCHEMA } [ IF NOT EXISTS ] datenbankname  
    [ [ DEFAULT ] CHARACTER SET zeichensatz]  
    [ COLLATE sortierung]  
;
```

### 5.1.1 Zuweisen eines CharacterSets

Was ist ein Zeichensatz? Auf dem Computer werden Buchstaben, Ziffern, Satz- und Sonderzeichen durch Zahlen kodiert. So ist beispielsweise der Buchstabe A im ASCII-Zeichensatz die Zahl 0x41 und a die Zahl 0x61. Da für die Kodierung des ASCII nur ein Byte (= 8 Bit) zur Verfügung steht, können nur 256 verschiedene Zahlen zur Kodierung verwendet werden. Die ersten 128 sind im Wesentlichen die Steuerzeichen (wie z.B. der Zeilenumbruch), das Leerzeichen, die lateinischen Buchstaben, die Ziffern 0 – 9, Satz- und einfache Sonderzeichen. Die restlichen 128 wurden mehr oder weniger willkürlich dazu verwendet, Umlaute oder andere sprachspezifische Sonderzeichen abzubilden. Und hier fing das Unglück an. Fast jeder Computer- oder Betriebssystemhersteller hat da sein eigenes Süppchen gekocht. So ist beispielsweise das Zeichen Ü im Zeichensatz ISO/IEC 8859-1 mit der Zahl 0xDC kodiert und in Codepage 850 mit 0x9A. Wird nun ein Text unter Windows erfasst, wird das Ü als 0xDC in die Datei geschrieben. Öffnet man nun diese Datei mit einem COMMAND-Editor wie EDIT, so erscheint aber ein anderes Zeichen und umgekehrt. Dieses Problem und die Beschränkung auf 256 Zeichen, was die Darstellung z.B. ostasiatischer Schriften unmöglich macht, haben dazu geführt, dass man eine neue, leicht erweiterbare Kodierung von Schriftzeichen baute. Unicode ward geboren! Unicode selbst liegt in verschiedenen Formatierungen vor. Derzeit gerne verwendet werden utf8, utf16 und utf32. Welche Zeichensätze von Ihrem Server unterstützt werden, können Sie leicht mit SHOW CHARACTER SET herausfinden.

## 5.1.2 Sortierreihenfolge

Für jede Sprache gibt selbst bei gleichen Zeichensätzen oft mehrere Arten, die Texte wie z.B. für eine Namensliste zu sortieren. In MySQL wird die Sortierreihenfolge über die Option **COLLATE** im CREATE SCHEMA festgelegt.

Die verfügbaren Sortierungen lassen sich leicht mit SHOW COLLATION anzeigen, wobei schnell deutlich wird, dass es mehrere Sortierreihenfolgen für einen Zeichensatz geben kann. Betrachten wir die Sortierreihenfolgen für den Zeichensatz cp850:

```
mysql> SHOW COLLATION LIKE 'cp850%';
+-----+-----+-----+-----+-----+
+          | Charset | Id | Default | Compiled | Sortlen |
+-----+-----+-----+-----+-----+
+          | cp850_general_ci | cp850 | 4 | Yes      | Yes        | 1       |
| cp850_bin           | cp850 | 80 |          | Yes        | 1       |
+-----+-----+-----+-----+-----+
+          2 rows in set (0.00 sec)
```

Bei cp850\_general\_ci wird nicht und bei cp850\_bin wird zwischen Groß- und Kleinschreibung unterschieden.

## 5.1.3 Löschen einer Datenbank

Das Löschen einer Datenbank erfolgt analog zum Erstellen mit dem Befehl **DROP SCHEMA**

```
--SQL92
DROP SCHEMA datenbankname
[ CASCADE | RESTRICT ]
;
--MySQL
DROP { DATABASE | SCHEMA } [ IF EXISTS ] datenbankname;
```

Mit Hilfe von CASCADE bzw. RESTRICT werden in der Datenbank existierende Tabellen entweder gleich mitgelöscht bzw. das Löschen der Datenbank wird verhindert, solange diese noch Tabellen aufweist.

## 5.2 Anlegen einer Tabelle

```
CREATE TABLE tabellenname (
    spaltenspezifikation
    [, spaltenspezifikation]*
    [, PRIMARY KEY(spaltenliste) ]
) [tabellenoptionen];
```

Der erste Teil des Befehls ist selbsterklärend. Danach kommt der Tabellenname, den wir der Namenskonvention entsprechend klein schreiben. Was ist aber eine *spaltenspezifikation*? Eine Spaltenspezifikation besteht aus drei Teilen:

1. Spaltenname: Er wird klein geschrieben und ergibt sich aus dem ER-Modell.
2. Datentyp: Dieser legt fest, was für eine Art von Information in der Spalte verwaltet und wie diese kodiert wird. Mögliche Datentypen finden Sie in Abschnitt 25.1 auf Seite 371.
3. Zusätze: Mit diesen kann man eine Spalte ausführlicher bestimmen. Eine Liste möglicher Zusätze finden Sie weiter unten.

Das aus der Notation für reguläre Ausdrücke entnommene Sternchen \* hinter der optionalen zweiten Spaltenspezifikation bedeutet: eine beliebige Anzahl viele, also auch 0.

```
use database artikel;
CREATE TABLE adresse (
    adresse_id INT UNSIGNED AUTO_INCREMENT,
    strasse VARCHAR(255),
    hnr VARCHAR(255),
    lkz CHAR(2),
    plz CHAR(5),
    ort VARCHAR(255),
    deleted TINYINT UNSIGNED NOT NULL DEFAULT 0,
    PRIMARY KEY(adresse_id)
);
```

## Beispiele

```

drop database if exists employee; (1)
create database employee;

use employee; (2)

create table department (3)
(
    departmentID int not null auto_increment primary key,
    name varchar(20)
) type=InnoDB;

create table employee (4)
(
    employeeID int not null auto_increment primary key,
    name varchar(80),
    job varchar(15),
    departmentID int not null
        references department(departmentID)
) type=InnoDB;

create table employeeSkills (5)
(
    employeeID int not null references employee(employeeID),
    skill varchar(15) not null,
    primary key (employeeID, skill)
) type=InnoDB;

create table client
(
    clientID int not null auto_increment primary key,
    name varchar(40),
    address varchar(100),
    contactPerson varchar(80),
    contactNumber char(12)
) type=InnoDB;

create table assignment
(
    clientID int not null references client(clientID),
    employeeID int not null references employee(employeeID),
    workdate date not null,
    hours float,
    primary key (clientID, employeeID, workdate)
) type=InnoDB;

```

1. Diese Anweisung stellt fest, ob die DB schon existiert und löscht sie gegebenfalls. Die Anweisung ist u.U. mit Vorsicht zu genießen.
2. Mit Hilfe von CREATE wird die Datenbank erstellt und durch USE zur aktuellen DB erklärt.
3. Die Tabelle verfügt über 2 Spalten, departmentID als Primärschlüssel und name als Abteilungsnamen. Als Tabellentyp ist InnoDB angegeben.
  - departmentID: Datentyp ist int (Integer)
  - not null; die Spalte muss in jeder Zeile einen Wert haben
  - auto-increment; der Wert wird von MySQL automatisch hochgezählt
  - primary key: Die Spalte ist der Primärschlüssel der Tabelle
  - Name: Die Spalte kann maximal 20 alphanumerische Zeichen aufnehmen (varchar(20))
  - Die Spalten werden in Form einer kommaseparierten Liste geführt. Üblicherweise wird immer eine Objektbeschreibung je Zeile geschrieben. Der SQL-Interpreter führt die Zeile erst aus, nachdem er ein Semikolon (;) gefunden hat.
4. Mit Hilfe des Schlüsselwortes REFERENCES weist man die Spalte departmentID als Fremdschlüssel aus. Sie verweist auf den Primärschlüssel der Tabelle department..
5. Ein zusammengesetzter Primärschlüssel wird gebildet, indem in der Klammer auf mehrere Felder verwiesen wird. Die einzelnen Spalten werden durch Komma getrennt

## 5.2.1 Datentypen

Datentypen.

## 5.2.2 Constraints

Constraints definieren ganz allgemein gesprochen *Einschränkungen*, denen ein relationales Datenmodell entsprechen muss. Diese Einschränkungen können sein:

### 5.2.2.1 NOT NULL constraints

Die Einschränkung bedeutet, dass der Wert einer Spalte kein NULL-Wert sein darf.

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

### 5.2.2.2 CHECK constraint

Damit wird ausgedrückt, dass Werte in Spalten gewisse Bedingungen einzuhalten haben, z.B. bestimmte Größenbereiche

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
); --column check constraint (benannt positive_price)
```

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
); -- letzte zwei Zeilen sind table check constraints
```

Der Constraint wird erfüllt, wenn die Bedingung true ist oder einen NULL-Wert annimmt.

**Der check-Constraint wird von MySQL zwar auf Syntaxfehler hin gepräst, jedoch nicht weiter umgesetzt :-).**

### 5.2.2.3 UNIQUE constraint

Dieser Constraint fordert, dass für eine Spalte(n) innerhalb einer Tabelle eine Wert nur einmal vorkommt. NULL-Werte sind dennoch über mehrere Zeilen hinweg erlaubt. Er wird durch folgende Befehle abgebildet.

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
); -- column constraint
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
); -- table constraint

CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c) -- eindeutig über mehrere Spalten
);
```

#### 5.2.2.4 PRIMARY/FOREIGN KEY Constraint

Ein PRIMARY KEY ist technisch gesehen ein UNIQUE constraint, der einen INDEX führt. Im Gegensatz zum UNIQUE Constraint dürfen die jeweiligen Spalten aber keinen NULL-Wert besitzen.

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

Ein FOREIGN KEY ist eine Einschränkung, dass sich der Wert einer Spalte an einem UNIQUE Wert einer anderen Tabelle orientieren muss. Ein FOREIGN KEY kann NULL-Werte besitzen, die gegenüberliegende Seite muss ein PRIMARY KEY oder ein UNIQUE constraint sein.

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
-- Zusammengesetzter Fremdschlüssel mit Namen fk_myFKey
-- Gut zum späteren Löschen des FK.
CREATE TABLE t1 (
    a integer PRIMARY KEY,
    b integer,
    c integer,
    CONSTRAINT fk_myFKey FOREIGN KEY (b, c)
        REFERENCES other_table (c1, c2)
);
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

-- Lösch/Änderungsweitergabe verhindern oder erlauben
```

```
CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders

    ON DELETE CASCADE ON UPDATE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
) ;
```

In MySQL muss als Tabellentyp InnoDB angegeben sein. Weiterhin muss vorher ein Index auf die Fremdschlüsselalte vergeben worden sein.

```
CREATE TABLE product (
    category INT NOT NULL, id INT NOT NULL,
    price DECIMAL,
    PRIMARY KEY(category, id)
) TYPE=INNODB;

CREATE TABLE customer (
    id INT NOT NULL,
    PRIMARY KEY (id)
) TYPE=INNODB;

CREATE TABLE product_order (
    no INT NOT NULL AUTO_INCREMENT,
    product_category INT NOT NULL,
    product_id INT NOT NULL,
    customer_id INT NOT NULL,
    PRIMARY KEY(no),
    INDEX (product_category, product_id),
    FOREIGN KEY (product_category, product_id)
    REFERENCES product(category, id)
        ON UPDATE CASCADE ON DELETE RESTRICT,
    INDEX (customer_id),
    FOREIGN KEY (customer_id) REFERENCES customer(id)
) TYPE=INNODB;
```

### 5.2.3 Ändern/Löschen von Datenstrukturen

Mit Hilfe des Befehl **ALTER** kann die Struktur einer bestehenden Tabelle verändert werden. Dazu wird dem Statement je nach Bedarf eine drop, add, change, modify - Klausel hinzugefügt.

Der grundlegende Aufbau sieht wie folgt aus.

```
ALTER [IGNORE] TABLE tbl_name alter_specification [, alter_specification ...]

alter_specification:
    ADD [COLUMN] create_definition [FIRST | AFTER column_name]
    | ADD [COLUMN] (create_definition, create_definition,...)
    | ADD INDEX [index_name] (index_col_name,...)
    | ADD PRIMARY KEY (index_col_name,...)
    | ADD UNIQUE [index_name] (index_col_name,...)
    | ADD FULLTEXT [index_name] (index_col_name,...)
    | ADD [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_col_name, ...)
        [reference_definition]
    | ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
    | CHANGE [COLUMN] old_col_name create_definition
        [FIRST | AFTER column_name]
    | MODIFY [COLUMN] create_definition [FIRST | AFTER column_name]
    | DROP [COLUMN] col_name
    | DROP PRIMARY KEY
    | DROP INDEX index_name
    | DISABLE KEYS
    | ENABLE KEYS
    | RENAME [TO] new_tbl_name
    | ORDER BY col
    | table_options
```

Das folgende Beispiel zeigt den Umgang mit dem ALTER TABLE-Statement.

```
C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.0.1-alpha-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> create database testAlter;
mysql> use testAlter;
      Wir beginnen mit dem Erzeugen einer Demodatenbank

mysql> -- Wir erzeugen eine Tabelle t1
      CREATE TABLE t1 (a INTEGER,b CHAR(10));

      -- Wir ändern den Tabellennamen von t1 in t
mysql> ALTER TABLE t1 RENAME t2;

      --Wir ändern den Spaltentyp von a in TINYINT NOT NULL und
ändern
      --den Spaltentyp von b nach CHAR(20) und geben der Spalte b
den
      -- Namen c .
mysql> ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c
CHAR(20);

      --Wir fügen eine neue Spalte d mit Datentyp TIMESTAMP hinzu.
mysql> ALTER TABLE t2 ADD d TIMESTAMP;

      --Wir fügen einen Index auf die Spalte d hinzu und machen aus
      --der Spalte a einen Primärschlüssel.
mysql> ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);

      --Wir löschen die Spalte c
mysql> ALTER TABLE t2 DROP COLUMN c;

      --Wir fügen eine neue Spalte c mit dem Datentyp INTEGER hinzu.
      -- Der Wert soll sich automatisch hochzählen.
mysql> ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,
      ADD INDEX (c);
```

## 5.2.4 Einfügen/Ändern/Löschen von Daten

### 5.2.4.1 INSERT

Mit Hilfe des Befehls INSERT wird ein neuer Datensatz in eine Tabelle eingefügt. Der INSERT-Befehl ist in verschiedenen Formen verfügbar.

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
INTO tbl_name [(Feld1, Feld2, ...)]
VALUES ({expr | DEFAULT}, ...), (...), ...
[ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Nach dem Nennen der Tabelle erfolgt die Definition der Feldnamen; mit Hilfe des Schlüsselwortes **VALUES** werden dann die jeweiligen Werte übergeben.

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
INTO tbl_name
SET col_name={expr | DEFAULT}, ...
[ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Das Einfügen der Werte erfolgt explizit durch eine **Feld=Wert-Zuweisung**

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
INTO tbl_name [(col_name, ...)]
SELECT ...
```

Die zum Füllen der Felder erforderlichen Werte kommen aus einem SELECT Statement

### 5.2.4.2 REPLACE

Mit Hilfe des Befehls REPLACE wird ebenfalls ein neuer Datensatz in eine Tabelle eingefügt. Wird jedoch ein Wert für einen Primärschlüssel oder einen UNIQUE KEY mit übergeben, so wird der alte Datensatz vor dem Einfügen des neuen Datensatzes gelöscht. Verschiedene Formen sind möglich.

```
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name [(col_name, ...)]
VALUES ({expr | DEFAULT}, ...), (...), ...

REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name
SET col_name={expr | DEFAULT}, ...

REPLACE [LOW_PRIORITY | DELAYED]
[INTO] tbl_name [(col_name, ...)]
SELECT ...
```

### 5.2.4.3 DELETE, UPDATE

Mit Hilfe des **DELETE**-Befehls werden Datensätze in einer Tabelle gelöscht. Es ist dabei zu beachten, dass **ohne einen WHERE-Teil alle Datensätze in einer Tabelle gelöscht** werden.

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name  
[WHERE where_definition]  
[ORDER BY ...]  
[LIMIT row_count]
```

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]  
tbl_name[.*] [, tbl_name[.*] ...]  
FROM table_references  
[WHERE where_definition]
```

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]  
FROM tbl_name[.*] [, tbl_name[.*] ...]  
USING table_references  
[WHERE where_definition]
```

#### 5.2.4.4 Aufgaben zu UPDATE, INSERT, DELETE

##### Lösung Link.

1. Folgende Tabelle ist gegeben

```
mysql> DESCRIBE twounique; SELECT * FROM twounique;

+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id1   | tinyint(3) unsigned |      | PRI | 0       |       |
| id2   | tinyint(3) unsigned |      | UNI | 0       |       |
+-----+-----+-----+-----+-----+
+-----+
| id1 | id2 |
+-----+
| 1   | 2   |
| 3   | 4   |
| 5   | 6   |
+-----+
```

Sie führen folgende beiden Befehle aus. Was ist anschließend der Inhalt der Tabelle?

```
mysql> REPLACE INTO twounique VALUES (2,2);
mysql> REPLACE INTO twounique VALUES (2,6);
```

2. Wie fängt man mehrere Datensätze mit Hilfe eines einzigen INSERT-Statements hinzu
3. INSERT unterstützt den sog. IGNORE Modifier, REPLACE jedoch nicht. Warum ist das so ?
4. Mit welchem Statement kann man eine Tabelle komplett leeren ?
5. Mit welchem Statement kann man eine Tabelle teilweise leeren ?
6. Welcher Unterschied besteht in der Art und Weise, wie MySQL Fehler behandelt für ein single-row oder multiple-row-statement, wenn NULL-Werte in ein NOT NULL-Feld eingefügt werden?
7. Was für Gründe kann es geben, wenn ein UPDATE-Statement keine Auswirkungen hat, d.h. keinen einzigen Wert ändert.
8. Warum ist die Zahl der betroffenen Zeilen im folgenden UPDATE-Statement 0, obwohl die WHERE-Klausel auf 5 Zeilen zutrifft ? Warum ist die Anzahl der zutreffenden Zeilen 5 und nicht 10?

```

mysql> SELECT pid, grade FROM personnel;
+----+-----+
| pid | grade |
+----+-----+
| 1   |     1 |
| 2   |     2 |
| 3   |   NULL |
| 4   |   NULL |
| 5   |   NULL |
| 6   |     1 |
| 7   |     1 |
| 8   |     1 |
| 9   |   NULL |
| 10  |   NULL |
| 11  |   NULL |
| 12  |     1 |
| 13  |   NULL |
+----+-----+
13 rows in set

mysql> UPDATE personnel SET grade = 1 WHERE grade != 2;
Query OK, 0 rows affected (0.00 sec)
rows matched: 5  Changed: 0  Warnings: 0

```

9. Ist das folgende Statement WAHR oder FALSCH?

Um zu verhindern, dass UPDATE-Statements alle Zeilen einer Tabelle ändern würden, kann man mysql mit der **-safe-updates-Option** starten

10. Folgende Tabelle ist gegeben.

```

mysql> SELECT * FROM personnel;
+----+-----+-----+
| pid | unit | grade |
+----+-----+-----+
| 1   |    42 |     1 |
| 2   |    42 |     2 |
| 3   |    42 |   NULL |
| 4   |    42 |   NULL |
| 5   |    42 |   NULL |
| 6   |    23 |     1 |
| 7   |    23 |     1 |
| 8   |    23 |     1 |
| 9   |    23 |   NULL |
| 10  |    42 |   NULL |
| 11  |    23 |   NULL |
| 12  |    23 |     1 |
| 13  |    42 |   NULL |
+----+-----+-----+

```

Mit welchem einzigen UPDATE-Statement würde man alle Zeilen, die im Feld grade keinen Wert besitzen, auf 3 ändern.

11. Beziehen Sie sich auf die vorhergehende Tabelle. Welches REPLACE-Statement würden Sie benutzen, um das grade-Feld auf 4 und das unit-Feld auf 45 für alle Zeilen zu ändern, wo das pid-Feld den Wert 10 hat.

12. Die Tabelle personnel hat den folgenden Aufbau:

```
mysql> DESCRIBE personnel; SELECT * FROM personnel;

+-----+-----+-----+-----+-----+
+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+
+
| pid   | smallint(5) unsigned |      | PRI | NULL    | auto_incremnt |
|
| unit  | tinyint(3) unsigned | YES  |      | NULL    |                |
| grade | tinyint(3) unsigned | YES  |      | NULL    |                |
+-----+-----+-----+-----+
+
+-----+-----+-----+
| pid | unit | grade |
+-----+-----+-----+
| 1   | 42   | 1     |
| 2   | 42   | 2     |
| 3   | 42   | 3     |
| 4   | 42   | 3     |
| 5   | 42   | 3     |
| 6   | 23   | 1     |
| 7   | 23   | 1     |
| 8   | 23   | 1     |
| 9   | 23   | 3     |
| 10  | 42   | 3     |
| 11  | 23   | 3     |
| 12  | 23   | 1     |
| 13  | 42   | 3     |
+-----+-----+-----+
```

Welches UPDATE-Statement benutzen Sie, um die Werte des Feldes grade mit 1000 zu multiplizieren. Welche Werte würde das Statement erzeugen.

13. In the table personnel, the unit numbers were interchanged for some reason. Unit 23 is supposed to be 42, and 42 is supposed to be 23. What statement would you use to resolve this problem? Currently the table looks like this:

```
mysql> SELECT * FROM personnel;

+----+----+----+
| pid | unit | grade |
+----+----+----+
| 1   | 42   | 255  |
| 2   | 42   | 255  |
| 3   | 42   | 255  |
| 4   | 42   | 255  |
| 5   | 42   | 255  |
| 6   | 23   | 255  |
| 7   | 23   | 255  |
| 8   | 23   | 255  |
| 9   | 23   | 255  |
| 10  | 42   | 255  |
| 11  | 23   | 255  |
| 12  | 23   | 255  |
| 13  | 42   | 255  |
```

14. The table petnames contains the following data:

```
mysql> SELECT * FROM petnames;

+----+
| name |
+----+
| Lucy |
| Macie|
| Myra |
| Cheep |
| Lucy |
| Myra |
| Cheep |
| Macie |
| Pablo |
| Stefan|
+----+
```

Assume that you issue the following statement:

```
UPDATE petnames SET name = CONCAT(name, 1) ORDER BY name LIMIT 1;
```

What will the table's contents be after the UPDATE?

15. Will the following statement delete all rows from the table mytable ?

```
TRUNCATE TABLE mytable;
```

16. Will the following statement delete all rows from the table mytable ?

```
DELETE FROM mytable;
```



## 6 SELECT

Eine Datenbank enthält eine Vielzahl von verschiedenen Daten. Abfragen dienen dazu, bestimmte Daten aus der Datenbank auszugeben. Dabei kann die Ergebnismenge entsprechend den Anforderungen eingegrenzt und genauer gesteuert werden. Der SQL-Befehl zum Abfragen von Daten lautet **SELECT** und besteht aus folgenden Bestandteilen ("Klauseln").

**select\_einfach**  
**order by.**  
**LIKE.**  
**group by.**  
**having.**  
**join\_where.**

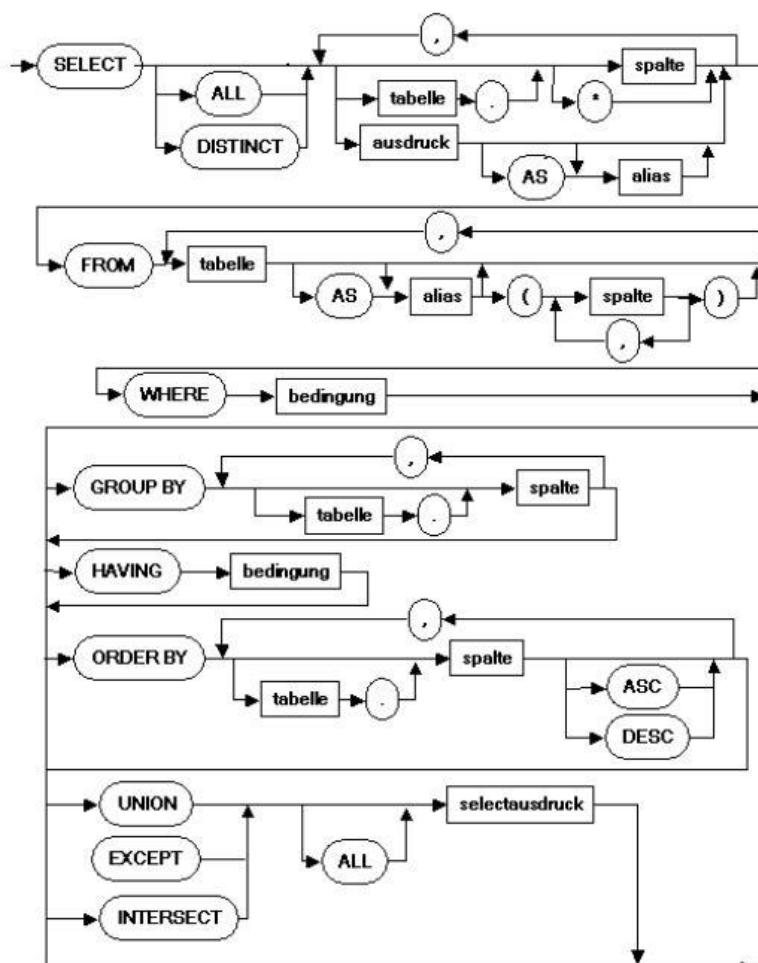
```
SELECT [DISTINCT | ALL]
<spaltenliste> | *
FROM <tabellenliste>
[WHERE <bedingungsliste>]
[GROUP BY <spaltenliste> ]
[HAVING <bedingungsliste>]
[UNION <select-ausdruck>]
[ORDER BY <spaltenliste> ]
;
```

Die Reihenfolge der Klauseln ist fest im SQL-Standard vorgegeben. Klauseln, die in [ ] stehen, sind nicht nötig, sondern können entfallen; der Name des Befehls und die FROM-Angaben sind unbedingt erforderlich, das Semikolon als Standard empfohlen.

## 6.1 SELECT-Anfragen im Überblick

Die SELECT-Anfrage orientiert sich grundsätzlich an folgendem Schema:

Reihe	Klausel	Bedeutung	notw.
1	SELECT	Auswahl der gewünschten Attribute	ja
2	FROM	Angabe der Tabelle(n) mit den gewünschten Information	ja
3	WHERE	Angabe von Bedingungen für die gesuchte Information	nein
4	GROUP BY (HAVING)	Gruppieren der Daten nach bestimmten Kategorien	nein
5	ORDER BY	Sortierung des Ergebnisses	nein



Die Klauseln mit den Schlüsselwörtern WHERE, GROUP BY, HAVING, ORDER BY sind optional. Die Reihenfolge der einzelnen Klauseln ist aber verbindlich. Wird eine Anfrage an die Datenbank gestellt, so erhält man die gewünschte Information in Form einer Tabelle, der Ergebnistabelle, zurück. Das nebenstehende Syntaxdiagramm zeigt alle Möglichkeiten einer SQL-Anfrage.

## 6.2 SELECT - FROM

Die einfachste Form einer SQL-Anfrage besteht aus der **SELECT** - und der **FROM** - Klausel.

Gesucht wird der Inhalt der Tabelle der Fahrzeugherrsteller mit all ihren Spalten und Datensätzen (Zeilen).

```
SELECT * FROM Fahrzeugherrsteller;
```

Schauen wir uns das Beispiel etwas genauer an: - Die beiden Begriffe SELECT und FROM sind SQL-spezifische Bezeichner. - Fahrzeugherrsteller ist der Name der Tabelle, aus der die Daten selektiert und ausgegeben werden sollen. - Das Sternchen, Asterisk genannt, ist eine Kurzfassung für "alle Spalten".

### 6.2.1 Eingrenzen der Spalten

Nun wollen wir nur bestimmte Spalten ausgeben, nämlich eine Liste aller Fahrzeugherrsteller; das Land interessiert uns dabei nicht. Dazu müssen zwischen den SQL-Bezeichnern SELECT und FROM die auszugebenden Spalten angegeben werden. Sind es mehrere, dann werden diese durch jeweils ein Komma getrennt.

#### Aufgabe

Ermitteln Sie die Namensliste aller Fahrzeugherrsteller!

```
SELECT Name FROM Fahrzeugherrsteller;
```

Folgendes Beispiel gibt die beiden Spalten für Name und Land des Herstellers aus. Die Spalten werden durch Komma getrennt.

```
SELECT Name, Land FROM Fahrzeugherrsteller;
```

Für die Ausgabe kann eine (abweichende) Spaltenüberschrift festgelegt werden. Diese wird als **Spalten-Alias** bezeichnet. Der Alias kann dem Spaltennamen direkt folgen oder mit dem Bindewort AS angegeben werden. Das vorherige Beispiel kann also wie folgt mit dem Alias Hersteller für Name und dem Alias Staat für Land versehen werden:

```
SELECT Name, Hersteller, Land AS Staat FROM Fahrzeugherrsteller;
```

## 6.3 DISTINCT - Keine doppelten Zeilen

### Aufgabe

Gesucht wird die Liste der Herstellerländer:

```
SELECT Land FROM Fahrzeugherrsteller;
```

Dabei stellen wir fest, dass je Hersteller eine Zeile ausgegeben wird. Somit erscheint beispielweise 'Deutschland' mehrmals. Damit keine doppelten Zeilen ausgegeben werden, wird DISTINCT vor den Spaltennamen in das SQL-Statement eingefügt:

```
SELECT DISTINCT Land FROM Fahrzeugherrsteller;
```

Damit erscheint jedes Herstellerland nur einmal in der Liste. Die Alternative zu DISTINCT ist übrigens das in der Syntax genannte ALL: alle Zeilen werden gewünscht, ggf. auch doppelte. Dies ist aber der Standardwert, ALL kann weggelassen werden.

Als Mengenquantifizierer stehen somit DISTINCT und ALL zur Verfügung. Außerdem ist es sehr oft möglich (wenn auch nicht im SQL-Standard vorgeschrieben), das Ergebnis auf eine bestimmte Anzahl von Zeilen zu beschränken, z. B. FIRST 3 oder LIMIT 7.

## 6.4 WHERE – Eingrenzen der Ergebnismenge

Fast immer soll nicht der komplette Inhalt einer Tabelle ausgegeben werden. Dazu wird die Ergebnismenge mittels Bedingungen in der WHERE-Klausel eingegrenzt, welche nach dem Tabellennamen im SELECT-Befehl steht. Eine Bedingung ist ein logischer Ausdruck, dessen Ergebnis WAHR oder FALSCH ist. In diesen logischen Ausdrücken werden die Inhalte der Spalten (vorwiegend) mit konstanten Werten verglichen. Hierbei stehen verschiedene Operatoren zur Verfügung, vor allem:

```
=      gleich
<>    ungleich; seltener auch: !=
<      kleiner als
<=     kleiner als oder gleich
>      größer als
>=     größer als oder gleich
```

Bedingungen können durch die logischen Operatoren OR und AND und die Klammern () verknüpft werden. Je komplizierter solche Verknüpfungen werden, desto sicherer ist es, die Bedingungen durch Klammern zu gliedern. Mit diesen Mitteln lässt sich die Abfrage entsprechend eingrenzen.

## Aufgabe

Es sollen alle Hersteller angezeigt werden, die ihren Sitz in Schweden oder Frankreich haben:

```
SELECT * FROM Fahrzeughhersteller WHERE (Land = 'Schweden')
OR ( Land = 'Frankreich' );
```

Hinter der WHERE-Klausel kann man also eine oder mehrere (mit einem booleschen Operator verknüpft) Bedingungen einfügen. Jede einzelne besteht aus dem Namen der Spalte, deren Inhalt überprüft werden soll, und einem Wert, wobei beide mit einem Vergleichsoperator verknüpft sind.

In einer anderen Abfrage sollen alle Fahrzeughhersteller angezeigt werden, die außerhalb Deutschlands sitzen. Jetzt könnte man alle anderen Fälle einzeln in der WHERE-Klausel auflisten, oder man dreht einfach den Vergleichsoperator um.

```
SELECT * FROM Fahrzeughhersteller WHERE Land <> 'Deutschland';
```

Mit der WHERE-Klausel wird eine Suchbedingung festgelegt, welche Zeilen der Ergebnistabelle zur Auswahl gehören sollen. Dieser Filter wird zuerst erstellt; erst anschließend werden Bedingungen wie GROUP BY und ORDER BY ausgewertet.

Die Suchbedingung <search condition> ist eine Konstruktion mit einem eindeutigen Prüfergebnis: Entweder die Zeile gehört zur Auswahl, oder sie gehört nicht zur Auswahl. Es handelt sich also um eine logische Verknüpfung von einer oder mehreren booleschen Variablen.

### 6.4.1 Einzelne Suchbedingung

Eine Suchbedingung hat eine der folgenden Formen, deren Ergebnis immer WAHR oder FALSCH (TRUE bzw. FALSE) lautet:

```
<wert1> [ NOT ] <operator> <wert2>
<wert1> [ NOT ] BETWEEN <wert2> AND <wert3>
<wert1> [ NOT ] LIKE <wert2> [ ESCAPE <wert3> ]
<wert1> [ NOT ] CONTAINING <wert2>
<wert1> [ NOT ] STARTING [ WITH ] <wert2>
<wert1> IS [ NOT ] NULL
<wert1> [ NOT ] IN <werteliste>
EXISTS <select expression>
NOT <search condition>
```

Anstelle eines Wertes kann auch ein Wertausdruck <value expression> stehen, also eine Unterabfrage, die genau einen Wert als Ergebnis liefert. Anstelle einer Werteliste kann auch ein Auswahl-Ausdruck <select expression> stehen, also eine Unterabfrage, die eine Liste mehrerer Werte als Ergebnis liefert. Als <operator> sind folgende Vergleiche möglich, und zwar sowohl für Zahlen als auch für Zeichenketten und Datumsangaben:

= <> <= >= <>

## 6.4.2 BETWEEN AND – Werte zwischen zwei Grenzen

Mit der Bedingung BETWEEN <wert1> AND <wert2> wird direkt mit einem Bereich verglichen; die Grenzwerte gehören meistens zum Bereich (abhängig vom DBMS). Auch dies ist möglich für Zahlen, Zeichenketten, Datumsangaben.

### Aufgabe

Suche Datensätze, bei denen die PLZ außerhalb eines Bereichs 45000...45999 liegt.

```
select * from Versicherungsnehmer where PLZ NOT BETWEEN
'45000' AND '45999';
```

## 6.4.3 LIKE – Ähnlichkeiten

Die LIKE-Bedingung vergleicht Zeichenketten „ungenau“: Der gesuchte Text soll als Wert in einer Spalte enthalten sein; dazu werden „Wildcards“ benutzt: Der Unterstrich ‘\_’ steht für ein beliebiges einzelnes Zeichen, das an der betreffenden Stelle vorkommen kann. Das Prozentzeichen ‘%’ steht für eine beliebige Zeichenkette mit 0 oder mehr Zeichen. Diese Bedingung wird vor allem in zwei Situationen gerne benutzt:

- Der Suchbegriff ist sehr lang; dem Anwender soll es genügen, den Anfang einzugeben.
- Der Suchbegriff ist nicht genau bekannt (z. B. nicht richtig lesbar).

### Beispiele

Der Ortsname beginnt nicht mit ‘B’; der Inhalt dahinter ist beliebig.	<code>select * from Versicherungsnehmer where Ort NOT LIKE 'B%';</code>
Der Ortsname enthält irgendwo ‘alt’ mit beliebigem Inhalt davor und dahinter.	<code>select * from Versicherungsnehmer where Ort LIKE '%alt%';</code>
Der Anfangsbuchstabe des Namens ist unklar, aber danach folgen die Buchstaben ‘ei’ und noch etwas mehr.	<code>select * from Versicherungsnehmer where Name LIKE '_ei%';</code>
Innerhalb der Beschreibung kommt die Zeichenfolge ‘10%’ vor.	<code>select * from Schadensfall where Beschreibung LIKE '%10\%%';</code>

Ein Problem haben wir, wenn eines der Wildcard-Zeichen Teil des Suchbegriffs sein soll. Dann muss dem LIKE-Parameter mitgeteilt werden, dass '%' bzw. '\_' als "echtes" Zeichen zu verstehen ist. Das geschieht dadurch, dass ein spezielles Zeichen davor gesetzt wird und dieses Zeichen als "ESCAPE-Zeichen" angegeben wird:

Das erste und das letzte Prozentzeichen stehen dafür, dass vorher und nachher beliebige Inhalte möglich sind. Das mittlere Prozentzeichen wird mit dem Escape-Zeichen " verbunden und ist damit Teil der gesuchten Zeichenfolge. Diese Angabe '%' ist als ein Zeichen zu verstehen.

Vergleichen Sie das Abfrageergebnis, wenn der ESCAPE-Parameter weggelassen wird oder wenn eines oder mehrere der Sonderzeichen im LIKE-Parameter fehlen.

#### 6.4.4 IS NULL – null-Werte prüfen

NULL-Werte haben eine besondere Bedeutung. Mit den folgenden beiden Abfragen werden nicht alle Datensätze gefunden:

```
select ID, Name, Vorname, Mobil from Mitarbeiter
where Mobil <> '';
//8 Mitarbeiter mit Mobil-Nummer

select ID, Name, Vorname, Mobil
from Mitarbeiter
where Mobil = '';
//10 Mitarbeiter ohne Mobil-Nummer
```

Es gibt allerdings 28 Mitarbeiter insgesamt, die nicht alle gefunden wurden. Für diese Fälle gibt es mit IS NULL eine spezielle Abfrage:

```
select ID, Name, Vorname, Mobil
from Mitarbeiter
where Mobil is null;
//10 Mitarbeiter ohne Angabe
```

Der Vollständigkeit halber sei darauf hingewiesen, dass die folgende Abfrage tatsächlich die richtige Gegenprobe liefert.

```
select ID, Name, Vorname, Mobil from Mitarbeiter
where Mobil is not null;
//18 Mitarbeiter mit irgendeiner Angabe (auch mit "leerer"
Angabe)
```

Die folgende Abfrage liefert eine leere Ergebnismenge zurück, weil NULL eben kein Wert ist.

```
select ID, Name, Vorname, Mobil
from Mitarbeiter
where Mobil = null;
```

Es gibt keine einzelne Bedingung, die alle Datensätze ohne explizite MobilAngabe auf einmal angibt. Es gibt nur die Möglichkeit, die beiden Bedingungen "IS NULL" und "ist leer" zu verknüpfen:

```
select ID, Name, Vorname, Mobil
from Mitarbeiter
where ( Mobil is null ) OR ( Mobil = '' );
```

Beachten Sie auch bei "WHERE ... IS [NOT] NULL" die Bedeutung von NULL:

- Bei Zeichenketten ist zu unterscheiden zwischen dem "leeren" String und dem NULL-Wert.
- Bei Zahlen ist zu unterscheiden zwischen der Zahl '0' (null) und dem NULL-Wert.
- Bei Datumsangaben ist zu unterscheiden zwischen einem vorhandenen Datum und dem NULL-Wert; ein Datum, das der Zahl 0 entspräche, gibt es nicht. (Man könnte allenfalls das kleinste mögliche Datum wie '01.01.0100' benutzen, aber dies ist bereits ein Datum.)

#### 6.4.5 IN – genauer Vergleich mit einer Liste

Der IN-Parameter vergleicht, ob der Inhalt einer Spalte in der angegebenen Liste enthalten ist. Die Liste kann mit beliebigen Datentypen arbeiten.

Hole die Liste aller Fahrzeuge, deren Typen als "VW-Kleinwagen" registriert sind.	<pre><b>select</b> * <b>from</b> Fahrzeug <b>where</b> Fahrzeugtyp_ID <b>in</b> (1, 2);</pre>
Suche nach einem Unfall Fahrzeuge mit einer von mehreren möglichen Farben.	<pre><b>select</b> * <b>from</b> Fahrzeug <b>where</b> Farbe <b>in</b> ('ocker', 'gelb');</pre>
Hole die Liste aller Fahrzeuge vom Typ „Volkswagen“.	<pre><b>select</b> * <b>from</b> Fahrzeug <b>where</b> Fahrzeugtyp_ID <b>in</b> (<b>select</b> ID <b>from</b> Fahrzeugtyp <b>where</b> Hersteller_ID = 1);</pre>

Dabei wird zuerst mit der Unterabfrage eine Liste aller Fahrzeugtypen-IDs für den Hersteller 1 (= Volkswagen) zusammengestellt; diese wird dann für den Vergleich über den IN-Parameter benutzt.

## 6.4.6 EXISTS – schneller Vergleich mit einer Liste

Im Gegensatz zu den anderen Parametern der WHERE-Klausel arbeitet der EXISTS-Parameter nicht mit fest vorgegebenen Werten, sondern nur mit dem Ergebnis einer Abfrage, also einer Unterabfrage. Das letzte Beispiel zum IN-Parameter kann auch so formuliert werden:

```
//Liste aller Fahrzeuge vom Typ 'Volkswagen'
select * from Fahrzeug fz
where EXISTS
  ( select * from Fahrzeugtyp ft
    where ft.Hersteller_ID = 1
    and fz.Fahrzeugtyp_ID = ft.ID );
```

Zu jedem Datensatz aus der Tabelle Fahrzeug wird zu dieser Fahrzeugtyp\_ID eine Unterabfrage aus den Fahrzeugtypen erstellt: Wenn es dort einen Datensatz mit passender ID und Hersteller-ID 1 (= Volkswagen) gibt, gehört der Fahrzeug-Datensatz zur Auswahl, andernfalls nicht.

Da Unterabfragen zuerst ausgeführt werden, wird eine EXISTS-Prüfung in aller Regel schneller erledigt als die entsprechende IN-Prüfung: Bei EXISTS handelt es sich um eine Feststellung “ist überhaupt etwas vorhanden”; bei IN dagegen muss ein exakter Vergleich mit allen Werten einer Liste durchgeführt werden. Bei unserer kleinen Beispieldatenbank spielt das natürlich keine Rolle, aber bei einer “echten” Datenbank mit Millionen von Einträgen schon.

## 6.4.7 Mehrere Bedingungen

Bei der WHERE-Klausel geht es darum festzustellen, ob ein Datensatz Teil des Abfrageergebnisses ist oder nicht; bei der <search condition> handelt sich also um einen booleschen Ausdruck, d. h. einen Ausdruck, der einen der booleschen Werte WAHR oder FALSCH – TRUE bzw. FALSE – als Ergebnis hat. Nur bei einfachen Abfragen genügt dazu eine einzelne Bedingung; meistens müssen mehrere Bedingungen verknüpft werden (wie beim letzten Beispiel unter IS NULL).

Dazu gibt es die booleschen Operatoren NOT, AND, OR.

### 6.4.7.1 NOT als Negation

Diese liefert die Umkehrung: aus TRUE wird FALSE, aus FALSE wird TRUE.

```
SELECT * FROM Versicherungsnehmer
WHERE NOT (Fuehrerschein >= '01.01.2007');
```

### 6.4.7.2 AND als Konjunktion

Eine Bedingung, die durch eine AND-Verknüpfung gebildet wird, ist genau dann TRUE, wenn beide (bzw. alle) Bestandteile TRUE sind.

```
--Die nach Alphabet erste Hälfte der Versicherungsnehmer ei-
nes --PLZ-Bereichs
```

```
SELECT ID, Name, Vorname, PLZ, Ort
FROM Versicherungsnehmer
WHERE PLZ BETWEEN '45000' AND '45999'
AND Name < 'K';
```

### 6.4.7.3 OR als Adjunktion

Eine Bedingung, die durch eine OR-Verknüpfung gebildet wird, ist genau dann TRUE, wenn mindestens ein Bestandteil TRUE ist; dabei ist es gleichgültig, ob die anderen Bestandteile TRUE oder FALSE sind.

```
//Die nach Alphabet erste Hälfte der Versicherungsnehmer und
//alle eines PLZ-Bereichs
```

```
SELECT ID, Name, Vorname, PLZ, Ort
FROM Versicherungsnehmer
WHERE PLZ BETWEEN '45000' AND '45999'
OR Name < 'K';
```

Bitte beachten Sie, dass der normale Sprachgebrauch "alle ... und alle ..." sagt. Gemeint ist nach logischen Begriffen aber, dass <Bedingung 1> erfüllt sein muss ODER <Bedingung 2> ODER BEIDE.

### 6.4.7.4 XOR als Kontravalenz

Eine Bedingung, die durch eine XOR-Verknüpfung gebildet wird, ist genau dann TRUE, wenn ein Bestandteil TRUE ist, aber der andere Bestandteil FALSE ist – "ausschließendes oder" bzw. "entweder – oder". Diese Verknüpfung gibt es selten, z. B. bei MySQL; hier wird es der Vollständigkeit halber erwähnt.

```
//Die nach Alphabet erste Hälfte der Versicherungsnehmer
//oder alle eines PLZ-Bereichs
```

```
SELECT ID, Name, Vorname, PLZ, Ort
FROM Versicherungsnehmer
WHERE PLZ BETWEEN '45000' AND '45999'
XOR Name < 'K';
```

Bitte beachten Sie, dass hier der normale Sprachgebrauch „oder“ sagt und „entweder-oder“ gemeint ist. Anstelle von XOR kann immer eine Kombination verwendet werden:

```
( <Bedingung 1> AND ( NOT <Bedingung 2> ) )
OR ( <Bedingung 2> AND ( NOT <Bedingung 1> ) )
```

#### 6.4.7.5 Klammern benutzen oder weglassen?

1. NOT ist die engste Verbindung und wird vorrangig ausgewertet.
2. AND ist die nächststärkere Verbindung und wird danach ausgewertet.
3. OR ist die schwächste Verbindung und wird zuletzt ausgewertet.
4. Was in Klammern steht, wird vor allem anderen ausgewertet.

Bitte setzen Sie im folgenden Beispiel Klammern an anderen Stellen oder streichen Sie Klammern, und vergleichen Sie die Ergebnisse.

```
SELECT ID, Name, Vorname, PLZ, Ort FROM Versicherungsnehmer
WHERE not ( PLZ BETWEEN '45000' AND '45999'
AND ( Name LIKE 'B%' )
OR Name LIKE 'K%'
OR NOT Name CONTAINING 'ei'
)
)
order by PLZ, Name;
```

Sie werden ziemlich unterschiedliche Ergebnisse erhalten. Es empfiehlt sich deshalb, an allen sinnvollen Stellen Klammern zu setzen – auch dort, wo sie nicht erforderlich sind – und das, was zusammengehört, durch Einrückungen sinnvoll zu gliedern.

#### 6.4.7.6 Zusammenfassung

In diesem Kapitel lernten wir neben dem Vergleich von Werten viele Möglichkeiten kennen, mit denen Bedingungen für Abfragen festgelegt werden können:

- Mit BETWEEN AND werden Werte innerhalb eines Bereichs geprüft.
- Mit LIKE und CONTAINS werden Werte gesucht, die mit vorgegebenen Werten teilweise übereinstimmen.
- Mit IS NULL werden null-Werte gesucht.
- Mit IN und EXISTS werden Spaltenwerte mit einer Liste verglichen.
- Mit AND, OR, NOT werden Bedingungen zusammengefasst.

## 6.4.8 Übungen

### Auswahl nach Zeichenketten

Suchen Sie alle Versicherungsnehmer, die folgenden Bedingungen entsprechen:

- Der erste Buchstabe des Nachnamens ist nicht bekannt, der zweite ist ein 'r'.
- Der Vorname enthält ein 'a'.
- Die Postleitzahl gehört zum Bereich Essen (PLZ 45...).

### Auswahl nach Datumsbereich

Suchen Sie alle Versicherungsnehmer, die in den Jahren 1967 bis 1970 ihren 18. Geburtstag hatten.

### Auswahl nach Ähnlichkeit

Zeigen Sie alle Schadensfälle an, bei denen in der Beschreibung auf eine prozentuale Angabe hingewiesen wird.

### Auswahl für unbekannte Werte

Zeigen Sie alle Dienstwagen an, die keinem Mitarbeiter persönlich zugeordnet sind.

### Bedingungen verknüpfen

Zeigen Sie alle Mitarbeiter der Abteilungen „Vertrieb“ (= ‘Vert’) und „Ausbildung“ (= ‘Ausb’) an.

Hinweis: Bestimmen Sie zunächst die IDs der gesuchten Abteilungen und benutzen Sie das Ergebnis für die eigentliche Abfrage.

### Bedingungen verknüpfen

Gesucht werden die Versicherungsverträge für Haftpflicht (= ‘HP’) und Teilkasko (= ‘TK’), die mindestens seit dem Ende des Jahres 1980 bestehen und aktuell nicht mit dem minimalen Prämiensatz berechnet werden. Hinweis: Tragen Sie ausnahmsweise nur die notwendigen Klammern ein, nicht alle sinnvollen.

## Lösung

### Auswahl nach Zeichenketten

Suchen Sie alle Versicherungsnehmer, die folgenden Bedingungen entsprechen: - Der erste Buchstabe des Nachnamens ist nicht bekannt, der zweite ist ein 'r'. - Der Vorname enthält ein 'a'. - Die Postleitzahl gehört zum Bereich Essen (PLZ 45...).

```
select * from Versicherungsnehmer  
where Name like '_r%' and Vorname like '%a%'  
and PLZ like '45%';
```

### Auswahl nach Datumsbereich

Suchen Sie alle Versicherungsnehmer, die in den Jahren 1967 bis 1970 ihren 18. Geburtstag hatten.

```
select * from Versicherungsnehmer  
where DATEADD(YEAR, 18, Geburtsdatum) BETWEEN '01.01.1967'  
AND '31.12.1970';
```

### Auswahl nach Ähnlichkeit

Zeigen Sie alle Schadensfälle an, bei denen in der Beschreibung auf eine prozentuale Angabe hingewiesen wird.

```
SELECT * from Schadensfall  
where Beschreibung like '%\%%' escape '\';
```

### Auswahl für unbekannte Werte

Zeigen Sie alle Dienstwagen an, die keinem Mitarbeiter persönlich zugeordnet sind.

```
SELECT * from Dienstwagen  
where Mitarbeiter_ID is null;
```

## Bedingungen verknüpfen

Zeigen Sie alle Mitarbeiter der Abteilungen „Vertrieb“ (= ‘Vert’) und „Ausbildung“ (= ‘Ausb’) an.

Hinweis: Bestimmen Sie zunächst die IDs der gesuchten Abteilungen und benutzen Sie das Ergebnis für die eigentliche Abfrage.

```
SELECT * from Mitarbeiter
where Abteilung_ID in (
    select id from Abteilung
    where Kuerzel in ('Vert', 'Ausb')
);
```

## Bedingungen verknüpfen

Gesucht werden die Versicherungsverträge für Haftpflicht (= ‘HP’) und Teilkasko (= ‘TK’), die mindestens seit dem Ende des Jahres 1980 bestehen und aktuell nicht mit dem minimalen Prämienatz berechnet werden. Hinweis: Tragen Sie ausnahmsweise nur die notwendigen Klammern ein, nicht alle sinnvollen.

where (Art = ‘HP’ or Art = ‘TK’) and Abschlussdatum <= ‘31.12.1980’ /\* ab hiergeht es nicht weil spalte praemiensatz nicht vorhanden / and (not Praemiensatz = 30) / oder and Praemiensatz > 30; \*/

## 6.5 ORDER BY – Sortieren

Nachdem wir nun die Zeilen und Spalten der Ergebnismenge eingrenzen können, wollen wir die Ausgabe der Zeilen sortieren. Hierfür wird die ORDER BY-Klausel genutzt. Diese ist die letzte im SQL-Befehl vor dem abschließenden Semikolon und enthält die Spalten, nach denen sortiert werden soll.

### Aufgabe

Lassen sie sich die Liste der Hersteller nach dem Namen sortiert ausgeben:

```
SELECT * FROM Fahrzeughhersteller ORDER BY Name;
```

Anstatt des Spaltennamens kann auch die Nummer der Spalte genutzt werden. Mit dem folgenden Statement erreichen wir also das gleiche Ergebnis, da Name die 2. Spalte in unserer Ausgabe ist:

```
SELECT * FROM Fahrzeughhersteller ORDER BY 2;
```

Die Angabe nach Spaltennummer ist unüblich; sie wird eigentlich höchstens dann verwendet, wenn die Spalten genau aufgeführt werden und komplizierte Angaben – z. B. berechnete Spalten enthalten. Die Sortierung erfolgt standardmäßig aufsteigend; das kann auch durch ASC ausdrücklich angegeben werden. Die Sortierreihenfolge kann mit dem DESC-Bezeichner in absteigend verändert werden.

```
SELECT * FROM Fahrzeughhersteller ORDER BY Name DESC;
```

In SQL kann nicht nur nach einer Spalte sortiert werden. Es können mehrere Spalten zur Sortierung herangezogen werden. Hierbei kann für jede Spalte eine eigene Regel verwendet werden. Dabei gilt, dass die Regel zu einer folgend angegebenen Spalte der Regel zu der vorig angegebenen Spalte untergeordnet ist. Bei der Sortierung nach Land und Name wird also zuerst nach dem Land und dann je Land nach Name sortiert. Eine Neusortierung nach Name, die jene Sortierung nach Land wieder verwirft, findet also nicht statt.

Der folgende Befehl liefert die Hersteller – zuerst absteigend nach Land und dann aufsteigend sortiert nach dem Namen – zurück.

```
SELECT * FROM Fahrzeughersteller ORDER BY Land DESC, Name
ASC;
```

## 6.6 FROM – Mehrere Tabellen verknüpfen

In fast allen Abfragen werden Informationen aus mehreren Tabellen zusammengefasst. Die sinnvolle Speicherung von Daten in getrennten Tabellen ist eines der Merkmale eines relationalen DBMS; deshalb müssen die Daten bei einer Abfrage nach praktischen Gesichtspunkten zusammengeführt werden.

### Aufgabe

Informieren Sie sich über den Begriff "Kartesisches Kreuzprodukt"

### 6.6.1 FROM und WHERE

Beim „traditionellen“ Weg werden dazu einfach alle Tabellen in der FROM-Klausel aufgeführt und durch jeweils eine Bedingung in der WHERE-Klausel verknüpft.

### Aufgabe

Ermittle die Angaben der Mitarbeiter incl. Abteilungsname, deren Abteilung ihren Sitz in Dortmund oder Bochum hat.

```
SELECT mi.Name, mi.Vorname, mi.Raum, ab.Ort
FROM Mitarbeiter mi, Abteilung ab
WHERE mi.Abteilung_ID = ab.ID
      AND ab.Ort in ('Dortmund', 'Bochum')
ORDER BY mi.Name, mi.Vorname;
```

Es werden also Informationen aus den Tabellen Mitarbeiter (Name und Raum) sowie Abteilung (Ort) gesucht.

- In der FROM-Klausel stehen die benötigten Tabellen.
- Zur Vereinfachung wird jeder Tabelle ein Kürzel als Tabellen-Alias zugewiesen.
- In der Spaltenliste wird jede einzelne Spalte mit dem Namen der betreffenden Tabelle bzw. dem Alias verbunden.
- Die WHERE-Klausel enthält die Verknüpfungsbedingung „mi.Abteilung\_ID = ab.ID“ – zusätzlich zur Einschränkung nach dem Sitz der Abteilung.
- Jede Tabelle in einer solchen Abfrage benötigt mindestens eine direkte Verknüpfung zu einer anderen Tabelle.

## 6.6.2 JOIN ... ON

Beim „modernen“ Weg wird eine Tabelle in der FROM-Klausel aufgeführt, nämlich diejenige, die als wichtigste oder „Haupttabelle“ der Abfrage angesehen wird. Eine weitere Tabelle wird durch JOIN und eine Bedingung in der ON-Klausel verknüpft. Das obige Beispiel sieht dann so aus:

```
SELECT mi.Name, mi.Vorname, mi.Raum, ab.Ort
FROM Mitarbeiter mi
JOIN Abteilung ab
    ON mi.Abteilung_ID = ab.ID
WHERE ab.Ort IN ('Dortmund', 'Bochum')
ORDER BY mi.Name, mi.Vorname;
```

Für die Verknüpfung der Tabellen werden folgende Bestandteile benötigt: - In der FROM-Klausel steht eine der benötigten Tabellen. - In der JOIN-Klausel steht jeweils eine weitere Tabelle. - Die ON-Klausel enthält die Verknüpfungsbedingung „mi.Abteilung\_ID = ab.ID“. - Die WHERE-Klausel beschränkt sich auf die wirklich gewünschten Einschränkungen für die Ergebnismenge.

Ein Tabellen-Alias ist wiederum für alle Tabellen sinnvoll. In der Spaltenliste und auch zur Sortierung können alle Spalten aller Tabellen benutzt werden. Vertiefte Erläuterungen sind unter „Arbeiten mit JOIN“ zu finden.

## 6.7 Zusammenfassung

- SELECT-Befehle werden zur Abfrage von Daten aus Datenbanken genutzt.
- Die auszugebenden Spalten können festgelegt werden, indem die Liste der Spalten zwischen den Bezeichnern SELECT und FROM angegeben wird.
- Mit DISTINCT werden identische Zeilen in der Ergebnismenge nur einmal ausgegeben.
- Die Ergebnismenge wird mittels der WHERE-Klausel eingegrenzt.
- Die WHERE-Klausel enthält logische Ausdrücke. Diese können mit AND und OR verknüpft werden.
- Mittels der ORDER BY-Klausel kann die Ergebnismenge sortiert werden.

Die Reihenfolge innerhalb eines SELECT-Befehls ist zu beachten. SELECT und FROM sind hierbei Pflicht, das abschließende Semikolon als Standard empfohlen. Alle anderen Klauseln sind optional.

## 6.8 Übungen

- Welche Bestandteile eines SELECT-Befehls sind unbedingt erforderlich und können nicht weggelassen werden?
- Geben Sie alle Informationen zu allen Abteilungen aus.
- Geben Sie alle Abteilungen aus, deren Standort Bochum ist.
- Geben Sie alle Abteilungen aus, deren Standort Bochum oder Essen ist. Hierbei soll nur der Name der Abteilung ausgegeben werden.
- Geben Sie nur die Kurzbezeichnungen aller Abteilungen aus. Hierbei sollen die Abteilungen nach den Standorten sortiert werden.

## 6.9 Lösung

Welche Bestandteile eines SELECT-Befehls sind unbedingt erforderlich und können nicht weggelassen werden?

```
SELECT, Spaltenliste oder '*', FROM, Tabellenname
```

Geben Sie alle Informationen zu allen Abteilungen aus.

```
select * from Abteilung;
```

Geben Sie alle Abteilungen aus, deren Standort Bochum ist.

```
select * from Abteilung where Ort = 'Bochum';
```

Geben Sie alle Abteilungen aus, deren Standort Bochum oder Essen ist. Hierbei soll nur der Name der Abteilung ausgegeben werden.

```
select Bezeichnung from Abteilung where Ort = 'Bochum' or Ort = 'Essen';
select Bezeichnung from Abteilung where Ort in ('Bochum', 'Essen');
```

Geben Sie nur die Kurzbezeichnungen aller Abteilungen aus. Hierbei sollen die Abteilungen nach den Standorten sortiert werden.

```
select Kuerzel from Abteilung order by Ort;
```

## 6.10 Aggregatfunktionen

Funktion	Beschreibung	Beispiel	Erklärung
COUNT()	Anzahl	COUNT(*)	Anzahl aller Zeilen einer Tabelle
SUM()	Summe	SUM(Preis*Menge)	Summe von Preis mal Menge
MAX()	Maximum	MAX(Bestand)	Großter Bestand
MIN()	Minimum	MIN(Preis)	Kleinster Preis
AVG()	Durchschnitt	AVG(Lieferdauer)	Durchschnittliche Lieferdauer
ROUND()	Runden	ROUND(Preis,3)	Runden auf 3 Nachkommast.
DISTINCT	Eindeutig		Verhindert mehrfache gleiche Ergebniszellen

Die Spaltenfunktionen werden auch als Aggregatfunktionen bezeichnet, weil sie eine Menge von Werten – nämlich aus allen Zeilen einer bestimmten Spalte – zusammenfassen und einen gemeinsamen Wert bestimmen. In der Regel wird dazu eine Spalte aus einer der beteiligten Tabellen verwendet; es kann aber auch ein sonstiger gültiger SQL-Ausdruck sein, der als Ergebnis einen einzelnen Wert liefert. Das Ergebnis der Funktion ist dann ein Wert, der aus allen passenden Zeilen der Abfrage berechnet wird. Bei Abfragen kann das Ergebnis einer Spaltenfunktion auch nach den Werten einer oder mehrerer Spalten oder Berechnungen gruppiert werden. Die Aggregatfunktionen liefern dann für jede Gruppe ein Teilergebnis.

### 6.10.1 COUNT – Anzahl

Die Funktion COUNT zählt alle Zeilen, die einen eindeutigen Wert enthalten, also nicht NULL sind. Sie kann auf alle Datentypen angewendet werden, da für jeden Datentyp NULL definiert ist.

#### Beispiel

```
SELECT COUNT(Farbe) AS Anzahl_Farbe FROM Fahrzeug;
```

Die Spalte Farbe ist als VARCHAR(30), also als Text variabler Länge, definiert und optional. Hier werden also alle Zeilen gezählt, die in dieser Spalte einen Eintrag haben. Dasselbe funktioniert auch mit einer Spalte, die numerisch ist:

```
SELECT COUNT(Schadenshoehe) AS Anzahl_Schadenshoehe FROM Schadensfall;
```

Hier ist die Spalte numerisch und optional. Die Zahl 0 ist bekanntlich nicht NULL. Wenn in der Spalte eine 0 steht, wird sie mitgezählt.

Ein Spezialfall ist der Asterisk '\*' als Parameter. Dies bezieht sich dann nicht auf eine einzelne Spalte, sondern auf eine ganze Zeile. So wird also die Anzahl der Zeilen in der Tabelle gezählt:

```
SELECT COUNT(*) AS Anzahl_Zeilen FROM Schadensfall;
```

Die Funktion COUNT liefert niemals NULL zurück, sondern immer eine Zahl; wenn alle Werte in einer Spalte NULL sind, ist das Ergebnis die Zahl 0 (es gibt 0 Zeilen mit einem Wert ungleich NULL in dieser Spalte).

## 6.10.2 SUM – Summe

Die Funktion SUM kann (natürlich) nur auf numerische Datentypen angewendet werden. Im Gegensatz zu COUNT liefert SUM nur dann einen Wert zurück, wenn wenigstens ein Eingabewert nicht NULL ist. Als Parameter kann nicht nur eine einzelne numerische Spalte, sondern auch eine Berechnung übergeben werden, die als Ergebnis eine einzelne Zahl liefert. Ein Beispiel für eine einzelne numerische Spalte ist:

```
SELECT SUM(Schadenshoehe) AS Summe_Schadenshoehe  
FROM Schadensfall;
```

Als Ergebnis werden alle Werte in der Spalte Schadenshoehe aufsummiert. Als Parameter kann aber auch eine Berechnung übergeben werden.

### Aufgabe

Hier werden Euro-Beträge aus Schadenshoehe zuerst in US-Dollar nach einem Tageskurs umgerechnet und danach aufsummiert.

```
SELECT SUM(Schadenshoehe * 1.5068) AS Summe_Schadenshoehe_Dollar  
FROM Schadensfall;
```

## 6.10.3 MAX, MIN – Maximum, Minimum

Diese Funktionen können auf jeden Datentyp angewendet werden, für den ein Vergleich ein gültiges Ergebnis liefert. Dies gilt für numerische Werte, Datumswerte und Textwerte, nicht aber für z. B. BLOBs (binary large objects). Bei Textwerten ist zu bedenken, dass die Sortierreihenfolge je nach verwendetem Betriebssystem, DBMS und Zeichensatzeinstellungen der Tabelle oder Spalte unterschiedlich ist, die Funktion demnach auch unterschiedliche Ergebnisse liefern kann.

### Aufgabe: Suche den kleinsten, von NULL verschiedenen Schadensfall.

```
SELECT MIN(Schadenshoehe) AS Minimum_Schadenshoehe  
FROM Schadensfall;
```

---

Kommen nur NULL-Werte vor, wird NULL zurückgegeben. Gibt es mehrere Zeilen, die den kleinsten Wert haben, wird trotzdem nur ein Wert zurückgegeben. Welche Zeile diesen Wert liefert, ist nicht definiert.

Für MAX gilt Entsprechendes wie für MIN.

#### 6.10.4 AVG – Mittelwert

AVG (average = Durchschnitt) kann nur auf numerische Werte angewendet werden. Das für SUM Gesagte gilt analog auch für AVG. Um die mittlere Schadenshöhe zu berechnen, schreibt man:

```
SELECT AVG(Schadenshoehe) AS Mittlere_Schadenshoehe FROM Schadensfall;
```

NUL-Werte fließen dabei nicht in die Berechnung ein, Nullen aber sehr wohl.

## 6.11 Gruppierungen

Abfragen werden sehr häufig gruppiert, weil nicht nur einzelne Informationen, sondern auch Zusammenfassungen gewünscht werden. Durch die GROUP BY-Klausel im SELECT-Befehl werden alle Zeilen, die in einer oder mehreren Spalten den gleichen Wert enthalten, in jeweils einer Gruppe zusammengefasst. Dies macht in der Regel nur dann Sinn, wenn in der Spaltenliste des SELECT-Befehls eine gruppenweise Auswertung, also eine der Spaltenfunktionen enthalten ist.

Die GROUP BY-Klausel hat folgenden allgemeinen Aufbau:

```
GROUP BY <Spaltenliste>
```

Die Spaltenliste enthält, durch Komma getrennt, die Namen von einer oder mehreren Spalten.

Für jede Spalte kann eine eigene Sortierung angegeben werden:

```
<Spaltenname>
-- oder
<Spaltenname> COLLATE <Collation-Name>
```

Die Spalten in der Spaltenliste können meistens wahlweise mit dem Spaltennamen der Tabelle, mit dem Alias-Namen aus der Select-Liste oder mit Spaltennummer gemäß der Select-Liste (ab 1 gezählt) angegeben werden. In der Regel enthält die Abfrage eine der Aggregatfunktionen und wird durch ORDER BY nach den gleichen Spalten sortiert.

Im einfachsten Fall werden Daten nach einer Spalte gruppiert und gezählt.

Im folgenden Beispiel wird die Anzahl der Abteilungen für jeden Ort aufgeführt.

```
SELECT Ort, COUNT(*) AS Anzahl
FROM Abteilung
GROUP BY Ort
ORDER BY Ort;

Bochum 3
Dortmund 4
Essen 4
Herne 1
```

## Aufgabe

Die folgende Abfrage listet auf, wie viele Mitarbeiter es in den Abteilungen und Raumnummern gibt:

```
SELECT Abteilung_ID AS Abt, Raum, COUNT(*) AS Anzahl
FROM Mitarbeiter
GROUP BY Abt, Raum
ORDER BY Abt, Raum;
ABT RAUM ANZAL
1 112 1
1 113 1
2 212 2
3 312 1
3 316 1
4 412 2 // usw.
```

Am folgenden Beispiel wird die Gruppierung besonders deutlich. Gruppierung über mehrere Tabellen

## Aufgabe

Berechne die mittlere Schadenshöhe für die Schadensfälle mit und ohne Personenschäden.

```
SELECT Verletzte,
AVG(Schadenshoehe) AS Mittlere_Schadens-
hoehe
FROM Schadensfall
GROUP BY Verletzte;

VERLETZTE MITTLERE_SCHADENSHOEHE
J 3.903,87
N 1.517,45
```

Die Spalte Verletzte enthält entweder 'J' oder 'N' und ist verpflichtend, kann also keine NULL-Werte enthalten. Deshalb werden durch die GROUP BY-Anweisung eine oder zwei Gruppen gebildet. Für jede Gruppe wird der Mittelwert gesondert berechnet aus den Werten, die in der Gruppe vorkommen. In diesem Fall liefert die Funktion AVG also ein oder zwei Ergebnisse, abhängig davon, welche Werte in der Spalte Verletzte überhaupt vorkommen. Zeilen, bei denen einer der Werte zur Gruppierung fehlt, oder Zeilen mit NULL-Werten werden als eigene Gruppe gezählt.

Eine Gruppierung kann auch Felder aus verschiedenen Tabellen auswerten. Dafür sind zunächst die Voraussetzungen für die Verknüpfung mehrerer Tabellen zu beachten.

## Beispiel

Gesucht wird für jeden Fahrzeughersteller (mit Angabe von ID und Name) und Jahr die Summe der Schadenshöhe aus der Tabelle Schadensfall.

```
SELECT fh.ID AS Hersteller_ID,
fh.Name AS Hersteller_Name,
EXTRACT(YEAR FROM sf.Datum) AS Jahr,
SUM(sf.Schadenshoehe) AS Schadenssumme
FROM Schadensfall sf
JOIN Zuordnung_SF_FZ zu ON sf.ID = zu.Schadensfall_ID
JOIN Fahrzeug fz ON fz.ID = zu.Fahrzeug_ID
JOIN Fahrzeugtyp ft ON ft.ID = fz.Fahrzeugtyp_ID
JOIN Fahrzeughhersteller fh ON fh.ID = ft.Hersteller_ID
GROUP BY Hersteller_ID, Hersteller_Name, Jahr
ORDER BY Jahr, Hersteller_ID;
```

	HERSTELLER_ID	HERSTELLER_NAME	JAHR	SCHADENSSUMME
9	Volvo	2007	2.066,00	
10	Renault	2007	5.781,60	
11	Seat	2007	1.234,50	
2	Opel	2008	1.438,75	
11	Seat	2008	1.983,00	
9	Volvo	2009	4.092,15	
10	Renault	2009	865,00	

Ausgangspunkt ist die Tabelle Schadensfall, weil aus deren Einträgen die Summe gebildet werden soll. Durch JOIN werden nacheinander die verknüpften Tabellen herangezogen, und zwar jeweils durch die ID auf die Verknüpfung: Schadensfall → Zuordnung → Fahrzeug → Fahrzeugtyp → Hersteller. Dann stehen ID und Name aus der Tabelle Fahrzeughhersteller zur Verfügung, die für die Gruppierung gewünscht werden.

Zur Gruppierung genügt eigentlich die Verwendung von Hersteller\_ID. Aber man möchte sich natürlich den Herstellernamen anzeigen lassen. Allerdings gibt es einen Fehler, wenn man den Namen nur in der SELECT-Liste benutzt und in der GROUP BY-Liste streicht.

```
/*Quelltext Falsch*/
SELECT fh.ID AS Hersteller_ID,
fh.Name AS Hersteller_Name,
YEAR(sf.Datum) AS Jahr,
SUM(sf.Schadenshoehe) AS Schadenssumme
FROM Schadensfall sf
join ... (wie oben)
group by Hersteller_ID, Jahr
order by Jahr, Hersteller_ID;
```

Ungültiger Ausdruck **in** der **Select**-Liste  
(fehlt entweder **in** einer Aggregatfunktion  
oder **in** der **GROUP BY**-Klausel).

## Einschränkungen

Wie das letzte Beispiel zeigt, muss die GROUP BY-Klausel gewisse Bedingungen erfüllen. Auch dafür gilt: Jedes DBMS weicht teilweise vom Standard ab.

Jeder Spaltenname der SELECT-Auswahl, der nicht zu einer Aggregatfunktion gehört, muss auch in der GROUP BY-Klausel benutzt werden.

Diese Bedingung wird im letzten Beispiel verletzt: Hersteller\_Name steht in der SELECT-Liste, aber nicht in der GROUP BY-Klausel. In diesem Fall ist eine Änderung einfach, weil ID und Name des Herstellers gleichwertig sind. Übrigens erlaubt MySQL auch die Auswahl von Feldern, die in der GROUP BY-Klausel nicht genannt sind.

Umgekehrt ist es in der Regel möglich, eine Spalte per GROUP BY zu gruppieren, ohne die Spalte selbst in der SELECT-Liste zu verwenden.

GROUP BY kann Spalten der Tabellen, abgeleiteter Tabellen oder VIEWS in der FROM-Klausel oder der JOIN-Klausel enthalten. Sie kann keiner Spalte entsprechen, die das Ergebnis einer Funktion (genauer: einer numerischen Berechnung, einer Aggregatfunktion oder einer benutzerdefinierten Funktion) sind.

## 6.11.1 HAVING

Diese Erweiterung ist eine selbständige Klausel des SELECT-Befehls und hat eigentlich nichts mit der GROUP BY-Klausel zu tun. In der Praxis wird sie aber überwiegend als Ergänzung zur Gruppierung verwendet und folgt ggf. direkt danach.

```
GROUP BY <spaltenliste>
HAVING <bedingungen>
```

Dieser Befehl dient dazu, nicht alle Gruppierungen in die Ausgabe zu übernehmen, sondern nur diejenigen, die den zusätzlichen Bedingungen entsprechen. Im folgenden Beispiel wird festgestellt, an welchen Orten sich genau eine Abteilung befindet.

```
SELECT Ort, COUNT(*) AS Anzahl
FROM Abteilung
GROUP BY Ort
HAVING COUNT(*) = 1
ORDER BY Ort;
```

Bitte beachten Sie, dass der Alias-Name nicht verwendet werden kann, sondern die Aggregatfunktion erneut aufgeführt werden muss.

## 6.11.2 Übungen

1. Welche der folgenden Feststellungen sind wahr, welche sind falsch?
  1. GROUP BY kann nur zusammen mit (mindestens) einer Spaltenfunktion benutzt werden.
  2. GROUP BY kann nur auf „echte“ Spalten angewendet werden, nicht auf berechnete Spalten.
  3. In der GROUP BY-Klausel kann ein Spaltenname ebenso angegeben werden wie ein Spalten-Alias.
  4. Die WHERE-Klausel kommt vor der GROUP BY-Klausel.
  5. Folgende Gruppierung nach den ersten zwei Ziffern der PLZ ist zulässig.

```
select PLZ, COUNT(*)  
from Versicherungsnehmer vn  
group by SUBSTRING(vn.PLZ from 1 for 2)  
order by 1;
```

6. HAVING darf nur zusammen mit einer Gruppierung verwendet werden.
2. Bestimmen Sie die Anzahl der Kunden (Versicherungsnehmer) in jedem Briefzentrumb (d. h. die Ziffern 1 und 2 der PLZ).
3. Bestimmen Sie, wie viele Fahrzeuge in jedem Kreis angemeldet sind.
4. Bestimmen Sie, wie viele Fahrzeugtypen pro Hersteller registriert sind, und nennen Sie Namen und Land der Hersteller.

Hinweis: Erstellen Sie zunächst eine Abfrage für Anzahl plus Hersteller-ID und verknüpfen Sie das Ergebnis mit der Tabelle Hersteller.

5. Bestimmen Sie, gruppiert nach Jahr des Schadensfalls und Kreis des Fahrzeugs, die Anzahl der Schadensfälle. Es sollen bei den Fahrzeugen nur Schadensfälle mit einem Schuldanteil von mehr als 50 [Prozent] berücksichtigt werden.

### 6.11.3 Lösungen

#### 1. Lösung zu Übung 1

Richtig sind die Aussagen 3, 4. Falsch sind die Aussagen 1, 2, 5, 6.

#### 2. Lösung zu Übung 2

```
select SUBSTRING(vn.PLZ from 1 for 2), COUNT(*)
from Versicherungsnehmer vn
group by 1
order by 1;
```

#### 3. Lösung zu Übung 3

```
select SUBSTRING(Kennzeichen from 1 for POSITION('-', Kennzeichen)-1 ) as Kreis,
COUNT(*) as Anzahl
from Fahrzeug fz
group by 1
order by 1;
```

#### 4. Lösung zu Übung 4

```
-- Lösung mit temp-tabelle als subselect
select Name, Land, Anzahl
from (
select ft.Hersteller_ID as ID, Count(ft.Hersteller_ID) as Anzahl
from Fahrzeugtyp ft
group by ft.Hersteller_ID
) temp
join Fahrzeugherrsteller fh
on fh.ID = temp.ID
order by Name;

-- so geht es auch
select name, land, count(*) as anzahl
from fahrzeugtyp inner join
fahrzeugherrsteller
    on fahrzeugtyp.Hersteller_ID = fahrzeugherrsteller.id
group by fahrzeugtyp.Hersteller_ID
```

## 5. Lösung zu Übung 5

```
-- mysql

select YEAR(sf.Datum) as Jahr,
       SUBSTRING(Kennzeichen from 1 for locate('-', Kennzeichen)-1 ) as Kreis, COUNT(*)

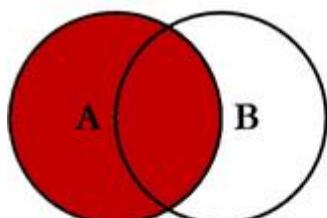
from Zuordnung_SF_FZ zu
right join Fahrzeug fz on fz.ID = zu.Fahrzeug_ID
inner join Schadensfall sf on sf.ID = zu.Schadensfall_ID
where zu.Schuldanteil > 50
group by 1, 2
order by 1, 2;

-- firebird ???
select extract(YEAR from Datum) as Jahr,
       SUBSTRING(Kennzeichen from 1 for POSITION('-', Kennzeichen)-1 ) as Kreis,
       COUNT(*)
from Zuordnung_SF_FZ zu
right join Fahrzeug fz on fz.ID = zu.Fahrzeug_ID
inner join Schadensfall sf on sf.ID = zu.Schadensfall_ID
where zu.Schuldanteil > 50
group by 1, 2
order by 1, 2;
```

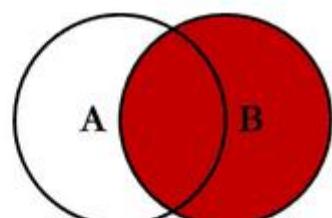
Erläuterungen: Die Tabelle der Zuordnungen ist kleiner als die diejenige der Fahrzeuge, und darauf bezieht sich die WHERE-Bedingung; deshalb ist sie als Haupttabelle am sinnvollsten. Wegen der Kennzeichen benötigen wir einen JOIN auf die Tabelle Fahrzeug. Wegen des Datums des Schadensfalls für die Gruppierung nach Jahr benötigen wir einen JOIN auf die Tabelle Schadensfall.

## 6.12 Joins

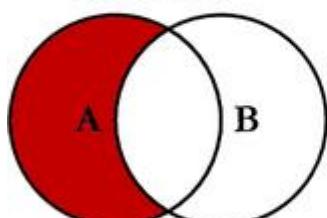
# SQL JOINS



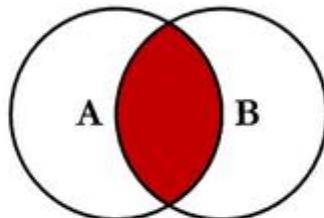
```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



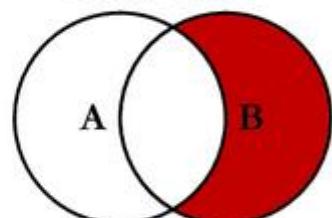
```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

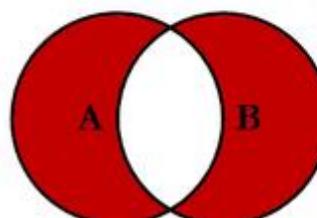
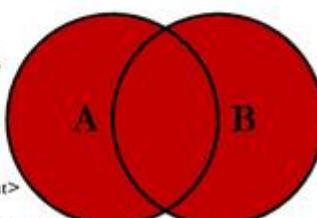


```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```



```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffett, 2008

Ein besonderes Merkmal von relationalen Datenbanken ist, dass die Informationen fast immer über mehrere Tabellen verteilt sind und bei Abfragen in der Ergebnismenge zusammengeführt werden müssen.

Bitte beachten Sie bei allen SELECT-Befehlen, die mehrere Tabellen verwenden:

- Wenn ein Spaltenname in Bezug auf den gesamten SQL-Befehl eindeutig ist, genügt dieser Name.
- Wenn ein Spaltenname mehrfach vorkommt (wie ID), dann muss der Tabellename vorangestellt werden; der Spaltenname wird nach einem Punkt angefügt.

### SELECT

```
Personalnummer as MitNr,
Name, Vorname,
Dienstwagen.ID, Kennzeichen, Fahrzeugtyp_ID as Typ
FROM Mitarbeiter, Dienstwagen;
```

- Wegen der Übersichtlichkeit wird die Tabelle meistens auch dann bei jeder Spalte angegeben, wenn es wegen der ersten Regel nicht erforderlich wäre.
- Anstelle des Namens einer Tabelle kann überall auch ein Tabellen-Alias benutzt werden; dieser muss einmal hinter ihrem Namen (in der FROM- oder in der JOIN-Klausel) angegeben werden.

**SELECT**

```
mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi, Dienstwagen dw;
```

**Verknüpfung über WHERE – EQUI JOIN**

Beim einfachsten Verfahren, mehrere Tabellen gleichzeitig abzufragen, stehen alle Tabellen in der FROM-Klausel; die WHERE-Klausel enthält neben den Auswahlbedingungen auch Bedingungen zur Verknüpfung der Tabellen.

**JOINS - der moderne Weg**

Beim „modernen“ Weg, mehrere Tabellen in einer gemeinsamen Abfrage zu verknüpfen, wird jede Tabelle in einer JOIN-Klausel aufgeführt; der ON-Parameter enthält die Verknüpfungsbedingung. Die WHERE-Klausel enthält nur die Auswahlbedingungen.

**OUTER JOIN - auch null-Werte zurückgeben**

Bei Abfragen mit einem „einfachen“ JOIN werden nicht alle Datensätze aufgeführt. Zeilen, zu denen es in der einen oder anderen Tabelle keine Verknüpfung gibt, fehlen im Ergebnis. Mit einem OUTER JOIN können auch solche „fehlenden“ Zeilen aufgeführt werden.

**Weitere Möglichkeiten von JOIN**

Als SELF JOIN wird eine Tabelle mit sich selbst verknüpft. Oft kommt es vor, dass man die Daten aus einer Tabelle erst bearbeiten möchte, bevor man sie mit einer anderen Tabelle verknüpft. Dazu gibt es die Möglichkeit einer „Inline-View“.

**6.12.1 WHERE - EQUI JOIN**

Der einfachste Weg, Tabellen zu verknüpfen, ist ein Befehl wie der folgende, in dem verschiedene Spalten aus zwei Tabellen zusammengefasst werden. Aber das Ergebnis sieht reichlich seltsam aus.

```
select mi.Personalnummer as MitNr, mi.Name, mi.Vorname,
dw.ID as DIW,
dw.Kennzeichen, dw.Fahrzeugtyp_ID as Typ
FROM Mitarbeiter mi, Dienstwagen dw;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	Typ
10001	Müller	Kurt	1	DO-WB 421	14
10002	Schneider	Daniela	1	DO-WB 421	14
20001	Meyer	Walter	1	DO-WB 421	14
20002	Schmitz	Michael	1	DO-WB 421	14
30001	Wagner	Gaby	1	DO-WB 421	14
30002	Feyerabend	Werner	1	DO-WB 421	14
40001	Langmann	Matthias	1	DO-WB 421	14

```

40002 Peters Michael 1 DO-WB 421 14
/* usw. */
10001 Müller Kurt 2 DO-WB 422 14
10002 Schneider Daniela 2 DO-WB 422 14
20001 Meyer Walter 2 DO-WB 422 14
20002 Schmitz Michael 2 DO-WB 422 14
/* usw. */

```

Tatsächlich erzeugt dieser Befehl das „kartesische Produkt“ der beiden Tabellen: Jeder Datensatz der einen Tabelle wird (mit den gewünschten Spalten) mit jedem Datensatz der anderen Tabelle verbunden. Das sieht also so aus, als wenn alle Dienstwagen zu jedem Mitarbeiter gehören würden, was natürlich Quatsch ist. Diese Variante ist also in aller Regel sinnlos (wenn auch syntaktisch korrekt).

Sinnvoll wird die vorstehende Abfrage durch eine kleine Ergänzung. Was will man denn eigentlich wissen?

Hole Spalten der Tabelle Mitarbeiter  
 sowie Spalten der Tabelle Dienstwagen  
 wobei die Mitarbeiter\_ID eines Dienstwagens gleich ist  
 der ID eines Mitarbeiters

Die obige Abfrage muss nur minimal erweitert werden:

```

SELECT mi.Personalnummer AS MitNr,  

  mi.Name, mi.Vorname,  

  dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ  

FROM Mitarbeiter mi, Dienstwagen dw  

WHERE dw.Mitarbeiter_ID = mi.ID  

order by MitNr;

```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYP
100001	Grosser	Horst	10	DO-WB 4210	14
10001	Müller	Kurt	1	DO-WB 421	14
110001	Eggert	Louis	11	DO-WB 4211	14
120001	Carlsen	Zacharias	12	DO-WB 4212	14
20001	Meyer	Walter	2	DO-WB 422	14
30001	Wagner	Gaby	3	DO-WB 423	14
40001	Langmann	Matthias	4	DO-WB 424	14
50001	Pohl	Helmut	5	DO-WB 425	14
50002	Braun	Christian	14	DO-WB 352	2
50003	Polovic	Frantisek	15	DO-WB 353	3
50004	Kalman	Aydin	16	DO-WB 354	4
/* usw. */					

In der gleichen Weise können auch mehr als zwei Tabellen verknüpft werden.

## Aufgabe

Gesucht wird für jeden Fahrzeugherrsteller (mit Angabe von ID und Name) und jedes Jahr die Summe der Schadenshöhe aus der Tabelle Schadensfall.

```
SELECT fh.ID AS Hersteller_ID,
fh.Name AS Hersteller_Name,
EXTRACT(YEAR FROM sf.Datum) AS Jahr,
SUM(sf.Schadenshoehe) AS Schadenssumme
FROM Schadensfall sf, Zuordnung_SF_FZ zu,
Fahrzeug fz, Fahrzeugtyp ft, Fahrzeugherrsteller fh
WHERE sf.ID = zu.Schadensfall_ID
AND fz.ID = zu.Fahrzeug_ID
AND ft.ID = fz.Fahrzeugtyp_ID
AND fh.ID = ft.Hersteller_ID
GROUP BY Hersteller_ID, Hersteller_Name, Jahr
ORDER BY Jahr, Hersteller_ID;
```

Je mehr Kombinationen benötigt werden, desto unübersichtlicher wird diese Konstruktion. Dabei enthält die WHERE-Klausel bisher nur die Verknüpfungen zwischen den Tabellen, aber noch keine Suchbedingungen wie hier:

```
select ... from ... where ...
and Jahr in [2006, 2007, 2008]
and fhe.Land in ['Schweden', 'Norwegen', 'Finnland']
order by Jahr, Hersteller_ID;
```

Das führt außerdem dazu, dass die WHERE-Klausel sachlich gewünschte Suchbedingungen und logisch benötigte Verknüpfungsbedingungen vermischt.

## Übungen

Bei den folgenden Abfragen beziehen wir uns auf den Bestand der Beispieldatenbank im „Anfangszustand“: die Tabellen Versicherungsvertrag, Fahrzeug, Mitarbeiter mit jeweils etwa 28 Einträgen und Versicherungsnehmer mit etwa 26 Einträgen.

### Aufgaben

- Erstellen Sie eine Abfrage zur Tabelle Versicherungsvertrag, die nur die wichtigsten Informationen (einschließlich der IDs auf andere Tabellen) enthält. Wie viele Einträge zeigt die Ergebnismenge an?
- Erweitern Sie die Abfrage von Aufgabe 1, sodass anstelle der Versicherungsnehmer\_ID dessen Name und Vorname angezeigt werden, und verzichten Sie auf eine WHERE-Klausel. Wie viele Einträge zeigt die Ergebnismenge an?
- Erweitern Sie die Abfrage von Aufgabe 2, sodass anstelle der Fahrzeug\_ID das Kennzeichen und anstelle der Mitarbeiter\_ID dessen Name und Vorname angezeigt werden, und verzichten Sie auf eine WHERE-Klausel. Wie viele Einträge zeigt die Ergebnismenge an?
- Erweitern Sie die Abfrage von Aufgabe 2, sodass Name und Vorname des Versicherungsnehmers genau zu einem jeden Vertrag passen. Wie viele Einträge zeigt die Ergebnismenge an?
- Erweitern Sie die Abfrage von Aufgabe 3, sodass Name und Vorname des Mitarbeiters sowie das Fahrzeug-Kennzeichen genau zu einem jeden Vertrag passen. Wie viele Einträge zeigt die Ergebnismenge an?
- Erweitern Sie die Abfrage von Aufgabe 5, sodass die ausgewählten Zeilen den folgenden Bedingungen entsprechen:
  - Es geht ausschließlich um Eigene Kunden.
  - Vollkasko-Verträge sollen immer angezeigt werden, ebenso Fahrzeuge aus dem Kreis Recklinghausen ‘RE’.
  - Teilkasko-Verträge sollen angezeigt werden, wenn sie nach 1990 abgeschlossen wurden.
  - Haftpflicht-Verträge sollen angezeigt werden, wenn sie nach 1985 abgeschlossen wurden.

Wie viele Einträge zeigt die Ergebnismenge an?

## Lösungen

- Erstellen Sie eine Abfrage zur Tabelle Versicherungsvertrag, die nur die wichtigsten Informationen (einschließlich der IDs auf andere Tabellen) enthält. Wie viele Einträge zeigt die Ergebnismenge an ?

```
/*Es werden 28 Zeilen angezeigt.*/
SELECT Vertragsnummer, Abschlussdatum, Art,
Versicherungsnehmer_ID, Fahrzeug_ID, Mitarbeiter_ID
from Versicherungsvertrag
```

- Erweitern Sie die Abfrage von Aufgabe 1, sodass anstelle der Versicherungsnehmer\_ID dessen Name und Vorname angezeigt werden, und verzichten Sie auf eine WHERE-Klausel. Wie viele Einträge zeigt die Ergebnismenge an?

```
/*Es werden etwa 728 Zeilen angezeigt.*/
SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,
vn.Name, vn.Vorname,
Fahrzeug_ID,
Mitarbeiter_ID
from Versicherungsvertrag vv, Versicherungsnehmer vn;
```

- Erweitern Sie die Abfrage von Aufgabe 2, sodass anstelle der Fahrzeug\_ID das Kennzeichen und anstelle der Mitarbeiter\_ID dessen Name und Vorname angezeigt werden, und verzichten Sie auf eine WHERE-Klausel. Wie viele Einträge zeigt die Ergebnismenge an?

```
/*Es werden etwa 570 752 Zeilen angezeigt.*/
SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,
vn.Name, vn.Vorname,
fz.Kennzeichen,
mi.Name, mi.Vorname
from Versicherungsvertrag vv, Versicherungsnehmer vn,
Fahrzeug fz, Mitarbeiter mi;
```

- Erweitern Sie die Abfrage von Aufgabe 2, sodass Name und Vorname des Versicherungsnehmers genau zu einem jeden Vertrag passen. Wie viele Einträge zeigt die Ergebnismenge an?

```
/*Es werden etwa 28 Zeilen angezeigt.*/
SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,
vn.Name, vn.Vorname,
Fahrzeug_ID,
Mitarbeiter_ID
from Versicherungsvertrag vv, Versicherungsnehmer vn
where vn.ID = vv.Versicherungsnehmer_ID;
```

- Erweitern Sie die Abfrage von Aufgabe 3, sodass Name und Vorname des Mitarbeiters sowie das Fahrzeug-Kennzeichen genau zu einem jeden Vertrag passen. Wie viele Einträge zeigt die Ergebnismenge an?

```
/*Es werden etwa 28 Zeilen angezeigt.*/
SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,
vn.Name, vn.Vorname,
fz.Kennzeichen,
mi.Name, mi.Vorname
from Versicherungsvertrag vv, Versicherungsnehmer vn,
Fahrzeug fz, Mitarbeiter mi
where vn.ID = vv.Versicherungsnehmer_ID
and fz.ID = vv.Fahrzeug_ID
and mi.ID = vv.Mitarbeiter_ID;
```

- Erweitern Sie die Abfrage von Aufgabe 5, sodass die ausgewählten Zeilen den folgenden Bedingungen entsprechen:
  - Es geht ausschließlich um Eigene Kunden.
  - Vollkasko-Verträge sollen immer angezeigt werden, ebenso Fahrzeuge aus dem Kreis Recklinghausen 'RE'.
  - Teilkasko-Verträge sollen angezeigt werden, wenn sie nach 1990 abgeschlossen wurden.
  - Haftpflicht-Verträge sollen angezeigt werden, wenn sie nach 1985 abgeschlossen wurden.

Wie viele Einträge zeigt die Ergebnismenge an?

```
/*Es werden etwa 19 Zeilen angezeigt. Die OR-Verknüpfungen
können
teilweise auch mit CASE geschrieben werden. */

SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,
        vn.Name, vn.Vorname,
        fz.Kennzeichen,
        mi.Name, mi.Vorname
from Versicherungsvertrag vv, Versicherungsnehmer vn,
Fahrzeug fz, Mitarbeiter mi
where vn.ID = vv.Versicherungsnehmer_ID
and fz.ID = vv.Fahrzeug_ID
and mi.ID = vv.Mitarbeiter_ID
and vn.Eigener_kunde = 'J'
and ( ( vv.Art = 'HP' and vv.Abschlussdatum > '31.12.1985'
)
or ( vv.Art = 'TK' and vv.Abschlussdatum > '31.12.1990' )
OR ( vv.Art = 'VK' )
or ( fz.Kennzeichen STARTING WITH 'RE-' ) );
```

## 6.12.2 JOIN

Um Tabellen sinnvoll miteinander zu verknüpfen (= verbinden, engl. join), wurde die JOIN-Klausel für den SELECT-Befehl mit folgender Syntax eingeführt.

```
SELECT <spaltenliste>
FROM <haupttabelle>
[<join-typ>] JOIN <verknüppte tabelle> ON <bedingung>
```

Als <join-typ> stehen zur Verfügung:

- **[INNER] JOIN**, auch Equi-Join genannt, ist eine Verknüpfung „innerhalb“ zweier Tabellen, bei dem ein Wert in beiden Tabellen vorhanden ist.
- **OUTER JOIN** bezeichnet Verknüpfungen, bei denen auch Datensätze liefert werden, für die eine Vergleichsbedingung nicht erfüllt ist.
- **LEFT JOIN, RIGHT JOIN, FULL JOIN** bezeichnen Spezialfälle von OUTER JOIN, je nachdem in welcher Tabelle ein gesuchter Wert fehlt.

Als <bedingung> wird normalerweise nur eine Übereinstimmung (also eine Gleichheit) zwischen zwei Tabellen geprüft, auch wenn jede Kombination von Bedingungen erlaubt ist.

Genauer: es geht um die Gleichheit von Werten je einer Spalte in zwei Tabellen. Auch mehrere Verknüpfungen sind möglich, entweder direkt hintereinander:

```
SELECT <spaltenliste>
FROM <haupttabelle>
[<join-typ>] JOIN <zusatztabelle1> ON <bedingung1>
[<join-typ>] JOIN <zusatztabelle2> ON <bedingung2>
[<join-typ>] JOIN <zusatztabelle3> ON <bedingung3>
```

oder durch Klammern gegliedert:

```
SELECT <spaltenliste>
FROM <haupttabelle>
[<join-typ>] JOIN
( <zusatztabelle1>
[<join-typ>] JOIN
( <zusatztabelle2>
[<join-typ>] JOIN <zusatztabelle3> ON <bedingung3>
) ON <bedingung2>
) ON <bedingung1>
```

Bitte beachten Sie dabei genau, wo und wie die Klammern und die dazugehörigen ON-Bedingungen gesetzt werden. Beide Varianten können unterschiedliche Ergebnisse liefern – abhängig vom JOIN-Typ und dem Zusammenhang zwischen den Tabellen.

### 6.12.2.1 INNER JOIN

Der INNER JOIN gibt lediglich die Datensätze aus, bei denen die Werte in den Verknüpfungsspalten übereinstimmen, d.h. er bezieht sich auf die Schnittmenge

#### Aufgabe

Zeige alle Mitarbeiter mit den dazugehörigen Dienstwagen

The screenshot shows the MySQL Workbench interface with two tables defined:

- dienstwagen** table columns: ID (INT(11)), Kennzeichen (VARCHAR(10)), Farbe (VARCHAR(30)), Fahrzeugtyp\_ID (INT(11)), and Mitarbeiter\_ID (INT(11)).
- mitarbeiter** table columns: ID (INT(11)), Personalnummer (VARCHAR(10)), Name (VARCHAR(30)), Vorname (VARCHAR(30)), and several other columns listed as "7 more...".

A join query is displayed in the center:

```
SELECT dw.Kennzeichen, dw.Mitarbeiter_ID, mi.ID, mi.Name
FROM Mitarbeiter mi JOIN Dienstwagen dw
ON dw.Mitarbeiter_ID = mi.ID
ORDER BY mi.Personalnummer;
```

Below the tables are their respective data grids:

Kennzeichen	Mitarbeiter_ID
DO-WB 421	1
DO-WB 422	3
DO-WB 423	5
DO-WB 424	7
DO-WB 425	9
DO-WB 426	13
DO-WB 427	15
DO-WB 428	17
DO-WB 429	21
DO-WB 4210	23
DO-WB 4211	25
DO-WB 4212	27
DO-WB 422	3
DO-WB 423	5
DO-WB 424	7
DO-WB 425	9
DO-WB 352	10
DO-WB 353	11
DO-WB 354	12
DO-WB 426	13
DO-WB 427	15
DO-WB 428	17
DO-WB 382	18
DO-WB 383	19
DO-WB 384	20
DO-WB 429	21

ID	Name
1	Müller
2	Schneider
3	Meyer
4	Schmitz
5	Wagner
6	Feyerabend
7	Langmann
8	Peters
9	Pohl
10	Braun
11	Polovic
12	Kalman
13	Aagenu
14	Pinkart
15	Olschewski
16	Nordmann
17	Schindler
18	Aliman
19	Langer
20	Kolic
21	Janssen

```
SELECT mi.Personalnummer as MitNr, mi.Name, mi.Vorname,
       dw.ID, dw.Kennzeichen, dw.Fahrzeugtyp_ID as Typ
  FROM Mitarbeiter mi JOIN Dienstwagen dw
    ON dw.Mitarbeiter_ID = mi.ID
ORDER BY MitNr;
```

Er kann mit beliebigen WHERE-Statements erweitert werden.

## Aufgabe

Zeige alle Mitarbeiter und deren Autokennzeichen, die als Dienstwagen einen Mercedes fahren.

```

/*Vermischung von join und where; geht, ist aber nicht schön*/
select mi.Personalnummer as MitNr,
mi.Name, mi.Vorname,
dw.ID, dw.Kennzeichen, dw.Fahrzeugtyp_ID as Typ
from Mitarbeiter mi
join Dienstwagen dw
on mi.ID = dw.Mitarbeiter_ID
and dw.Fahrzeugtyp_ID in ( SELECT ft.ID
from Fahrzeugtyp ft
join Fahrzeugherrsteller fh
on ft.Hersteller_ID = fh.ID
and fh.Name = 'Mercedes-Benz' );

/*Trennung von join und where */
select mi.Personalnummer as MitNr, vmi.Name, mi.Vorname,
dw.ID, dw.Kennzeichen, dw.Fahrzeugtyp_ID as Typ
from Mitarbeiter mi join Dienstwagen dw
on mi.ID = dw.Mitarbeiter_ID
where dw.Fahrzeugtyp_ID in ( SELECT ft.ID
from Fahrzeugtyp ft join Fahrzeugherrsteller fh
on ft.Hersteller_ID = fh.ID
where fh.Name = 'Mercedes-Benz') ;

```

In den bisherigen Beispielen können die beiden Tabellen ohne weiteres vertauscht werden, da lediglich die Schnitmenge gesucht wird:

```

select mi.Personalnummer as MitNr, mi.Name, mi.Vorname,
dw.ID, dw.Kennzeichen, dw.Fahrzeugtyp_ID as Typ
from Dienstwagen dw
join Mitarbeiter mi
on mi.ID = dw.Mitarbeiter_ID
where dw.Fahrzeugtyp_ID in ( SELECT ft.ID
from Fahrzeugtyp ft
join Fahrzeugherrsteller fh
on ft.Hersteller_ID = fh.ID
where fh.Name = 'Mercedes-Benz')
and mi.Name like 'M%';

```

Die Haupttabelle kann nach folgenden Überlegungen gewählt werden:

- Es sollte die Tabelle sein, die die „wichtigste“ bei der Abfrage ist.
- Es sollte diejenige mit den größten Einschränkungen sein, da dies die Abfrage besonders stark beschleunigt.

## Verknüpfungen über mehr als zwei Tabellen

Wenn mehrere Tabellen miteinander verbunden werden, werden die einzelnen JOIN-Anweisungen und deren ON-Bedingungen hintereinander in der passenden Verknüpfung angeordnet.

### Beispiel

Gesucht wird für jeden Fahrzeugherrsteller (mit Angabe von ID und Name) und jedes Jahr die Summe der Schadenshöhe aus der Tabelle Schadensfall.

```
SELECT fh.ID AS Hersteller_ID,  
fh.Name AS Hersteller_Name,  
EXTRACT(YEAR FROM sf.Datum) AS Jahr,  
SUM(sf.Schadenshoehe) AS Schadenssumme  
FROM Schadensfall sf  
JOIN Zuordnung_SF_FZ zu ON sf.ID = zu.Schadensfall_ID  
JOIN Fahrzeug fz ON fz.ID = zu.Fahrzeug_ID  
JOIN Fahrzeugtyp ft ON ft.ID = fz.Fahrzeugtyp_ID  
JOIN Fahrzeugherrsteller fh ON fh.ID = ft.Hersteller_ID  
GROUP BY Hersteller_ID, Hersteller_Name, Jahr  
ORDER BY Jahr, Hersteller_ID;
```

## Übungen zu Inner Join

1. Welche der folgenden Aussagen sind wahr, welche falsch, welche sinnvoll?
  1. Der INNER JOIN liefert das kartesische Produkt zwischen den Tabellen.
  2. LEFT JOIN ist ein Spezialfall von OUTER JOIN.
  3. Für einen JOIN ist dies eine zulässige Verknüpfungsbedingung:

```
ON Fahrzeug.ID >= Versicherungsvertrag.Fahrzeug_ID
```

4. Eine Einschränkung auf die mit JOIN verknüpfte Tabelle gehört in die ON- Klausel:

```
... FROM Zuordnung_SF_FZ zu JOIN Schadensfall sf  

ON sf.ID = zu.Schadensfall_ID AND YEAR(sf.Datum) = 2008;
```

2. Erläutern Sie, was am folgenden Befehl falsch oder äußerst ungünstig ist. Es handelt sich um diese Abfrage:

```
SELECT Datum, SUBSTRING(Ort from 1 for 30) as Ort,  

       Schadenshoehe, zu.Schadenshoehe, fz.Kennzeichen,  

       Vertragsnummer as Vertrag, Abschlussdatum, Art,  

       vn.Name as VN-Name, vn.Vorname as VN-Vorname  

  from Schadensfall sf  

    join Zuordnung_SF_FZ zu on ID = zu.Schadensfall_ID  

    join Fahrzeug fz on ID = zu.Fahrzeug_ID  

    join Versicherungsnehmer vn on ID = vv.Versicherungs-  

        nehmer_ID  

    join Versicherungsvertrag vv on vv.Fahrzeug_ID =  

        zu.Fahrzeug_ID  

   where YEAR (Datum) = 2008  

  order by Schadensfall_ID, Fahrzeug_ID;
```

Gesucht sind die Schadensfälle des Jahres 2008. Zu jedem Schadensfall sind die beteiligten Fahrzeuge, der Schadensanteil sowie die Versicherungsdaten des Fahrzeugs (einschließlich Name des Halters) anzugeben.

3. Erstellen Sie eine Abfrage zur Tabelle Versicherungsvertrag mit den wichtigsten Informationen (einschließlich der IDs auf andere Tabellen). Beim Versicherungsnehmer sollen dessen Name und Vorname angezeigt werden. Es werden nur Verträge ab 1990 gesucht.

4. Erweitern Sie die obige Abfrage, sodass Name und Vorname des Mitarbeiters sowie das Fahrzeug-Kennzeichen eines jeden Vertrags angezeigt werden.
5. Ändern Sie die obige Abfrage so, dass die ausgewählten Zeilen den folgenden Bedingungen entsprechen:
  - o Es geht ausschließlich um Eigene Kunden.
  - o Vollkasko-Verträge sollen immer angezeigt werden, ebenso Fahrzeuge aus dem Kreis Recklinghausen 'RE'.
  - o Teilkasko-Verträge sollen angezeigt werden, wenn sie nach 1990 abgeschlossen wurden.
  - o Haftpflicht-Verträge sollen angezeigt werden, wenn sie nach 1985 abgeschlossen wurden.



## Lösungen zu Inner Join

### Lösung zu Übung 1

1. Falsch; es liefert einen Teil des kartesischen Produkts, der durch die ON- Bedingung bestimmt wird.
2. Richtig.
3. Diese Bedingung ist zulässig, aber nicht sinnvoll. JOIN-ON passt in der Regel nur für Gleichheiten.
4. Diese Bedingung ist zulässig. Besser ist es aber, eine Einschränkung der Auswahl in die WHERE-Klausel zu setzen.

### Lösung zu Übung 2

Richtig ist beispielsweise die folgende Version. Als Haupttabelle wurde wegen der WHERE-Klausel die Tabelle Schadensfall gewählt; wegen der Reihenfolge der Verknüpfungen wäre auch Zuordnung\_SF\_FZ als Haupttabelle geeignet.

```
SELECT sf.Datum, SUBSTRING(sf.Ort from 1 for 30) as Ort,
sf.Schadenshoehe,
zu.Schadenshoehe as Teilschaden,
fz.Kennzeichen,
vv.Vertragsnummer as Vertrag, vv.Abschlussdatum, vv.Art,
vn.Name as VN_Name, vn.Vorname as VN_Vorname
from Schadensfall sf
join Zuordnung_SF_FZ zu on sf.ID = zu.Schadensfall_ID
join Fahrzeug fz on fz.ID = zu.Fahrzeug_ID
join Versicherungsvertrag vv on fz.ID = vv.Fahrzeug_ID
join Versicherungsnehmer vn on vn.ID = vv.Versicherungsnehmer_ID
where EXTRACT(YEAR from sf.Datum) = 2008
order by zu.Schadensfall_ID,
```

Die Variante aus der Aufgabenstellung enthält folgende Problemstellen:

- Zeile 1: Der Tabellen-Alias sf fehlt bei Schadenshoehe und bei Ort. Bei Datum fehlt er auch, aber das ist kein Problem, weil es diese Spalte nur bei dieser Tabelle gibt.
- Zeile 2: Diese Spalte sollte einen Spalten-Alias bekommen wegen der abweichenden Bedeutung zu sf.Schadenshoehe.
- Zeile 4: Es ist schöner, auch hier mit einem Tabellen-Alias zu arbeiten.
- Zeile 5: Der Bindestrich in der Bezeichnung des Spalten-Alias wird nicht bei allen DBMS akzeptiert.

- Zeile 7, 8, 9: Zur Spalte ID ist jeweils die Tabelle anzugeben, ggf. mit dem Alias, Die JOIN-ON-Bedingung bezieht sich nicht automatisch auf diese Spalte und diese Tabelle.
- Zeile 9, 10: In Zeile 9 ist die Tabelle Versicherungsvertrag vv noch nicht bekannt.  
Wegen der Verknüpfungen ist zuerst Zeile 10 zu verwenden, danach Zeile 9. Die Verknüpfung über vv.Fahrzeug\_ID = zu.Fahrzeug\_ID ist nicht glücklich (wenn auch korrekt); besser ist der Bezug auf die direkt zugeordnete Tabelle Fahrzeug und deren PrimaryKey, nämlich ID.
- Zeile 11: Es ist klarer, auch hier den Tabellen-Alias sf zu verwenden.
- Zeile 12: Der Tabellen-Alias zu fehlt bei beiden Spalten. Bei Fahrzeug\_ID ist er erforderlich (doppelte Verwendung bei vv), bei Schadensfall\_ID sinnvoll.

### Lösung zu Übung 3

```
SELECT Vertragsnummer, Abschlussdatum, Art,
Name, Vorname,
Fahrzeug_ID,
Mitarbeiter_ID
from Versicherungsvertrag vv
join Versicherungsnehmer vn on vn.ID = vv.Versicherungsnehmer_ID
where vv.Abschlussdatum >= '01.01.1990';
```

### Lösung zu Übung 4

```
SELECT vv.Vertragsnummer as Vertrag, vv.Abschlussdatum,
vv.Art,
vn.Name as VN_Name, vn.Vorname as VN_Vorname,
fz.Kennzeichen,
mi.Name as MI_Name, mi.Vorname as MI_Vorname
from Versicherungsvertrag vv
join Versicherungsnehmer vn on vn.ID = vv.Versicherungsnehmer_ID
join Fahrzeug fz on fz.ID = vv.Fahrzeug_ID
join Mitarbeiter mi on mi.ID = vv.Mitarbeiter_ID
where vv.Abschlussdatum >= '01.01.1990';
```

## Lösung zu Übung 5

```
SELECT vv.Vertragsnummer as Vertrag, vv.Abschlussdatum,  
vv.Art,  
vn.Name as VN_Name, vn.Vorname as VN_Vorname,  
fz.Kennzeichen,  
mi.Name as MI_Name, mi.Vorname as MI_Vorname  
from Versicherungsvertrag vv  
join Versicherungsnehmer vn on vn.ID = vv.Versicherungsneh-  
mer_ID  
join Fahrzeug fz on fz.ID = vv.Fahrzeug_ID  
join Mitarbeiter mi on mi.ID = vv.Mitarbeiter_ID  
where vn.Eigener_kunde = 'J'  
and ( ( vv.Art = 'HP' and vv.Abschlussdatum > '31.12.1985'  
)  
or ( vv.Art = 'TK' and vv.Abschlussdatum > '31.12.1990' )  
OR ( vv.Art = 'VK' )  
or ( fz.Kennzeichen LIKE 'RE-%' ) );
```

### 6.12.2.2 OUTER JOIN

Bei den Abfragen im vorigen Kapitel nach "alle Mitarbeiter und ihre Dienstwagen" werden nicht alle Mitarbeiter aufgeführt, weil in der Datenbank nicht für alle Mitarbeiter ein Dienstwagen registriert ist. Ebenso gibt es einen Dienstwagen, der keinem bestimmten Mitarbeiter zugeordnet ist. Mit einem OUTER JOIN werden auch Mitarbeiter ohne Dienstwagen oder Dienstwagen ohne Mitarbeiter aufgeführt.

Die Syntax entspricht derjenigen von JOIN allgemein. Wegen der speziellen Bedeutung sind die Tabellen nicht gleichberechtigt, sondern werden begrifflich unterschieden:

```
SELECT <spaltenliste>
FROM <linke tabelle>
[<join-typ>] JOIN <rechte tabelle> ON <bedingung>
```

Das Wort OUTER kann entfallen und wird üblicherweise nicht benutzt, weil durch die Begriffe LEFT, RIGHT, FULL bereits ein OUTER JOIN gekennzeichnet wird. Die Begriffe <linke tabelle> und <rechte tabelle> beziehen sich auf die beiden Tabellen bezüglich der normalen Lesefolge: Wir lesen von links nach rechts, also ist die unter FROM genannte Tabelle die <linke Tabelle> (bisher <Haupttabelle> genannt) und die unter JOIN genannte Tabelle die <rechte Tabelle> (bisher <Zusatztabelle> genannt). Bei Verknüpfungen mit mehreren Tabellen ist ebenfalls die unter JOIN genannte Tabelle die <rechte Tabelle>; die unmittelbar vorhergehende Tabelle ist die <linke Tabelle>.

#### 6.12.2.2.1 LEFT JOIN

Dieser JOIN liefert alle Datensätze der linken Tabelle, ggf. unter Berücksichtigung der WHERE-Klausel. Aus der rechten Tabelle werden nur diejenigen Datensätze übernommen, die nach der Verknüpfungsbedingung passen.

```
SELECT <spaltenliste>
FROM <linke Tabelle>
LEFT [OUTER] JOIN <rechte Tabelle> ON <bedingung>;
```

#### Aufgabe

Hole alle Mitarbeiter und (sofern vorhanden) die Angaben zum Dienstwagen.

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi
LEFT JOIN Dienstwagen dw ON dw.Mitarbeiter_ID = mi.ID;
MITNR NAME VORNAME DIW KENNZEICHEN TYP
30001 Wagner Gaby 3 DO-WB 423 14
30002 Feyerabend Werner
40001 Langmann Matthias 4 DO-WB 424 14
40002 Peters Michael
50001 Pohl Helmut 5 DO-WB 425 14
50002 Braun Christian 14 DO-WB 352 2
50003 Polovic Frantisek 15 DO-WB 353 3
50004 Kalman Aydin 16 DO-WB 354 4
60001 Aagenau Karolin 6 DO-WB 426 14
60002 Pinkart Petra
```

Beim Vertauschen der beiden Tabellen erhält man alle Dienstwagen und dazu die passenden Mitarbeiter.

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Dienstwagen dw
LEFT JOIN Mitarbeiter mi ON dw.Mitarbeiter_ID = mi.ID;
MITNR NAME VORNAME DIW KENNZEICHEN TYP
80001 Schindler Christina 8 DO-WB 428 14
90001 Janssen Bernhard 9 DO-WB 429 14
10001 Eggert Louis 11 DO-WB 4211 14
120001 Carlsen Zacharias 12 DO-WB 4212 14
111 16
13 DO-WB
50002 Braun Christian 14 DO-WB 352 2
50003 Polovic Frantisek 15 DO-WB 353 3
50004 Kalman Aydin 16 DO-WB 354 4
```

### 6.12.2.2.2 RIGHT OUTER JOIN

Dieser JOIN liefert alle Datensätze der rechten Tabelle, ggf. unter Berücksichtigung der WHERE-Klausel. Aus der linken Tabelle werden nur diejenigen Datensätze übernommen, die nach der Verknüpfungsbedingung passen.

```
SELECT <spaltenliste>
FROM <linke Tabelle>
RIGHT [OUTER] JOIN <rechte Tabelle> ON <bedingung>;
```

Für unser Beispiel „Mitarbeiter und Dienstwagen“ sieht das dann so aus:

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi
RIGHT JOIN Dienstwagen dw ON dw.Mitarbeiter_ID = mi.ID;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYP
80001	Schindler	Christina	8	DO-WB 428	14
90001	Janssen	Bernhard	9	DO-WB 429	14
100001	Grosser	Horst	10	DO-WB 4210	14
110001	Eggert	Louis	11	DO-WB 4211	14
120001	Carlsen	Zacharias	12	DO-WB 4212	14
			13	DO-WB 111	16
50002	Braun	Christian	14	DO-WB 352	2
50003	Polovic	Frantisek	15	DO-WB 353	3
50004	Kalman	Aydin	16	DO-WB 354	4

Das Ergebnis ist das Gleich wie beim LEFT JOIN. Bei genauerem Überlegen wird klar: Beim LEFT JOIN gibt es alle Datensätze der linken Tabelle mit Informationen der rechten Tabelle; nun haben wir die beiden Tabellen vertauscht. Beim RIGHT JOIN werden alle Datensätze der rechten Tabelle mit Daten der linken Tabelle verknüpft; das entspricht diesem Beispiel.

### 6.12.2.2.3 FULL OUTER JOIN

Dieser JOIN liefert alle Datensätze beider Tabellen, ggf. unter Berücksichtigung der WHERE-Klausel. Wenn Datensätze nach der Verknüpfungsbedingung zusammenpassen, werden sie in einer Zeile angegeben; wo es keinen „Partner“ gibt, wird ein NULL-Wert angezeigt.

```
SELECT <spaltenliste>
FROM <linke Tabelle>
FULL [OUTER] JOIN <rechte Tabelle> ON <bedingung>;
```

Für unser Beispiel sieht das dann so aus:

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi
FULL JOIN Dienstwagen dw ON dw.Mitarbeiter_ID = mi.ID;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	Typ
100001	Grosser	Horst	10	DO-WB 4210	14
110001	Eggert	Louis	11	DO-WB 4211	14
120001	Carlsen	Zacharias	12	DO-WB 4212	14
			13	DO-WB 111	16
50002	Braun	Christian	14	DO-WB 352	2
50003	Polovic	Frantisek	15	DO-WB 353	3
50004	Kalman	Aydin	16	DO-WB 354	4
80002	Aliman	Zafer	17	DO-WB 382	2
80003	Langer	Norbert	18	DO-WB 383	3
80004	Kolic	Ivana	19	DO-WB 384	4
10002	Schneider	Daniela			
20002	Schmitz	Michael			
30002	Feyerabend	Werner			
40002	Peters	Michael			

Auch hier wollen wir wieder die beiden Tabellen vertauschen:

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Dienstwagen dw
FULL JOIN Mitarbeiter mi ON dw.Mitarbeiter_ID = mi.ID;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYP
80001	Schindler	Christina	8	DO-WB 428	14
80002	Aliman	Zafer	17	DO-WB 382	2
80003	Langer	Norbert	18	DO-WB 383	3
80004	Kolic	Ivana	19	DO-WB 384	4
90001	Janssen	Bernhard	9	DO-WB 429	14
90002	Hinkel	Martina			
100001	Grosser	Horst	10	DO-WB 4210	14
100002	Friedrichsen	Angelina			
110001	Eggert	Louis	11	DO-WB 4211	14
110002	Deiters	Gisela			
120001	Carlsen	Zacharias	12	DO-WB 4212	14
120002	Baber	Yvonne			
			13	DO-WB 111	16

Bei detaillierterem Vergleich des vollständigen Ergebnisses ergibt sich: Es ist gleich, nur in anderer Reihenfolge. Das sollte nicht mehr verwundern.

#### 6.12.2.2.4 Verknüpfung mehrerer Tabellen

Alle bisherigen Beispiele kranken daran, dass als Typ des Dienstwagens nur die ID angegeben ist. Selbstverständlich möchte man die Typbezeichnung und den Hersteller lesen. Dazu müssen die beiden Tabellen Fahrzeugtyp und Fahrzeugherrsteller eingebunden werden.

Beim INNER JOIN war das kein Problem; beim OUTER JOIN kann es anders aussehen.

#### Beispiel

Erweitern wir dazu die Aufstellung "alle Dienstwagen zusammen mit den zugeordneten Mitarbeitern" um die Angabe zu den Fahrzeugen.

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS TypID,
ft.Bezeichnung as Typ, ft.Hersteller_ID as FheID
FROM Dienstwagen dw
left JOIN Mitarbeiter mi ON dw.Mitarbeiter_ID = mi.ID
join Fahrzeugtyp ft on dw.Fahrzeugtyp_ID = ft.ID;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYPID	TYP	FHEID
100001	Grosser	Horst	10	DO-WB 4210	14	A160	6
110001	Eggert	Louis	11	DO-WB 4211	14	A160	6
120001	Carlsen	Zacharias	12	DO-WB 4212	14	A160	6
			13	DO-WB 111	16	W211	6
50002	Braun	Christian	14	DO-WB 352	2	Golf	1
50003	Polovic	Frantisek	15	DO-WB 353	3	Passat	1
50004	Kalman	Aydin	16	DO-WB 354	4	Kadett	2

Der zweite JOIN wurde nicht genauer bezeichnet, ist also ein INNER JOIN. Das gleiche Ergebnis erhalten wir, wenn wir die Tabelle Fahrzeugtyp ausdrücklich als LEFT JOIN verknüpfen (bitte selbst ausprobieren!). Anders sieht es beim Versuch mit RIGHT JOIN oder FULL JOIN aus:

```
SELECT mi.Personalnummer AS MitNr,
mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS TypID,
ft.Bezeichnung as Typ, ft.Hersteller_ID as FheID
FROM Dienstwagen dw
left JOIN Mitarbeiter mi ON dw.Mitarbeiter_ID = mi.ID
right | full join Fahrzeugtyp ft on dw.Fahrzeugtyp_ID = ft.ID;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYPID	Typ	FHEID
80001	Schindler	Christina	8	DO-WB 428	14	A160	6
90001	Janssen	Bernhard	9	DO-WB 429	14	A160	6
100001	Grosser	Horst	10	DO-WB 4210	14	A160	6
110001	Eggert	Louis	11	DO-WB 4211	14	A160	6
120001	Carlsen	Zacharias	12	DO-WB 4212	14	A160	6
						W204	6
.			13	DO-WB 111	16	W211	6
						Saab 9-3	8
						S40	9
						C30	9

### Erklärung:

Die beiden JOINS stehen sozusagen auf der gleichen Ebene; jede JOIN-Klausel wird für sich mit der Tabelle Dienstwagen verknüpft. An der Verknüpfung zwischen Dienstwagen und Mitarbeiter ändert sich nichts. Aber für die Fahrzeugtypen gilt:

- Das erste Beispiel benutzt einen INNER JOIN, nimmt also für jeden vorhandenen Dienstwagen genau “seinen” Typ.
- Wenn man stattdessen einen LEFT JOIN verwendet, erhält man alle vorhandenen Dienstwagen, zusammen mit den passenden Typen. Das ist faktisch identisch mit dem Ergebnis des INNER JOIN.
- Das zweite Beispiel benutzt einen RIGHT JOIN, das liefert alle registrierten Fahrzeugtypen und (soweit vorhanden) die passenden Dienstwagen.
- Wenn man stattdessen einen FULL JOIN verwendet, erhält man alle Kombinationen von Dienstwagen und Mitarbeitern, zusammen mit allen registrierten Fahrzeugtypen. Das ist faktisch identisch mit dem Ergebnis des RIGHT JOIN.

### 6.12.2.2.5 Wann welcher JOIN

Im untenstehender Grafik sind zwei Tabellen vorhanden: Paare (also Eltern) und Kinder. Es gibt kinderlose Paare, Paare mit Kindern und Waisenkinder. Wir wollen die Eltern und Kinder in Abfragen verknüpfen; bei den Symbolen steht der linke Kreis für die Tabelle Paare und der rechte Kreis für die Tabelle Kinder. Die schwarze Fläche beschreibt die jeweils gewünschte Ergebnismenge.

Schreiben Sie in die links leerstehenden Felder den jeweiligen Join-Begriff; in die rechts stehenden Felder das entsprechende SQL-Statement

		Nur Waisenkinder	
Die zweite Variante – INNER JOIN – kann man auch so ausdrücken:			
LEFT JOIN IS NOT NULL			

Lösung:

LEFT JOIN		Alle Paare und (falls es Kinder gibt) auch diese	SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key
INNER JOIN		Nur Paare, die Kinder haben	SELECT * FROM Paare INNER JOIN Kinder ON Paare.Key = Kinder.Key
RIGHT JOIN		Alle Kinder und (falls es Eltern gibt) auch diese	SELECT * FROM Paare RIGHT JOIN Kinder ON Paare.Key = Kinder.Key
LEFT JOIN IS NULL		Nur Paare, die keine Kinder haben	SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NULL
RIGHT JOIN IS NULL		Nur Waisenkinder	SELECT * FROM Paare RIGHT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Paare.Key IS NULL
FULL JOIN		Alle Paare und alle Kinder	SELECT * FROM Paare FULL JOIN Kinder ON Paare.Key = Kinder.Key
FULL JOIN IS NULL		Alle kinderlosen Paare und alle Waisenkinder	SELECT * FROM Paare FULL JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NULL OR Paare.Key IS NULL
Die zweite Variante – INNER JOIN – kann man auch so ausdrücken:			
LEFT JOIN IS NOT NULL		Alle Paare und (falls es Kinder gibt) auch diese, wobei es ein Kind geben muss	SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NOT NULL

### 6.12.2.3 Auswertungsreihenfolge von joins

Gegeben sei folgende Datenbank:

#### Tabellen

Tabelle Kunden

KndNr	Nachname	Vorname	Strasse	PLZ	Ort
123456	Mustermann	Max	Musterweg 1	12345	Musterstadt
123457	Musterfrau	Katrin	Musterstraße 7	12345	Musterstadt
123458	Müller	Lieschen	Beispielweg 3	23987	Irgendwo
123459	Schmidt	Hans	Hauptstraße 2	98765	Anderswo
123460	Becker	Heinz	Mustergasse 4	12543	Musterdorf

Tabelle Kreditkarten

KartenNr	Firma	KndNr	Ablaufdatum
12345	VISA	123457	05/2011
12346	Mastercard	123459	01/2012
12347	American Express	123459	01/2011
12348	Diners Club	123458	03/2012
12349	VISA	123458	07/2011

Tabelle Bestellungen\_Oktober

KndNr	BestellungsNr	Datum
123456	987654	2009-10-15
123456	987655	2009-10-16
123457	987656	2009-10-16

Tabelle Positionen

PositionsNr	BestellungsNr	Artikel	Anzahl	Preis
10241	987654	CD-Player	2	49.95
10242	987654	DVD-Player	3	59.95
10243	987654	CD xyz	10	15.95
10244	987654	DVD abc	5	9.95
10245	987655	CD-Player	1	51.20
10246	987655	CD xyz extra	20	16.25
10247	987656	DVD-Player	1	64.95

Tabelle Vorteilsclub

KndNr	ClubNr	Kategorie
123458	1214	3
123456	1415	1
123460	1616	1

```
create table kunden(
    kundennr integer not null primary KEY,
    nachname varchar(20),
    vorname varchar(20),
    strasse varchar(20),
    plz varchar(5),
    ort varchar(20)
) engine = innodb;

insert into kunden( kundennr, nachname, vorname, strasse, PLZ,
    ort) values(123456, 'mustermann', 'max', 'musterweg 1',
'12345', 'musterstadt');

insert into kunden( kundennr, nachname, vorname, strasse, PLZ,
    ort) values(123457, 'musterfrau', 'katrin', 'musterstrasse
1', '12345', 'musterstadt');

insert into kunden(kundennr, nachname, vorname, strasse, PLZ,
    ort) values(123458, 'Müller', 'lieschen', 'beispielweg 3',
'23987', 'irgendwo');

insert into kunden( kundennr, nachname, vorname, strasse, PLZ,
    ort) values(123459, 'Schmidt', 'hans', 'hauptstraße 2',
'98765', 'anderswo') ;

insert into kunden( kundennr, nachname, vorname, strasse, PLZ,
    ort) values(123460, 'Becker', 'heinz', 'mustergasse 4',
'12543', 'musterdorf') ;

create table kreditkarten(
    kartennr integer not null,
    firma varchar(20),
    kundennr integer,
    ablaufdatum varchar(20)
) engine=innodb;

insert into kreditkarten(kartennr, firma, kundennr, ablaufdatum)
values(12345, 'Visa', 123457, '05/2011' );

insert into kreditkarten(kartennr, firma, kundennr, ablaufdatum)
values(12346, 'Mastercard', 123459, '01/2012' );

insert into kreditkarten(kartennr, firma, kundennr, ablaufdatum)
values(12347, 'American Express', 123459, '01/2011' );

insert into kreditkarten(kartennr, firma, kundennr, ablaufdatum)
values(12348, 'Diners Club', 123458, '03/2012');
```

```
insert into kreditkarten(kartennr, firma, kundennr, ablaufdatum)
values(12349, 'Visa', 123458, '07/2011');

create table bestellungen_oktober(
kundennr integer,
bestellungsnr integer,
datum date
) engine = innodb;

insert into bestellungen_oktober (kundennr, bestellungsnr,
datum) values(123456, 987654, '2009-10-15');

insert into bestellungen_oktober (kundennr, bestellungsnr,
datum) values(123456, 987655, '2009-10-16');

insert into bestellungen_oktober (kundennr, bestellungsnr,
datum) values(123457, 987656, '2009-10-16');

create table positionen(
positionsnr integer,
bestellungsnummer integer,
artikel varchar(20),
anzahl integer,
preis double
) engine = innodb;

insert into positionen ( positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10241, 987654, 'CD-Player', 2, 49.95);

insert into positionen ( positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10242, 987654, 'DVD-Player', 3, 59.95);

insert into positionen ( positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10243, 987654, 'CD xyz', 10, 19.95);

insert into positionen(positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10244, 987654, 'DVD abc', 5, 9.95);

insert into positionen (positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10245, 987655, 'CD-Player', 1, 51.20);

insert into positionen (positionsnr,bestellungsnummer, artikel ,
anzahl, preis) values(10246, 987655,'CD xyz extra',20, 16.25);

insert into positionen (
positionsnr,bestellungsnummer, artikel, anzahl, preis)
values(10247, 987656, 'DVD-Player', 1, 64.95);
```

```

create table vorteilsclub(
    kndnr INTEGER,
    clubnr integer,
    kategorie integer
) engine = innodb;

insert into vorteilsclub(kndnr, clubnr, kategorie)
    values (123458, 1214, 3);

insert into vorteilsclub(kndnr, clubnr, kategorie)
    values (123456, 1415, 1);

insert into vorteilsclub(kndnr, clubnr, kategorie)
    values (123460, 1616, 1);

```

[http://wiki.selfhtml.org/wiki/Datenbank/Fortgeschrittene\\_Jointtechniken#Die\\_Beispiel Datenbank](http://wiki.selfhtml.org/wiki/Datenbank/Fortgeschrittene_Jointtechniken#Die_Beispiel Datenbank)

1. Sie möchten z.B. für eine spezielle Marketingaktion wissen, welche Kunden (KundenNr) sowohl über eine Mastercard als auch eine Kreditkarte von American Express verfügen.
2. Nun möchten Sie nicht alle möglichen Kombinationen von Kundennummern und zu gehörigen Kreditkarten haben, sondern wissen, wer sowohl eine 'Mastercard' und eine 'American Express' hat. Dazu nutzen Sie die WHERE-Klausel.
3. Im vorhergehenden Abschnitt haben Sie für Ihre Marketingabteilung die Kundennummern der Kunden herausgefunden, die sowohl über eine 'Mastercard' als auch eine 'American Express' verfügen. Nun möchte die Marketingabteilung wissen, um welche Person es sich handelt, d.h. Nachname, Vorname, Anschrift
4. Diesmal benötigt sie die Adressen von allen Kunden, von den Kreditkarteninhabern zusätzlich die Kreditkarteninformationen aber nur, wenn sie Mitglied im Vorteilsclub sind. Ihnen ist klar, dass Sie für die Lösung dieser Aufgabe die Tabellen 'Kunden', 'Kreditkarten' und 'Vorteilsclub' miteinander verknüpfen müssen
5. Ihr Chef will eine Übersicht über die im Monat Oktober bestellten Artikel haben, mit den Detailinformationen zu den Kunden, die diese Bestellungen getätigt haben. Die benötigten Daten verteilen sich auf die Tabellen 'Kunden', 'Bestellungen\_Oktober' und 'Positionen'. Diese drei Tabellen sind über unterschiedliche Spalten miteinander verknüpft: 'Kunden' und 'Bestellungen\_Oktober' über die Spalte 'KndNr', 'Bestellungen\_Oktober' und 'Positionen' über die Spalte 'BestellungsNr'.
6. Nun möchte Ihr Chef eine Übersicht über alle Kunden mit den Artikeln, die diese bestellt haben. Auch Kunden ohne Bestellungen sollen mit Vor- und Nachname aufgeführt werden; diese Spalten sollen vorne stehen.

## Lösung

- Sie möchten z.B. für eine spezielle Marketingaktion wissen, welche Kunden (KundenNr) sowohl über eine Mastercard als auch eine Kreditkarte von American Express verfügen.

```
SELECT KK1.KndNr, KK1.Firma, KK2.KndNr, KK2.Firma
FROM
    Kreditkarten KK1
INNER JOIN
    Kreditkarten KK2
ON
    KK1.KndNr = KK2.KndNr
```

- Nun möchten Sie nicht alle möglichen Kombinationen von Kundennummern und zugehörigen Kreditkarten haben, sondern wissen, wer sowohl eine 'Mastercard' und eine 'American Express' hat. Dazu nutzen Sie die WHERE-Klausel.

```
SELECT
KK1.KndNr, KK1.Firma, KK2.KndNr, KK2.Firma
FROM
    Kreditkarten KK1
INNER JOIN
    Kreditkarten KK2
ON
    KK1.KndNr = KK2.KndNr
WHERE
    KK1.Firma = 'Mastercard' AND KK2.Firma = 'American Express'
```

- Um welche Person handelt es sich

```
SELECT
    KK1.KndNr, Nachname, Vorname, Strasse, PLZ, Ort
FROM (
    Kreditkarten KK1
INNER JOIN
    Kreditkarten KK2
ON
    KK1.KndNr = KK2.KndNr
)
INNER JOIN
    Kunden
ON
    KK1.KndNr = Kunden.KndNr
WHERE
    KK1.Firma = 'Mastercard' AND KK2.Firma = 'American Express'
```

4. Diesmal benötigt sie die Adressen von allen Kunden, von den Kreditkarteninhabern zusätzlich die Kreditkarteninformationen aber nur, wenn sie Mitglied im Vorteilsclub sind

```
SELECT Kunden.KndNr, Nachname, Vorname, Strasse, PLZ, Ort,
       Firma, KartenNr, Ablaufdatum
FROM Kunden
      LEFT JOIN (
        Kreditkarten INNER JOIN Vorteilsclub
        ON Kreditkarten.KndNr = Vorteilsclub.KndNr
      )
      ON Kunden.KndNr = Kreditkarten.KndNr
```

5. Ihr Chef will eine Übersicht über die im Monat Oktober bestellten Artikel haben, mit den Detailinformationen zu den Kunden

```
SELECT Artikel, Anzahl, Preis, Datum, Nachname, Vorname,
       Strasse, PLZ, Ort
FROM Positionen
      INNER JOIN ( Bestellungen_Oktober
        INNER JOIN Kunden
        ON Bestellungen_Oktober.KndNr = Kunden.KndNr
      )
      ON Positionen.BestellungsNr = Bestellungen_Oktober.BestellungsNr
```

6. Nun möchte Ihr Chef eine Übersicht über alle Kunden mit den Artikeln, die diese bestellt haben. Auch Kunden ohne Bestellungen sollen mit Vor- und Nachname aufgeführt werden; diese Spalten sollen vorne stehen

```
SELECT Nachname, Vorname, Artikel, Anzahl, Preis, Datum
FROM
  Kunden LEFT JOIN (
    Bestellungen_Oktober INNER JOIN Positionen
      ON Positionen.BestellungsNr = Bestellungen_Oktober.BestellungsNr
  )
  ON
    Kunden.KndNr = Bestellungen_Oktober.KndNr
```

#### 6.12.2.4 Zusammenfassung

- Mit OUTER JOIN werden auch Datensätze abgefragt und angezeigt, bei denen es in einer der Tabellen keinen zugeordneten Datensatz gibt.
- Mit einem LEFT JOIN erhält man alle Datensätze der linken Tabelle, ergänzt durch passende Angaben aus der rechten Tabelle.
- Mit einem RIGHT JOIN erhält man alle Datensätze der rechten Tabelle, ergänzt durch passende Angaben aus der linken Tabelle.
- Mit einem FULL JOIN erhält man alle Datensätze beider Tabellen, wenn möglich ergänzt durch passende Angaben aus der jeweils anderen Tabelle.
- Bei der Verknüpfung mehrerer Tabellen ist genau auf den JOIN-Typ und ggf. auf Klammerung zu achten.

### 6.12.2.5 Übungen OUTER JOIN

1. Welche der folgenden Aussagen sind wahr, welche sind falsch?
  1. Um alle Mitarbeiter mit Dienstwagen aufzulisten, benötigt man einen LEFT OUTER JOIN.
  2. LEFT JOIN ist nur eine Kurzschreibweise für LEFT OUTER JOIN und hat keine zusätzliche inhaltliche Bedeutung.
  3. Ein LEFT JOIN von zwei Tabellen enthält alle Zeilen, die nach Auswahlbedingung in der linken Tabelle enthalten sind.
  4. Ein RIGHT JOIN von zwei Tabellen enthält nur noch diejenigen Zeilen, die nach der Verknüpfungsbedingung in der linken Tabelle enthalten sind.
  5. Wenn wir bei einer LEFT JOIN-Abfrage mit zwei Tabellen die beiden Tabellen vertauschen und stattdessen einen RIGHT JOIN verwenden, erhalten wir dieselben Zeilen in der Ergebnismenge.
  6. Wir erhalten dabei nicht nur dieselben Zeilen, sondern auch dieselbe Reihenfolge.
2. Was ist am folgenden SELECT-Befehl falsch und warum?

Die Aufgabe dazu lautet:

Gesucht werden Kombinationen von Fahrzeug-Kennzeichen und Fahrzeugtypen, wobei alle Typen aufgeführt werden sollen; es werden nur die ersten 20 Fahrzeuge nach ID benötigt.

```
select Kennzeichen, Bezeichnung from Fahrzeug fz
left join Fahrzeugtyp ft on fz.Fahrzeugtyp_ID = ft.ID
where fz.ID <= 20 ;
```

3. Gesucht werden alle registrierten Versicherungsgesellschaften und (soweit vorhanden) deren Kunden mit Name, Vorname.
4. Gesucht werden die Dienstwagen, deren Fahrzeugtypen sowie die Hersteller. Die Liste der Typen soll vollständig sein.
5. Gesucht werden Kombinationen von Mitarbeitern und ihren Dienstwagen (einschl. Typ). Es geht um die Abteilungen 1 bis 5; auch nicht-persönliche Dienstwagen sollen aufgeführt werden.
6. Gesucht werden alle registrierten Versicherungsgesellschaften sowie alle Kunden mit Name, Vorname, soweit der Nachname mit 'S' beginnt.
7. Vertauschen Sie in der Lösung von Übung 5 die beiden Tabellen Mitarbeiter und Dienstwagen und erläutern Sie:
  1. Warum werden jetzt mehr Mitarbeiter angezeigt, und zwar auch solche ohne Dienstwagen?
  2. Warum fehlt jetzt der "nicht-persönliche" Dienstwagen?
8. Gesucht werden Angaben zu den Mitarbeitern und den Dienstwagen. Beim Mitarbeiter sollen Name und Vorname angegeben werden, bei den Dienstwagen Bezeichnung und Name des Herstellers. Die Liste aller Fahrzeugtypen soll vollständig sein.
9. Ergänzen Sie die Lösung zur obenstehenden Übung insofern, dass nur Mitarbeiter der Abteilungen 1 bis 5 angezeigt werden; die Zeilen ohne Mitarbeiter sollen unverändert ausgegeben werden.

### 6.12.2.6 Lösung OUTER JOIN

1. Die Aussagen 2, 3, 5 sind richtig, die Aussagen 1, 4, 6 sind falsch.
2. Richtig ist folgender Befehl:

```
select Kennzeichen, Bezeichnung
from Fahrzeug fz
right join Fahrzeugtyp ft on fz.Fahrzeugtyp_ID = ft.ID
where fz.ID <= 20 or fz.ID is null;
```

Weil alle Typen aufgeführt werden sollen, wird ein RIGHT JOIN benötigt. Damit auch der Vermerk „es gibt zu einem Typ keine Fahrzeuge“ erscheint, muss die WHERE-Klausel um die IS NULL-Prüfung erweitert werden.

3. Lösung

```
select Bezeichnung, Name, Vorname
from Versicherungsgesellschaft vg
left join Versicherungsnehmer vn on vg.ID = vn.Versicherungsgesellschaft_ID
order by Bezeichnung, Name, Vorname;
```

4. Lösung

```
select Kennzeichen, Bezeichnung, Name
from Dienstwagen dw
right join Fahrzeugtyp ft on ft.ID = dw.Fahrzeugtyp_ID
inner join Fahrzeughersteller fh on fh.ID = ft.Hersteller_ID
order by Name, Bezeichnung, Kennzeichen;
```

5. Lösung

```
SELECT mi.Name, mi.Vorname,
dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi
RIGHT JOIN Dienstwagen dw ON dw.Mitarbeiter_ID = mi.ID
where mi.Abteilung_ID <= 5 or mi.ID is null;
```

Die IS NULL-Prüfung wird wegen der "nicht-persönlichen" Dienstwagen benötigt.

## 6. Lösung

```
SELECT Bezeichnung, Name, Vorname
FROM Versicherungsgesellschaft vg
full JOIN Versicherungsnehmer vn ON vg.id = vn.Versi-
cherungsgesellschaft_id
where vn.Name like 'S%' or vn.id is null
ORDER BY Bezeichnung, Name, Vorname;
```

## 7. Lösung

1. Bei einem RIGHT JOIN werden alle Einträge der rechten Tabelle angezeigt. „Rechts“ stehen jetzt die Mitarbeiter, also werden alle Mitarbeiter der betreffenden Abteilungen angezeigt.
2. Bei einem RIGHT JOIN werden die Einträge der linken Tabelle nur dann angezeigt, wenn sie zu einem Eintrag der rechten Tabelle gehören. Der „nicht-persönliche“ Dienstwagen aus der linken Tabelle gehört aber zu keinem der Mitarbeiter.

## 8. Lösung

```
SELECT mi.Name, mi.Vorname,
dw.Kennzeichen, ft.Bezeichnung, fh.Name as HST
FROM Dienstwagen dw
left join Mitarbeiter mi on mi.id = dw.Mitarbeiter_id
right JOIN Fahrzeugtyp ft on ft.Id = dw.Fahrzeugtyp_id
inner join Fahrzeughersteller fh on fh.Id = ft.Her-
steller_id;
```

Hinweise: Die Reihenfolge der JOINS ist nicht eindeutig; der LEFT JOIN kann auch später kommen. Wichtig ist, dass die Verbindung Dienstwagen — Fahrzeugtyp ein RIGHT JOIN ist (oder bei Vertauschung der Tabellen ein LEFT JOIN).

## 9. Lösung

```
SELECT mi.Name, mi.Vorname,
dw.Kennzeichen, ft.Bezeichnung, fh.Name as HST
FROM Dienstwagen dw
left join Mitarbeiter mi on mi.id = dw.Mitarbeiter_id
right JOIN Fahrzeugtyp ft on ft.Id = dw.Fahrzeugtyp_id
inner join Fahrzeughersteller fh on fh.Id = ft.Her-
steller_id
where (mi.Abteilung_id <= 5) or (mi.id is null);
```

### 6.12.3 Unterabfragen

Immer wieder werden zur Durchführung einer Abfrage oder eines anderen Befehls Informationen benötigt, die zuerst durch eine eigene Abfrage geholt werden müssen. Solche "Unterabfragen" werden in diesem Kapitel behandelt.

- Wenn eine Abfrage als Ergebnis einen einzelnen Wert liefert, kann sie anstelle eines Wertes benutzt werden.
- Wenn eine Abfrage als Ergebnis eine Liste von Werten liefert, kann sie anstelle einer solchen Liste benutzt werden.
- Wenn eine Abfrage eine Ergebnismenge, also etwas in Form einer Tabelle liefert, kann sie anstelle einer Tabelle benutzt werden.

Bitte beachten Sie, dass die Unterabfrage schon aus Übersichtlichkeitsgründen immer in Klammern gesetzt werden sollte, auch wenn ein DBMS das nicht verlangen immer sollte

Allgemeiner Hinweis: Unterabfragen arbeiten in vielen Fällen langsamer als andere Verfahren. Soweit es irgend möglich ist, versuchen Sie, eine der JOIN-Varianten vorzuziehen.

#### Beispiel 1

Nenne alle Mitarbeiter der Abteilung „Schadensabwicklung“.

- Lösung Teil 1: Hole die ID dieser Abteilung anhand des Namens.
- Lösung Teil 2: Hole die Mitarbeiter der Abteilung unter Benutzung der gefundenen ID.

```
select Personalnummer, Name, Vorname
from Mitarbeiter
where Abteilung_ID =
( select ID from Abteilung
where Kuerzel = 'ScAb' );

PERSONALNUMMER NAME VORNAME
80001 Schindler Christina
80002 Aliman Zafer
80003 Langer Norbert
80004 Kolic Ivana
```

#### Beispiel 2

Nenne alle anderen Mitarbeiter der Abteilung, deren Leiterin Christina Schatzing ist.

- Lösung Teil 1: Hole die Abteilung\_ID, die bei Christina Schatzing registriert ist.
- Lösung Teil 2: Hole die Mitarbeiter dieser Abteilung unter Benutzung der gefundenen Abteilung\_ID.

```
select Personalnummer, Name, Vorname
from Mitarbeiter
where ( Abteilung_ID = (
    SELECT Abteilung_ID
    from Mitarbeiter
    where ( Name = 'Schindler' )
    and ( Vorname = 'Christina' )
    and ( Ist_Leiter = 'J' )
)
) and ( Ist_Leiter = 'N' );
```

### 6.12.3.1 Ergebnisse von Spaltenfunktionen

Häufig werden Ergebnisse von Aggregatfunktionen als Teil der WHERE-Klausel benötigt.

#### Beispiel 3

Hole die Schadensfälle mit unterdurchschnittlicher Schadenshöhe.

- Lösung Teil 1: Berechne die durchschnittliche Schadenshöhe aller Schadensfälle.
- Lösung Teil 2: Übernimm das Ergebnis als Vergleichswert in die eigentliche Abfrage.

```
SELECT ID, Datum, Ort, Schadenshoehe
from Schadensfall
where Schadenshoehe < (
    select AVG(Schadenshoehe) from Schadensfall
)
```

#### Beispiel 4

Bestimme alle Schadensfälle, die von der durchschnittlichen Schadenshöhe eines Jahres maximal 300 € abweichen.

- Lösung Teil 1: Bestimme den Durchschnitt aller Schadensfälle innerhalb eines Jahres.
- Lösung Teil 2: Hole alle Schadensfälle, deren Schadenshöhe im betreffenden Jahrinnerhalb des Bereichs „Durchschnitt plus/minus 300“ liegen.

```
select sf.ID, sf.Datum, sf.Schadenshoehe, EXTRACT(YEAR from
sf.Datum) AS Jahr
from Schadensfall sf
where ABS(Schadenshoehe - (
    select AVG(sf2.Schadenshoehe)
    from Schadensfall sf2
    where YEAR(sf2.Datum) = YEAR(sf.Datum)
)
) <= 300;
```

Zuerst muss für jeden einzelnen Schadensfall aus sf das Jahr bestimmt werden. In der Unterabfrage, die in der inneren Klammer steht, wird für alle Schadensfälle des betreffenden Jahres die durchschnittliche Schadenshöhe bestimmt. Dieser Wert wird mit der aktuellen Schadenshöhe verglichen; dazu wird die ABS-Funktion benutzt, also der absolute Betrag der Differenz der beiden Werte.

Dies ist ein Paradebeispiel dafür, wie Unterabfragen nicht benutzt werden sollen. Für jeden einzelnen Datensatz muss in der WHERE-Bedingung eine neue Unterabfrage gestartet werden – mit eigener WHERE-Klausel und Durchschnittsberechnung. Viel besser wäre eine der JOIN-Varianten.

Weitere mögliche Lösungen (Lutz, 11FI1, 2013/14

```
select beschreibung, schadenshoehe
from schadensfall where
schadenshoehe <= (
select avg(schadenshoehe)
from schadensfall) + 300
and schadenshoehe >= (select avg(schadenshoehe)
from schadensfall) - 300

select beschreibung, schadenshoehe
from schadensfall where
schadenshoehe between (
select avg(schadenshoehe)
from schadensfall) - 300
and (select avg(schadenshoehe)
from schadensfall) + 300

select @average:=avg(schadenshoehe) from schadensfall;
select id from schadensfall where abs(schadenshoehe -
@average) <= 300;
```

### 6.12.3.2 Ergebnis als Liste mehrerer Werte

Das Ergebnis einer Abfrage kann als Filter für die eigentliche Abfrage benutzt werden.

#### Beispiel 5

Bestimme alle Fahrzeuge eines bestimmten Herstellers.

- Lösung Teil 1: Hole die ID des gewünschten Herstellers.
- Lösung Teil 2: Hole alle IDs der Tabelle Fahrzeugtyp zu dieser Hersteller-ID.
- Lösung Teil 3: Hole alle Fahrzeuge, die zu dieser Liste von Fahrzeugtypen-IDs passen.

```
select ID, Kennzeichen, Fahrzeugtyp_ID as TypID
from Fahrzeug
where Fahrzeugtyp_ID in (
    select ID
    from Fahrzeugtyp
    where Hersteller_ID = (
        select ID
        from Fahrzeughhersteller
        where Name = 'Volkswagen' ) );
```

ID	KENNZEICHEN	TYPID
22	BOR-PQ 567	3
23	BOR-RS 890	2

Teil 1 der Lösung ist die „innere“ Klammer; dies ist das gleiche Verfahren wie im Abschnitt „Ergebnisse einfacher Abfragen“. Teil 2 der Lösung ist die „äußere“ Klammer; Ergebnis ist eine Liste von IDs der Tabelle Fahrzeugtyp, die als Werte für den Vergleich der WHERE-IN-Klausel verwendet werden.

Wenn im Ergebnis der Fahrzeugtyp als Text angezeigt werden soll, muss die Abfrage erweitert werden, weil die Bezeichnung in der Tabelle Fahrzeugtyp zu finden ist. Dafür kann diese Tabelle ein zweites Mal benutzt werden; es ist auch ein Verfahren möglich, wie es weiter unten erläutert wird.

### Beispiel 6

Gib alle Informationen zu den Schadensfällen des Jahres 2008, die von der durchschnittlichen Schadenshöhe 2008 maximal 300 € abweichen.

- Lösung Teil 1: Bestimme den Durchschnitt aller Schadensfälle innerhalb von 2008.
- Lösung Teil 2: Hole alle IDs von Schadensfällen, deren Schadenshöhe innerhalb des Bereichs „Durchschnitt plus/minus 300“ liegen.
- Lösung Teil 3: Hole alle anderen Informationen zu diesen IDs.

```
select *
from Schadensfall
where ID in ( SELECT ID
from Schadensfall
where ( ABS(Schadenshoehe - (
        select AVG(sf2.Schadenshoehe)
        from Schadensfall sf2
        where YEAR(sf2.Datum) = 2008
        )
    ) <= 300 )
and ( YEAR(Datum) = 2008 )
);
```

Diese Situation wird dadurch einfacher, dass das Jahr 2008 fest vorgegeben ist. Die innerste Klammer bestimmt als Teil 1 der Lösung die durchschnittliche Schadenshöhe dieses Jahres. Die nächste Klammer vergleicht diesen Wert (absolut gesehen) mit der Schadenshöhe eines jeden einzelnen Schadensfalls im Jahr 2008; alle „passenden“ IDs werden in der äußersten Klammer als Teil 2 der Lösung in einer weiteren Unterabfrage zusammengestellt. Diese Liste liefert die Werte für die eigentliche Abfrage.

### 6.12.3.3 Ergebnis in Form einer Tabelle

Das Ergebnis einer Abfrage kann in der Hauptabfrage überall dort eingesetzt werden, wo eine Tabelle vorgesehen ist. Die Struktur dieser Situation sieht so aus:

```
SELECT <spaltenliste>
FROM <haupttabelle>,
      (SELECT <spaltenliste>
       FROM <zusatztabellen>
       <weitere Bestandteile der Unterabfrage>
      ) <name>
      <weitere Bestandteile der Hauptabfrage>
```

Eine solche Unterabfrage kann grundsätzlich alle SELECT-Bestandteile enthalten. Bitte beachten Sie dabei: - Nach der schließenden Klammer muss ein Name als Tabellen-Alias angegeben werden, der als Ergebnistabelle in der Hauptabfrage verwendet wird.

- Die Unterabfrage kann eine oder mehrere Tabellen umfassen – wie jede andere Abfrage auch.
- In der Spaltenliste sollte jeweils ein Name als Spalten-Alias vor allem dann vorgesehen werden, wenn mehrere Tabellen verknüpft werden; andernfalls erzeugt SQL selbstständig Namen wie ID, ID1 usw., die man nicht ohne Weiteres versteht und zuordnen kann.
- ORDER BY kann nicht sinnvoll genutzt werden, weil das Ergebnis der Unterabfrage als Tabelle behandelt wird und mit der Haupttabelle oder einer anderen Tabelle verknüpft wird, wodurch eine Sortierung sowieso verlorenginge.

Wie gesagt: Eine solche Unterabfrage kann überall stehen, wo eine Tabelle vorgesehen ist. In der vorstehenden Syntax steht sie nur beispielhaft innerhalb der FROM-Klausel.

#### Beispiel 7

Bestimme alle Schadensfälle, die von der durchschnittlichen Schadenshöhe eines Jahres maximal 300 € abweichen.

- Lösung Teil 1: Stelle alle Jahre zusammen und bestimme den Durchschnitt aller Schadensfälle innerhalb eines Jahres.
- Lösung Teil 2: Hole alle Schadensfälle, deren Schadenshöhe im jeweiligen Jahr innerhalb des Bereichs „Durchschnitt plus/minus 300“ liegen.

```

SELECT sf.ID, sf.Datum, sf.Schadenshoehe, temp.Jahr,
temp.Durchschnitt
FROM Schadensfall sf,
( SELECT AVG(sf2.Schadenshoehe) AS Durchschnitt,
EXTRACT(YEAR FROM sf2.Datum) as Jahr
FROM Schadensfall sf2
group by EXTRACT(YEAR FROM sf2.Datum)
) temp
WHERE temp.Jahr = EXTRACT(YEAR FROM sf.Datum)
and ABS(Schadenshoehe - temp.Durchschnitt) <= 300;

```

Zuerst stellen wir durch eine Gruppierung alle Jahreszahlen und die durchschnittlichen Schadenshöhen zusammen (Teil 1 der Lösung). Für Teil 2 der Lösung muss für jeden Schadensfall nur noch Jahr und Schadenshöhe mit dem betreffenden Eintrag in der Ergebnistabelle temp verglichen werden.

Das ist der wesentliche Unterschied und entscheidende Vorteil zu anderen Lösungen: Die Durchschnittswerte werden einmalig zusammengestellt und nur noch abgerufen; sie müssen nicht bei jedem Datensatz neu (und ständig wiederholt) berechnet werden.

### Beispiel 8

Bestimme alle Fahrzeuge eines bestimmten Herstellers mit Angabe des Typs.

- Lösung Teil 1: Hole die ID des gewünschten Herstellers.
- Lösung Teil 2: Hole alle IDs und Bezeichnungen der Tabelle Fahrzeugtyp, die zu dieser Hersteller-ID gehören.
- Lösung Teil 3: Hole alle Fahrzeuge, die zu dieser Liste von Fahrzeugtyp-IDs gehören.

```

SELECT Fahrzeug.ID, Kennzeichen, Typen.ID As TYP, Typen.Bezeichnung
FROM Fahrzeug,
(SELECT ID, Bezeichnung
 FROM Fahrzeugtyp
 WHERE Hersteller_ID =
 (SELECT ID
  FROM Fahrzeughersteller
  WHERE Name = 'Volkswagen' )
 ) Typen
WHERE Fahrzeugtyp_ID = Typen.ID;

ID KENNZEICHEN TYP BEZEICHNUNG
23 BOR-RS 890    2      Golf
22 BOR-PQ 567    3      Passat

```

Teil 1 der Lösung ist die „innere“ Klammer; dies entspricht dem obigen Verfahren. Teil 2 der Lösung ist die „äußere“ Klammer; Ergebnis ist eine Tabelle von IDs und Bezeichnungen, also ein Teil der Tabelle Fahrzeugtyp, deren Werte für den Vergleich der WHERE-Klausel und außerdem für die Ausgabe verwendet werden.

### 6.12.3.4 Übungen

1. Welche der folgenden Feststellungen sind richtig, welche sind falsch?
  1. Das Ergebnis einer Unterabfrage kann verwendet werden, wenn es ein einzelner Wert oder eine Liste in Form einer Tabelle ist. Andere Ergebnisse sind nicht möglich.
  2. Ein einzelner Wert als Ergebnis kann durch eine direkte Abfrage oder durch eine Spaltenfunktion erhalten werden.
  3. Unterabfragen sollten nicht verwendet werden, wenn die WHERE-Bedingung für jede Zeile der Hauptabfrage einen anderen Wert erhält und deshalb die Unterabfrage neu ausgeführt werden muss.
  4. Mehrere Unterabfragen können verschachtelt werden.
  5. Für die Arbeitsgeschwindigkeit ist es gleichgültig, ob mehrere Unterabfragen oder JOINs verwendet werden.
  6. Eine Unterabfrage mit einer Tabelle als Ergebnis kann GROUP BY nicht sinnvoll nutzen.
  7. Eine Unterabfrage mit einer Tabelle als Ergebnis kann ORDER BY nicht sinnvoll nutzen.
  8. Bei einer Unterabfrage mit einer Tabelle als Ergebnis ist ein Alias-Name für die Tabelle sinnvoll, aber nicht notwendig.
  9. Bei einer Unterabfrage mit einer Tabelle als Ergebnis sind Alias-Namen für die Spalten sinnvoll, aber nicht notwendig.
2. Welche Verträge (mit einigen Angaben) hat der Mitarbeiter „Braun, Christian“ abgeschlossen? Ignorieren Sie die Möglichkeit, dass es mehrere Mitarbeiter dieses Namens geben könnte.
3. Zeigen Sie alle Verträge, die zum Kunden „Heckel Obsthandel GmbH“ gehören. Ignorieren Sie die Möglichkeit, dass der Kunde mehrfach gespeichert sein könnte.
4. Ändern Sie die Lösung von Übung 3, sodass auch mehrere Kunden mit diesem Namen als Ergebnis denkbar sind.
5. Zeigen Sie alle Fahrzeuge, die im Jahr 2008 an einem Schadensfall beteiligt waren.
6. Zeigen Sie alle Fahrzeugtypen (mit ID, Bezeichnung und Name des Herstellers), die im Jahr 2008 an einem Schadensfall beteiligt waren.
7. Bestimmen Sie alle Fahrzeuge eines bestimmten Herstellers mit Angabe des Typs.
8. Zeigen Sie zu jedem Mitarbeiter der Abteilung „Vertrieb“ den ersten Vertrag (mit einigen Angaben) an, den er abgeschlossen hat. Der Mitarbeiter soll mit ID und Name/Vorname angezeigt werden.

9. Von der Deutschen Post AG wird eine Tabelle PLZ\_Aenderung mit folgenden Inhalten geliefert:

ID	PLZalt	Ortalt	PLZneu	Ortneu
1	45658	Recklinghausen	45659	Recklinghausen
2	45721	Hamm-Bossendorf	45721	Haltern OT Hamm
3	45772	Marl	45770	Marl
4	45701	Herten	45699	Herten

Ändern Sie die Tabelle Versicherungsnehmer so, dass bei allen Adressen, bei denen PLZ/Ort mit PLZalt/Ortalt übereinstimmen, diese Angaben durch PLZneu/Ortneu geändert werden.

Hinweise: Beschränken Sie sich auf die Änderung mit der ID=3. (Die vollständige Lösung ist erst mit SQL-Programmierung möglich.) Bei dieser Änderungsdatei handelt es sich nur um fiktive Daten, keine echten Änderungen.

### 6.12.3.5 Lösungen

- Lösung zu Übung 1

Richtig sind 2, 3, 4, 7, 9; falsch sind 1, 5, 6, 8.

- Lösung zu Übung 2

```
select ID, Vertragsnummer, Abschlussdatum, Art
from Versicherungsvertrag
where Mitarbeiter_ID
in ( select ID
      from Mitarbeiter
      where Name = 'Braun'
        and Vorname = 'Christian' );
```

- Lösung zu Übung 3

```
select ID, Vertragsnummer, Abschlussdatum, Art
from Versicherungsvertrag
where Versicherungsnehmer_ID
= ( select ID from Versicherungsnehmer
      where Name = 'Heckel Obsthandel GmbH' );
```

- Lösung zu Übung 4

```
select ID, Vertragsnummer, Abschlussdatum, Art
from Versicherungsvertrag
where Versicherungsnehmer_ID
in ( select ID from Versicherungsnehmer
      where Name = 'Heckel Obsthandel GmbH' );
```

- Lösung zu Übung 5

```
select ID, Kennzeichen, Fahrzeugtyp_ID as TypID
from Fahrzeug fz
where ID in ( select Fahrzeug_ID
      from Zuordnung_sf_fz zu
      join Schadensfall sf on sf.ID = zu.Schadensfall_ID
      where EXTRACT(YEAR from sf.Datum) = 2008 );
```

- Lösung zu Übung 6

```
SELECT distinct ft.ID as TypID, ft.Bezeichnung as Typ,
fh.Name as Hersteller
FROM Fahrzeugtyp ft
inner join Fahrzeugherrsteller fh on fh.ID = ft.Hersteller_ID
right join Fahrzeug fz on ft.ID = fz.Fahrzeugtyp_ID
WHERE fz.ID IN ( SELECT Fahrzeug_ID
FROM Zuordnung_sf_fz zu
JOIN Schadensfall sf ON sf.ID = zu.Schadensfall_ID
WHERE EXTRACT(YEAR FROM sf.Datum) = 2008 );
```

Beachten Sie vor allem, dass die WHERE-Bedingung übernommen werden konnte, aber die Tabellen anders zu verknüpfen sind. Die Bedingung könnte in die ON-Klausel einbezogen werden; da sie aber die Auswahl beschränken soll, ist die WHERE-Klausel vorzuziehen.

- Lösung zu Übung 7

```
SELECT fz.ID, fz.Kennzeichen, Typen.ID AS TYP, Typen.Bezeichnung
FROM Fahrzeug fz
join ( SELECT ID, Bezeichnung
FROM Fahrzeugtyp
WHERE Hersteller_ID =
( SELECT ID FROM Fahrzeugherrsteller
WHERE Name = 'Volkswagen' )
) Typen on fz.Fahrzeugtyp_ID = Typen.ID;
```

- Lösung zu Übung 8

```
SELECT vv.ID as VV, vv.Vertragsnummer, vv.Abschlussdatum,
vv.Art,
mi.ID as MI, mi.Name, mi.Vorname
from Versicherungsvertrag vv
right join ( select MIN(vv2.ID) as ID, vv2.Mitarbeiter_ID
from Versicherungsvertrag vv2
group by vv2.Mitarbeiter_id ) Temp
on Temp.ID = vv.ID
right join Mitarbeiter mi on mi.ID = vv.Mitarbeiter_ID
where mi.Abteilung_ID = ( select ID from Abteilung
where Bezeichnung = 'Vertrieb' );
```

**Erläuterungen:**

Wir benötigen eine einfache Unterabfrage, um die Liste der Mitarbeiter für "Vertrieb" zu erhalten, und wir benötigen eine Unterabfrage, die uns zur Mitarbeiter-ID die kleinste Vertrags-ID liefert. Wegen der Aufgabenstellung "zu jedem Mitarbeiter" sowie "mit einigen Angaben" muss es sich bei beiden Verknüpfungen um einen RIGHT JOIN handeln.

- Lösung zu Übung 9

```
UPDATE Versicherungsnehmer
set PLZ, Ort
= ( Select PLZneu, Ortneu
from PLZ_Aenderg
where ID = 3 )
where PLZ = ( Select PLZalt
from PLZ_Aenderg
where ID = 3 )
and Ort = ( Select Ortalt
from PLZ_Aenderg
where ID = 3 );
```

Vielleicht funktioniert diese Variante bei Ihrem DBMS nicht; dann ist die folgende

**Version** nötig:

```
UPDATE Versicherungsnehmer
set PLZ = ( Select PLZneu
from PLZ_Aenderg
where ID = 3 ),
Ort = ( Select Ortneu
from PLZ_Aenderg
where ID = 3 )
where PLZ = ( Select PLZalt
from PLZ_Aenderg
where ID = 3 )
and Ort = ( Select Ortalt
from PLZ_Aenderg
where ID = 3 );
```

# 7 Transaktionen

- Datenbanken müssen für den gleichzeitigen Zugriff mehrerer Benutzer ausgelegt sein. Daraus können sich mannigfaltige Probleme ergeben. DBMS versuchen mit dem Prinzip der Transaktion diese Probleme in den Griff zu bekommen.
- Transaktionen führen einen konsistenten Datenbankzustand in einen anderen konsistenten Datenbankzustand über.
- Eine Transaktion ist als eine Folge von Anweisungen zu sehen, die entweder komplett übernommen (committ) oder abgebrochen (rollback) werden.

## 7.1 Probleme von Multi-User-Datenbanken

- Verlorengegangene Änderungen

2 User ändern zur gleichen Zeit einen einzelnen Datensatz (Lost Updates).

Zeit	Transaktion 1	Transaktion 2
2		
1	read(x)	
2		read(x)
3		$x = x + 100$
4		write(x)
5	$x = x + 1$	
6	write(x)	

- Schmutziges Lesen

Ein User sieht Änderungen, die noch nicht von ihm selbst oder von anderen Usern committed oder rollbacked wurden. (Dirty Reads)

Zeit	Transaktion 1	Transaktion 2
1	read(x)	
2	$x = x + 100$	
3	write(x)	
4		read(x)
5		$x = x - 100$
6	ABORT	
7		write(x)

- Nicht wiederholbare Lesezugriffe

User wählen wiederholt Zeilen aus, die andere User ändern oder löschen. Ob dies ein Problem darstellt hängt von den jeweiligen Umständen ab (Inventory vs. Reisebüro) ==> Unrepeatable Reads.

```
x = 40 y = 50 z = 30
Zeit          Transaktion 1           Transaktion 2
1             sum = 0
2             read( x )
3             read( y )
4             sum = sum + x
5             sum = sum + y
6                   read( z )
7             z = z + 10
8             write( z )
9             read( x )
10            x = x + 10
11            write( x )
12            read( z )
13            sum = sum + z
```

- Phantom-Zeilen

Ein User kann einige, aber nicht alle neuen Datensätze lesen, die ein anderer User eingegeben hat (Phantom rows).

Zeit	Transaktion 1	Transaktion 2
1	Select Counter from PassCounter	
2		Update Passengers set Flight = 4711 where Name = "Phantom"
3		Update PassCounter set Counter = Counter + 1
4	select * from Passengers	

- Inkonsistenzen (Interleaved Transactions)

Sie können auftreten, wenn Tabellen miteinander über Spaltenwerte verbunden sind (Fremdschlüssel), und diese Spalten nicht durch Trigger oder Constraints geschützt sind. Üblicherweise handelt es sich um ein timing-

Problem, wenn nicht vorhergesagt werden kann, in welcher Reihenfolge oder in welcher Häufigkeit auf die Tabellen zugegriffen wird.

Integritätsbedingung $x = y$		
Zeit	Transaktion 1	Transaktion 2
	Inkrementieren T1	Verdoppeln T2
1	read( x )	
2	$x = x + 1$	
3	write( x )	
4		read( x )
5		$x = 2 * x$
6		write( x )
7		read( y )
8		$y = 2 * y$
9		write( y )
10	read( y )	
11	$y = y + 1$	
12	write( y )	

## 7.2

### Eigenschaften einer Transaktion

Transaktionen werden durch die sog. ACID - Eigenschaften beschrieben:

- Atomarität

Eine Transaktion wird entweder ganz oder gar nicht ausgeführt

- Konsistenz (Serialisierbarkeit)

Transaktionen überführen die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand. Dies wird durch das Prinzip der Serialisierung erreicht.

- Isolation

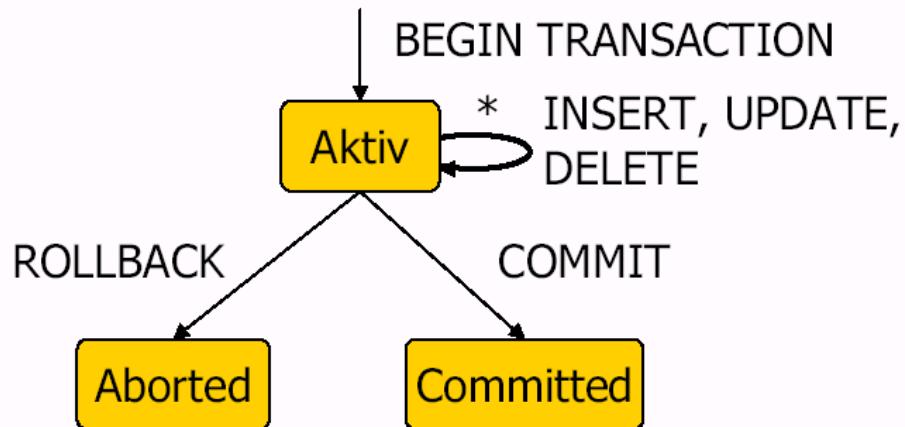
Nebenläufige (gleichzeitige) Transaktionen laufen jede für sich so ab, als ob sie alleine ablaufen würden. Verschiedene Isolationslevel sind möglich

- Dauerhaftigkeit

Die Wirkung einer abgeschlossenen (Dauerhaftigkeit) Transaktionen bleibt (auch nach einem Systemausfall) erhalten. Dies wird durch spezielle Recovery-Mechanismen erreicht

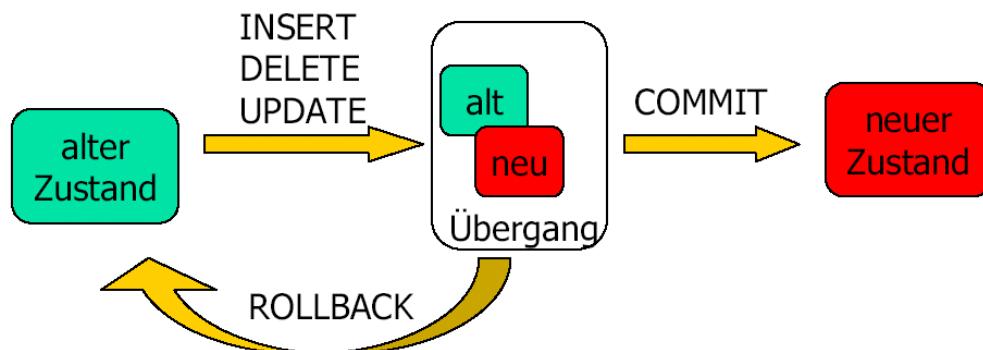
## COMMIT und ABORT

### ■ Zustände von Transaktionen



## Wirkung von COMMIT und ROLLBACK

### ■ Konsistenz-Garantie für beide Fälle



### 7.2.1 Serialisierung als Garantie der Transaktion

Serialisierung bedeutet, dass geprüft wird, ob gleichzeitig stattfindende Transaktionen sich gegenseitig behindern würden.

- Strenge Serialisierbarkeit - pessimistisch

Die einzelnen Transaktionen der User werden nur streng hintereinander ausgeführt

- Keine Überlappung der einzelnen Transaktionen.
- Geschwindigkeitsnachteile.
- Weiche Serialisierung - optimistisch

Erst beim Abschluss einer Transaktion wird geprüft, ob sie sich serialisieren lässt. Sie kann deshalb abgebrochen werden und muss von vorne beginnen. Sie durchlaufen normalerweise drei Phasen:

- Lesephase (Daten lesen und Berechnungen durchführen)
- Validierungsphase (überprüft Einhaltung der Konsistenzkriterien)
- Schreibphase (die in der Lesephase berechneten Änderungen werden eingetragen)

Optimistische Serialisierung lässt alle Reihenfolgen zu, die keinen Schaden anrichten können.

Beispiel: überlappend, aber unschädlich

Zeit	Transaktion 1	Transaktion 2
1	read(x) 1	
2		read(y) 2
3	x = x + 100 3	
4		y = y - 100 4
5	write(x) 5	
6		write(y) 6

Beispiel: überlappend und schädlich, nicht serialisierbar

Zeit	Transaktion 1	Transaktion 2
--		
1	read(x) 1	
2		read(x) 2
3		x = x + 100
4		write(x) 4
5	x = x + 1	
6	write(x)	

## 7.2.2 Zusammenfassung Serialisierung

### Pessimistische Verfahren:

- frühzeitige Erkennung der Konflikte
- mit Arbeitsverlust verbundene Abbrüche der Transaktionen nur bei Deadlock
- Zur Vermeidung von SchneeballEffekten notwendige Striktheit verhindert die mögliche Nebenläufigkeit

### Optimistische Verfahren:

- Nur LeseTransaktionen brauchen sich nicht um Synchronisation zu kümmern
- Keine DeadlockGefahr
- Beim Konflikttest können die tatsächlichen Ergebnisse der Operationen der Transaktionen berücksichtigt werden
- Änderungen müssen bis zum Ende der Transaktion in einem lokalen Speicher gehalten werden
- Ein großer Teil des Synchronisierungsaufwandes fällt gesammelt am Schluss an

## 7.2.3 Isolation-Level

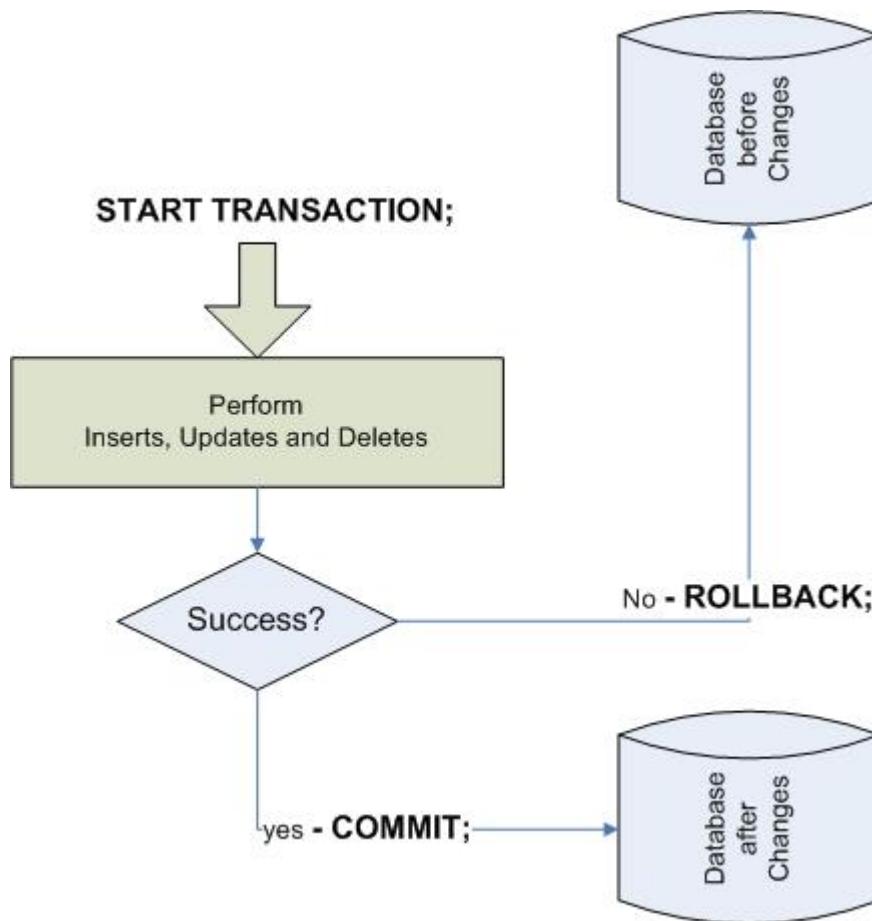
Im gleichen Zusammenhang wie der Begriff der Serialisierung ist der Begriff des Isolation-Level zu sehen. Je nach Grad des Isolation-Level können damit verschiedene Probleme des Multi-User-Zugriffes gelöst werden.

Isolationsgrad	Effekte
0	Dirty Read, Non-Repeatable-Read, Phantom
1	Non-Repeatable-Read, Phantom
2	Phantom
3	-

Standard ist der Isolation-Level 3. Geringere Level sind manchmal aus Performancegründen sinnvoll. Die Umsetzung der Serialisierung bzw. des Isolation-Level erfolgt normalerweise pessimistisch, und zwar mit Hilfe des **2- Phasen-Sperrprotokolls**.

Siehe dazu auch [folgenden link](#)

## 7.3 Beispiel zu Transaktionen in MySQL



Das folgende Beispiel geht von der Datenbank **test\_transaktion** aus. Diese besitzt eine Tabelle **Studio**.

```

-----
-- 
-- Tabellenstruktur für Tabelle `Studio`
-- 

create database test_transaktion;
use test_transaktion;

CREATE TABLE IF NOT EXISTS `Studio` (
    `studio_id` int(11) NOT NULL,
    `studio_name` varchar(30) DEFAULT NULL,
    PRIMARY KEY (`studio_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
  
```

- Wir beginnen mit demn Einfügen von 2 Datensätzen

```
START TRANSACTION;
INSERT INTO Studio VALUES (101, 'MGM Studios');
INSERT INTO Studio VALUES (102, 'Wannabe Studios');
COMMIT;
SELECT * FROM Studio;
```

- Ein Rollback würde zuvor eingefügte Datensätze wieder entfernen:

```
START TRANSACTION;
UPDATE Studio SET studio_name = 'Temporary Studios' WHERE
studio_id = 101;
UPDATE Studio SET studio_name = 'Studio with no buildings'
WHERE studio_id = 102;
SELECT * FROM Studio;
ROLLBACK;
SELECT * FROM Studio;
```

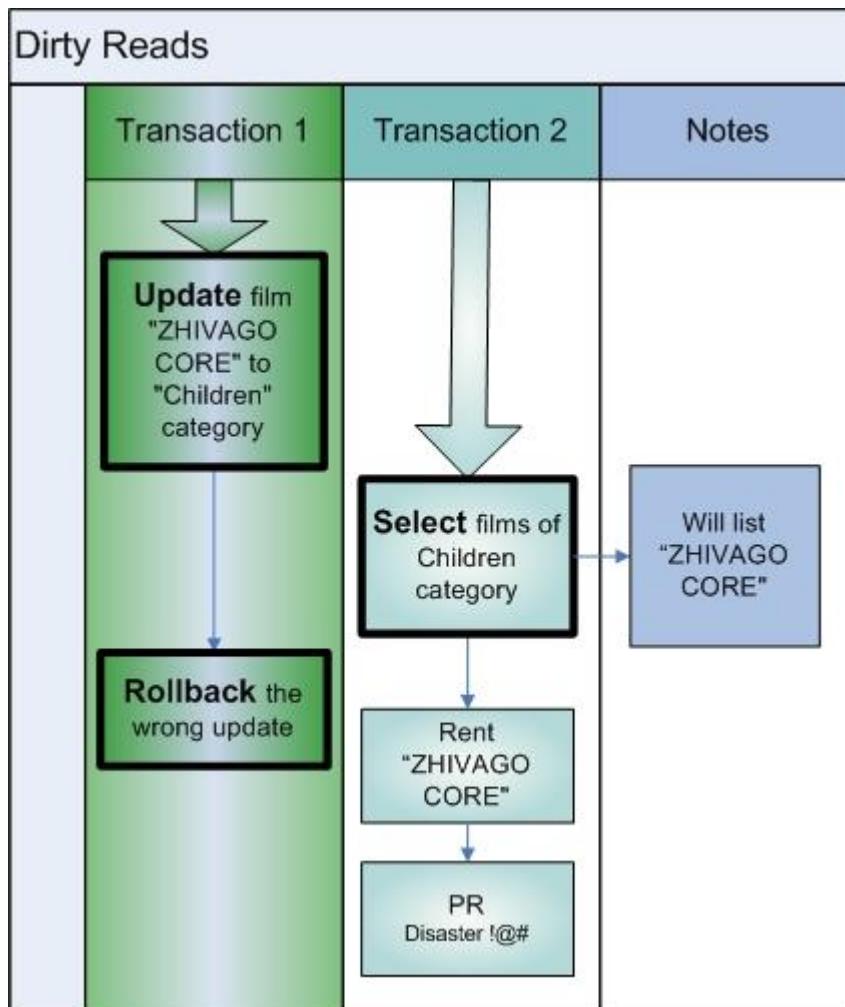
- SavePoints sind Haltepunkte innerhalb einer Transaktion, die dazu führen, dass auch bei einem Rollbac bestimme Anweisungen innerhalb einer Transaktion bestehen bleiben.

### 7.3.1 DirtyReads in MySQL

Das folgende Beispiel geht von der Datenbank **sakila** aus. Es werden zum Nachvollziehen zwei unterschiedliche Sessions auf die Datenbank benötigt.

```
use sakila;
-- Session 1
START TRANSACTION;
-- wir ändern die Kategory des Horror film "ZHIVAGO CORE" zu
"Children"
-- natürlich völlig unabsichtlich :-
UPDATE film_category SET category_id = 3 WHERE film_id = 998;

-- session 2
-- session 2 holt sich jetzt eine Liste aller Kinderfilme
-- wichtig ist dass der T-Level auf READ UNCOMMITTED steht
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM film f, film_category cat
WHERE f.film_id = cat.film_id
AND cat.category_id = 3;
-- session 1
-- wir bemerken den Fehler; machen ein Rollback in session 1
ROLLBACK;
SELECT * FROM film WHERE film_id = 998;
```



### 7.3.2 PhantomRows in MySQL

Sie brauchen 3 verschiedene Sessions zu einem DB-Server

```
--session 1
-- wir fügen 2 Datensätze hinzu
START TRANSACTION;
SELECT * FROM Studio;
INSERT INTO Studio VALUES (105, 'Noncomformant Studios');
INSERT INTO Studio VALUES (106, 'Studio Cartel');
SELECT * FROM Studio;

-- session2
-- sieht die neu eingefügten Datensätze
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM studio;

--session 1
-- Rücknahme der Inserts
ROLLBACK;
SELECT * FROM studio;

--session3
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM studio;
```

### 7.3.3 Zusatzbeispiel IsolationLevel

Die Isolationsebene wird mit dem Befehl

```
SET SESSION TRANSACTION ISOLATION LEVEL isolationsebene;
```

definiert. Zurzeit sind die folgenden 4 Ebenen spezifiziert:

**READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ und SERIALIZABLE.**

### 7.3.3.1 READ UNCOMMITTED

Es handelt sich eigentlich nicht um eine Isolationsebene, da die anderen Transaktionen oder Sessions den Zwischenstand lesen können. Da dabei auch inkonsistente Zustände auftreten können, wird dieser auch dirty read genannt.

SESSION 1	SESSION 2
<pre> 1 SET AUTOCOMMIT = 0; 2 3 4 SELECT artikel_id , menge_aktuell 5 FROM lagerbestand 6 WHERE artikel_id = 9015; 7 +-----+-----+ 8   artikel_id   menge_aktuell   9 +-----+-----+ 10   9015   16.000000  </pre>	<pre> 1 SET SESSION TRANSACTION ISOLATION LEVEL 2 READ UNCOMMITTED; 3 4 SELECT artikel_id , menge_aktuell 5 FROM lagerbestand 6 WHERE artikel_id = 9015; 7 +-----+-----+ 8   artikel_id   menge_aktuell   9 +-----+-----+ 10   9015   16.000000  </pre>
<pre> 11 +-----+-----+ 12 13 UPDATE lagerbestand 14 SET menge_aktuell = menge_aktuell-1 15 WHERE artikel_id = 9015; 16 17 SELECT artikel_id , menge_aktuell 18 FROM lagerbestand 19 WHERE artikel_id = 9015; 20 +-----+-----+ 21   artikel_id   menge_aktuell   22 +-----+-----+ 23   9015   15.000000  </pre>	<pre> 13 14 15 16 17 SELECT artikel_id , menge_aktuell 18 FROM lagerbestand 19 WHERE artikel_id = 9015; 20 +-----+-----+ 21   artikel_id   menge_aktuell   22 +-----+-----+ 23   9015   15.000000  </pre>
<pre> 24 +-----+-----+ 25 26 ROLLBACK; 27 28 SELECT artikel_id , menge_aktuell 29 FROM lagerbestand 30 WHERE artikel_id = 9015; 31 +-----+-----+ 32   artikel_id   menge_aktuell   33 +-----+-----+ 34   9015   16.000000  </pre>	<pre> 26 27 28 SELECT artikel_id , menge_aktuell 29 FROM lagerbestand 30 WHERE artikel_id = 9015; 31 +-----+-----+ 32   artikel_id   menge_aktuell   33 +-----+-----+ 34   9015   16.000000  </pre>
<pre> 35 +-----+-----+</pre>	<pre> 35 +-----+-----+ +-----+-----+</pre>

Um dieses Beispiel nachzubauen, sollten Sie zwei Instanzen des MySQL-Clients gleichzeitig vor sich haben. In Session 1 wird in der Zeile 1 der AUTOCOMMIT-Modus so eingestellt, dass er sich automatisch immer in einer Transaktion befindet. In der gleichen Zeile wird in

Session 2 dafür gesorgt, dass diese einen dirty read auf die Datenbank durchführen kann. Danach wird überprüft, dass auch in beiden Sessions die gleichen Daten vorliegen.

In Session 1 wird ab Zeile 13 der Lagerbestand der Spalten aktualisiert. Bitte beachten Sie, dass dies in der zweiten Session nicht passiert. In Zeile 17 wird jetzt in beiden Sessions wieder der Lagerbestand überprüft. Beide geben nun das gleiche Ergebnis aus, obwohl in Session 1 noch kein COMMIT erfolgt ist. Session 2 arbeitet also mit inkonsistenten Daten. An diesem Beispiel können Sie auch den Effekt von ROLLBACK beobachten. In Session 1 Zeile 26 wird der ROLLBACK angewiesen. Eine anschließende Überprüfung zeigt, dass in beiden Sessions wieder der ursprüngliche Zustand hergestellt wurde.



### 7.3.3.2 READ COMMITTED

Bei dieser Isolationsebene werden nur die Daten in einer Transaktion berücksichtigt, die durch andere mit einem COMMIT bestätigt wurden.

```
SESSION 1
1 SET AUTOCOMMIT = 0;
2
3
4 SELECT artikel_id , menge_aktuell
5 FROM lagerbestand
6 WHERE artikel_id = 9015;
7 +-----+-----+
8 | artikel_id | menge_aktuell |
9 +-----+-----+
10 | 9015 | 16.000000 |
11 +-----+-----+
12
13 UPDATE lagerbestand
14 SET menge_aktuell = menge_aktuell - 1
15 WHERE artikel_id = 9015;
16
17 SELECT artikel_id , menge_aktuell
18 FROM lagerbestand
19 WHERE artikel_id = 9015;
20 +-----+-----+
21 | artikel_id | menge_aktuell |
22 +-----+-----+
23 | 9015 | 15.000000 |
24 +-----+-----+
25
26 COMMIT;
26
28 SELECT artikel_id , menge_aktuell
29 FROM lagerbestand
30 WHERE artikel_id = 9015;
31 +-----+-----+
32 | artikel_id | menge_aktuell |
33 +-----+-----+
34 | 9015 | 15.000000 |
35 +-----+-----+
1 SET SESSION TRANSACTION ISOLATION LEVEL
2 READ COMMITTED;
3
4 SELECT artikel_id , menge_aktuell
5 FROM lagerbestand
6 WHERE artikel_id = 9015;
7 +-----+-----+
8 | artikel_id | menge_aktuell |
9 +-----+-----+
10 | 9015 | 16.000000 |
11 +-----+-----+
12
13
14
15
16
17 SELECT artikel_id , menge_aktuell
18 FROM lagerbestand
19 WHERE artikel_id = 9015;
20 +-----+-----+
21 | artikel_id | menge_aktuell |
22 +-----+-----+
23 | 9015 | 16.000000 |
24 +-----+-----+
25
26
27
28 SELECT artikel_id , menge_aktuell
29 FROM lagerbestand
30 WHERE artikel_id = 9015;
31 +-----+-----+
32 | artikel_id | menge_aktuell |
33 +-----+-----+
34 | 9015 | 15.000000 |
35 +-----+-----+
```

## 7.4 Aufgabe zu Transaktionen

- Für das Sperren und Freigeben im Rahmen von Transaktionen sind die Bereiche **Lock-Manager, Scheduler und Recovery-Manager** zuständig. Informieren Sie sich im Internet über deren Aufgaben und Arbeitsweisen.
- Was sind sog. Deadlocks ?
- Was ist bzw. wie funktioniert das pessimistische Serialisieren bzw. das optimistische Serialisieren
- Wie funktioniert das 2-Phasen-Sperrprotokoll ?

## 7.5 Weiterführende Links

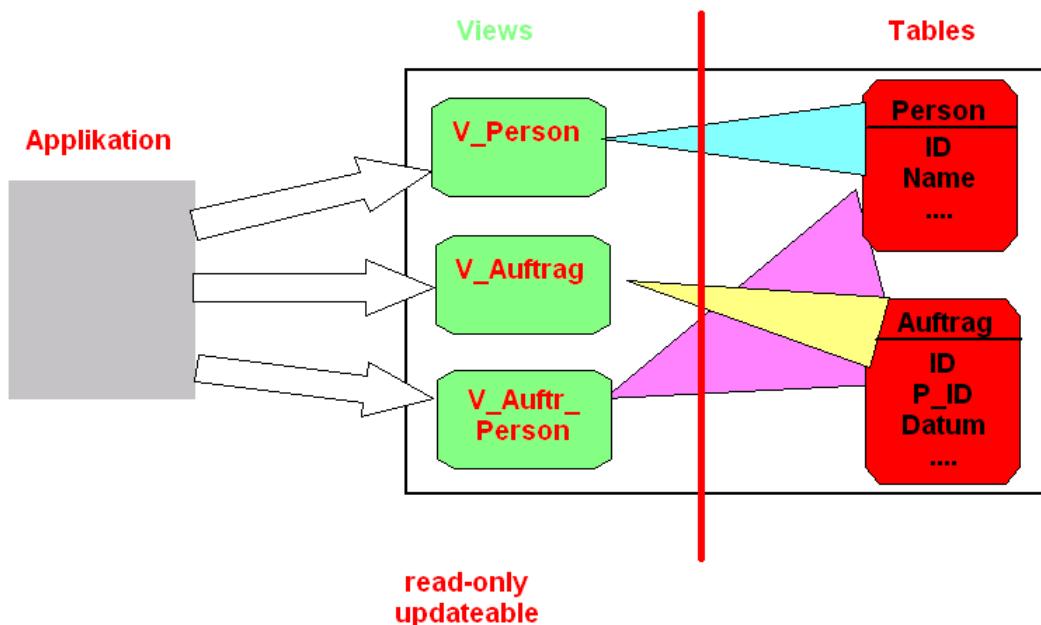
<http://rajeevrastogi.blogspot.de/2016/04/basic-of-transaction.html>

<http://rajeevrastogi.blogspot.de/2016/06/savepoint-andpreparetransaction.html>



## 8 Views

VIEWS sind Abfragen, die in der Datenbank als Objekt fest gespeichert sind. Sie können als virtuelle Tabellen verstanden werden, deren Inhalt und Struktur auf anderen Tabellen oder Views basieren, und können in (fast) jedem SELECT-Befehl anstelle einer „echten“ Tabelle verwendet werden.



Die Abfragen, auf denen Views basieren, können grundsätzlich alle Klauseln wie eine normale Abfrage enthalten. Somit ist es möglich, bestimmte Daten in einer View zu selektieren und zu gruppieren. Hierbei können die Daten aus mehreren Tabellen oder Views selektiert werden.

Je nach DBMS und Situation kann eine einzelne Klausel der View unwirksam sein oder zu unklaren Ergebnissen führen.

- Eine ORDER BY-Klausel der View wird ignoriert, wenn der SELECT-Befehl, der sie benutzt, selbst eine Sortierung verwendet.
- Bei einer Beschränkung durch LIMIT o. ä. weiß das DBMS oft nicht, nach welchen Regeln diese Beschränkung verwirklicht werden soll.
- WHERE-Bedingungen können nur fest eingebaut werden, aber nicht mit variablen Parametern.

```

CREATE VIEW name          --Name des Views
[(view_col           --Spaltennamen
[, view_col ...])]    --des Views
AS
select .....          --Select-Statement
[WITH CHECK OPTION];

```

z.B.

```

CREATE VIEW PHONE_LIST(
    EMP_NO,
    FIRST_NAME,
    LAST_NAME,
    PHONE_EXT,
    LOCATION,
    PHONE_NO)
AS

SELECT emp_no, first_name, last_name,
phone_ext, location, phone_no
FROM employee, department
WHERE employee.dept_no = department.dept_no;

```

Die View wird dann wie jede Tabelle benutzt, z. B. einfach:

```
SELECT * FROM <View-Name>
```

Oder auch als Teil einer komplexen Abfrage:

```

SELECT <irgendwas>
FROM <Tabelle>
JOIN <View-Name> ON /* usw. */

```

- Ein View ist grundsätzlich eine Art von SELECT - Abfrage, die Datensätze zurückgibt und sich in vielen Fällen wie eine normale Tabelle verhält.
- Normaler Tabellentyp im Sinne von SQL-89 und SQL-92
- Wird auch viewed table bzw. virtual table bezeichnet
- Es wird keine echte Tabelle angelegt, sondern nur die Metadaten werden gespeichert.
  - den eindeutigen Namen (identifier),
  - eine Liste der Spaltennamen
  - den eindeutigen Namen (identifier),
  - das SQL-Statement, mit dem die Daten aus den eigentlichen Basistabellen gewonnen werden

- Kann Inhalte aus mehreren Tabellen anzeigen
- In manchen Fällen sind Views updatebar, d.h. Änderungen am View wirken sich unmittelbar auf die zugrundeliegenden Tabellen aus.

### Beispiel

Erstelle eine View, die eine Liste aller Fahrzeugtypen deutscher Hersteller anzeigt.

```
CREATE VIEW Deutscher_Fahrzeugtyp
AS SELECT DISTINCT ft.Bezeichnung AS Fahrzeugtyp, fh.Name AS
Hersteller
FROM Fahrzeugtyp ft
join Fahrzeugherrsteller fh on ft.Hersteller_ID = fh.ID
WHERE fh.Land = 'Deutschland';
```

Die Abfrage basiert auf den beiden Tabellen Fahrzeugtyp und Fahrzeugherrsteller. Es werden nur die Spalten Bezeichnung und Name abgefragt; durch die WHERE-Klausel wird das Ergebnis auf Fahrzeuge deutscher Hersteller beschränkt. Für die Spalten werden Spalten-Aliase genutzt.

Diese View wird dann wie eine „normale“ Tabellen in Abfragen genutzt.

```
SELECT * FROM Deutscher_Fahrzeugtyp ORDER BY Hersteller;
```

Gerade in Verbindung mit Joins kann ein View wertvolle Zeit sparen, da er beispielsweise häufig benötigte Verknüpfungen bereits zur Verfügung stellen kann, ohne dass diese immer wieder neu definiert werden müssen.

### Beispiel

Bereite eine (fiktive) Tabelle Fahrzeugart vor mit allen relevanten Informationen aus den Tabellen Fahrzeugtyp und Fahrzeugherrsteller.

```
CREATE VIEW Fahrzeugart
( ID, Bezeichnung, Hersteller, Land )
AS
  SELECT ft.ID, ft.Bezeichnung, fh.Name, fh.Land
  FROM Fahrzeugtyp ft
  join Fahrzeugherrsteller fh on ft.Hersteller_ID = fh.ID;
```

Damit kann das nun folgende Beispiel vereinfacht werden, weil der letzte JOIN wegfällt.

Hole alle Dienstwagen (ggf. mit den zugehörigen Mitarbeitern) und nenne dazu alle Fahrzeugdaten.

```
SELECT
    mi.Personalnummer AS MitNr,
    mi.Name, mi.Vorname,
    dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS
    TypID,
    fa.Bezeichnung AS Typ, fa.Hersteller
FROM Dienstwagen dw
    LEFT JOIN Mitarbeiter mi ON mi.ID = dw.Mitarbeiter_ID
    INNER JOIN Fahrzeugart fa ON fa.ID = dw.Fahrzeugtyp_ID;
```

### Vorteil

- Vereinfachter Zugriff auf Daten, Wiederverwendbarkeit
- Situationsspezifischer Zugriff auf Daten
- Datenunabhängigkeit durch Abschirmen der User-Anwendungen von den Nebenwirkungen einer Änderung des Datenbankdesigns.
- So könnte man z.B. eine Tabelle in zwei Teiltabellen aufsplitten. Bisherige SQL-Statements wären damit unbrauchbar. Auf der anderen Seite kann aber nun ein View implementiert werden, der einen Join beider neuen Tabellen darstellt. Die User-Applikationen können nun ihre SQL-Statements weiterbenutzen. Sie müssen nur den Tabellennamen durch den Viewnamen ändern.
- Datensicherheit
- Views können den Zugriff auf sensitive oder irrelevante Teile einer Tabelle verweigern. So könnte z.B. ein Benutzer das Recht haben, sich Job-Informationen bzgl. der EMPLOYEE-Tabelle über einen View zu besorgen. Der Zugriff auf die Gehaltsinformation der Datensätze kann ihm aber dennoch verwehrt werden.

## 8.1 Aufgabe zu Views

1. Welche der folgenden Feststellungen sind richtig, welche sind falsch?
  1. Eine View ist wie eine „normale“ Abfrage, deren Bestandteile in der Datenbank fest gespeichert werden.
  2. Das Ergebnis dieser Abfrage wird gleichzeitig gespeichert und steht damit beim nächsten Aufruf der View sofort zur Verfügung.
  3. Eine ORDER BY-Klausel kann in einer View immer benutzt werden
  4. Eine ORDER BY-Klausel ist in einer View nicht erforderlich.
  5. Wenn diese Klausel in einer View benutzt wird, hat diese Sortierung Vorrang vor einer ORDER BY-Klausel in dem SELECT-Befehl, der die View benutzt.
  6. Wenn ein SELECT-Befehl komplexe JOINS oder andere Klauseln benutzt und häufiger benutzt wird, ist es sinnvoll, ihn in einer View zu kapseln.
  7. Wenn ein Anwender nicht alle Daten sehen darf, ist es notwendig, die Zugriffsrechte auf die Spalten zu beschränken; diese Beschränkung kann nicht über eine View gesteuert werden.
  8. Eine View kann in einem SELECT-Befehl in der FROM-Klausel anstatt einer Tabelle aufgerufen werden.
  9. Eine View kann nicht in einem JOIN benutzt werden.
2. Bei der Suche nach Dienstwagen sollen mit der View Dienstwagen\_Anzeige immer auch angezeigt werden:
  - o Name und Vorname des Mitarbeiters
  - o ID und Bezeichnung seiner Abteilung
  - o der Fahrzeugtyp (nur als ID)

Stellen Sie sicher, dass auch nicht-persönliche Dienstwagen immer angezeigt werden, und kontrollieren Sie das Ergebnis durch eine Abfrage ähnlich diesem Muster:

```
SELECT * FROM Dienstwagen_Anzeige
WHERE ( Abt_ID BETWEEN 5 AND 8 ) or ( Mi_Name is null );
```

3. Erweitern Sie die vorstehende View so, dass mit Hilfe der View Fahrzeugart auch Bezeichnung, Hersteller und Land angezeigt werden. Kontrollieren Sie das Ergebnis durch die o. g. Abfrage.
4. Erstellen Sie eine Sicht Vertrag\_Anzeige, die zu jedem Vertrag anzeigt:
  - o ID, Vertragsnummer, Abschlussdatum, Art (als Text)
  - o Name, Vorname des Mitarbeiters
  - o Name, Vorname des Versicherungsnehmers
  - o Kennzeichen des Fahrzeugs

5. Erweitern Sie die vorstehende View so, dass mit Hilfe der View Fahrzeugart auch Bezeichnung, Hersteller und Land angezeigt werden.
6. Erstellen Sie eine Sicht Schaden\_Anzeige, bei der zu jedem an einem Schadensfall beteiligten Fahrzeug angezeigt werden:
  - o ID, Datum, Gesamthöhe eines Schadensfalls
  - o Kennzeichen und Typ des beteiligten Fahrzeugs
  - o Anteiliger Schaden
  - o ID des Versicherungsvertrags
7. Erweitern Sie die vorstehende View so, dass mit Hilfe der View Fahrzeugart auch Bezeichnung, Hersteller und Land sowie Vertragsnummer und ID des Versicherungsnehmers angezeigt werden.
8. Erstellen Sie eine weitere View so, dass die vorstehende View für alle Schadensfälle des aktuellen Jahres benutzt wird.

## 8.2 Lösung zu Views

1. Richtig sind die Aussagen 1, 3, 4, 6, 8.

Falsch sind die Aussagen 2, 5, 7, 9.

2. View Dienstwagen\_Anzeige

```
create view Dienstwagen_Anzeige
    ( Kennzeichen, TypId,
      Mi_Name, Mi_Vorname,
      Ab_ID, Ab_Name )
as select dw.Kennzeichen, dw.Fahrzeugtyp_ID,
          mi.Name, mi.Vorname,
          mi.Abteilung_ID,
          ab.Bezeichnung
from Dienstwagen dw
left join Mitarbeiter mi
on mi.ID = dw.Mitarbeiter_ID
left join Abteilung ab
on ab.ID = mi.Abteilung_ID;
```

Erläuterung: LEFT JOIN in beiden Fällen wird benötigt, damit auch NULL - Werte, nämlich die nicht-persönlichen Dienstwagen angezeigt werden.

3. Erweitern Dienstwagen\_Anzeige

```
alter view Dienstwagen_Anzeige
    ( Kennzeichen, TypId,
      Typ, Fz_Hersteller, Fz_Land,
      Mi_Name, Mi_Vorname,
      Ab_ID, Ab_Name )
as select dw.Kennzeichen, dw.Fahrzeugtyp_ID,
          fa.Bezeichnung, fa.Hersteller, fa.Land,
          mi.Name, mi.Vorname,
          mi.Abteilung_ID,
          ab.Bezeichnung
from Dienstwagen dw
left join Mitarbeiter mi
on mi.ID = dw.Mitarbeiter_ID
left join Abteilung ab
on ab.ID = mi.Abteilung_ID
inner join Fahrzeugart fa
on fa.ID = dw.Fahrzeugtyp_ID;
```

#### 4. View Vertrag\_Anzeige

```
create view Vertrag_Anzeige
    ( ID, Vertragsnummer, Abschlussdatum, Art,
      Mi_Name, Mi_Vorname, Vn_Name, Vn_Vorname, Kennzeichen)
as select vv.ID, vv.Vertragsnummer, vv.Abschlussdatum,
CASE vv.Art
WHEN 'TK' THEN 'Teilkasko'
WHEN 'VK' THEN 'Vollkasko'
ELSE 'Haftpflicht'
END,
mi.Name, mi.Vorname,
vn.Name, vn.Vorname,
fz.Kennzeichen
from Versicherungsvertrag vv
join Mitarbeiter mi
on mi.ID = vv.Mitarbeiter_ID
join Versicherungsnehmer vn
on vn.ID = vv.Versicherungsnehmer_ID
join Fahrzeug fz
on fz.ID = vv.Fahrzeug_ID;
```

#### 5. Erweitern Vertrag\_Anzeige

```
alter view Vertrag_Anzeige
    ( ID, Vertragsnummer, Abschlussdatum, Art,
      Mi_Name, Mi_Vorname, Vn_Name, Vn_Vorname,
      Kennzeichen, Typ, Hersteller, Land )
as select vv.ID, vv.Vertragsnummer, vv.Abschlussdatum,
CASE vv.Art
WHEN 'TK' THEN 'Teilkasko'
WHEN 'VK' THEN 'Vollkasko'
ELSE 'Haftpflicht'
END,
mi.Name, mi.Vorname,
vn.Name, vn.Vorname,
fz.Kennzeichen, fa.Bezeichnung, fa.Hersteller, fa.Land
from Versicherungsvertrag vv
join Mitarbeiter mi
on mi.ID = vv.Mitarbeiter_ID
join Versicherungsnehmer vn
on vn.ID = vv.Versicherungsnehmer_ID
join Fahrzeug fz
on fz.ID = vv.Fahrzeug_ID
join Fahrzeugart fa
on fa.ID = fz.Fahrzeugtyp_ID;
```

## 6. View Schaden\_Anzeige

```
create view Schaden_Anzeige
  (ID, Datum, Gesamtschaden,
   Kennzeichen, Typ,
   Schadensanteil, VV_ID)
as select sf.ID, sf.Datum, sf.Schadenshoehe,
          fz.Kennzeichen, fz.Fahrzeugtyp_ID,
          zu.Schadenshoehe, vv.ID
from Zuordnung_SF_FZ zu
  join Schadensfall sf
    on sf.ID = zu.Schadensfall_ID
  join Fahrzeug fz
    on fz.ID = zu.Fahrzeug_ID
  join Versicherungsvertrag vv
    on fz.ID = vv.Fahrzeug_ID;
```

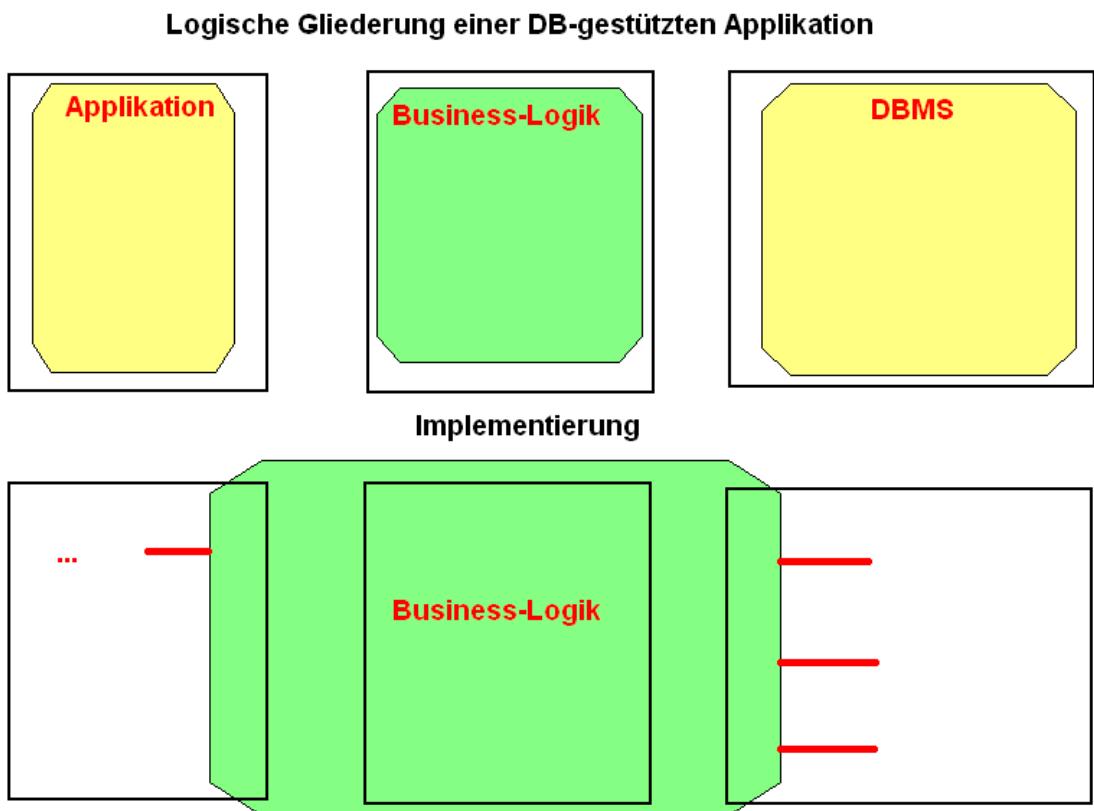
## 7. Erweiterung Schaden\_Anzeige

```
alter view Schaden_Anzeige
  (ID, Datum, Gesamtschaden,
   Kennzeichen, Typ, Hersteller, Land,
   Schadensanteil, VV_ID, Vertragsnummer, VN_ID )
as select sf.ID, sf.Datum, sf.Schadenshoehe,
          fz.Kennzeichen, fa.Bezeichnung, fa.Hersteller,
          fa.Land, zu.Schadenshoehe, vv.ID,
          vv.Vertragsnummer, vv.Versicherungsnehmer_ID
from Zuordnung_SF_FZ zu
  join Schadensfall sf
    on sf.ID = zu.Schadensfall_ID
  join Fahrzeug fz
    on fz.ID = zu.Fahrzeug_ID
  join Versicherungsvertrag vv
    on fz.ID = vv.Fahrzeug_ID
  join Fahrzeugart fa
    on fa.ID = fz.Fahrzeugtyp_ID;
```

## 8. View erweitern

```
create view Schaden_Anzeige_Jahr
as select *
  from Schaden_Anzeige
  where YEAR(Datum) =YEAR(NOW());
```

# 9 Trigger



Ein Trigger ist so etwas wie eine Routine zur Ereignisbehandlung: Immer dann, wenn sich in der Datenbank eine bestimmte Situation ereignet, wird eine spezielle Prozedur automatisch ausgeführt.

SQL-Triggers werden zur Überwachung von semantischen Integritätsbedingungen eingesetzt. Trigger bilden dabei eine einfache Form von ECA-Regeln. ECA steht für „Event, Control, Action“ und definiert in Zusammenhang mit Datenbanken eine Aktion, die nach der Kontrolle eines Ereignis ausgeführt werden soll.

Ein Trigger ist in folgenden Situationen nützlich:

- Werte in einer Zeile einer Tabelle sollen festgelegt werden.
- Werte sollen vor dem Speichern auf ihre Plausibilität geprüft werden.
- Veränderungen in der Datenbank sollen automatisch protokolliert werden.

## 9.1 Trigger erstellen

Die Syntax für die Definition eines Triggers sieht grundsätzlich so aus:

```
CREATE OR ALTER TRIGGER <routine-name> FOR <Tabellename>
[ ACTIVE | INACTIVE ]
{ BEFORE | AFTER }
{ INSERT | UPDATE | DELETE }
[ POSITION <zahl> ]
AS
BEGIN
[ <variablenliste> ]
<routine body>
END
```

Notwendig sind folgende Angaben:

- neben dem Befehlsnamen der Name des Triggers
- dazu die Tabelle, zu der er gehört
- mehrere Angaben, bei welchem Befehl und an welcher Stelle er wirksam sein soll;
- das Schlüsselwort AS als Zeichen für den Inhalt
- die Schlüsselwörter BEGIN und END als Begrenzer für den Inhalt

Hinzu kommen die folgenden Angaben:

- eine Liste von Variablen, die innerhalb der Routine verwendet werden; diese Liste steht je nach DBMS zwischen AS und BEGIN oder innerhalb des Rumpfes (also zwischen BEGIN und END)
- die Befehle, die innerhalb der Prozedur ausgeführt werden sollen
- Eingabe- und Ausgabe-Parameter gibt es nicht, weil es sich um automatische Arbeitsabläufe handelt.

Die folgenden Variablen stehen immer zur Verfügung.

- **OLD** ist der Datensatz vor einer Änderung.
- **NEW** ist der Datensatz nach einer Änderung.

Mit diesen Variablen zusammen mit den Spaltennamen können die Werte „vorher“ und „nachher“ geprüft und bearbeitet werden. Bei den Befehlen innerhalb des Triggers handelt es sich (eher selten) um „normale“ SQL-Befehle und überwiegend um Bestandteile der SQL- Programmiersprache.

## Beispiel

Skizzieren Sie den Inhalt eines Triggers Mitarbeiter\_On\_Delete für die Tabelle Mitarbeiter, der folgende Aufgaben ausführen soll:

Zu behandeln ist die Situation, dass ein Mitarbeiter aus der Firma ausscheidet, also zu löschen ist.

- Suche die ID des zugehörigen Abteilungsleiters.
- In allen Tabellen und Datensätzen, in denen der Mitarbeiter als Sachbearbeiter registriert ist, ist stattdessen der Abteilungsleiter als „zuständig“ einzutragen. Ignorieren Sie die Situation, dass der Abteilungsleiter selbst ausscheidet.

## Lösung

- Wir brauchen einen Trigger BEFORE DELETE zur Tabelle Mitarbeiter.
- Wir brauchen eine Variable für die ID des Abteilungsleiters.
- Durch einen SELECT auf die Abteilung\_ID des ausscheidenden Mitarbeiters bekommen wir die ID des Abteilungsleiters; die wird in der Variablen gespeichert.
- Für alle betroffenen Tabellen ist ein UPDATE zu machen, durch das die Mitarbeiter\_ID durch die ID des Abteilungsleiters ersetzt wird. Dies betrifft die Tabellen Schadensfall, Versicherungsvertrag.
- Die Tabelle Dienstwagen kann unberücksichtigt bleiben wegen des o. g. Beispiels.

```
-- Firebird-Version
create or alter trigger Mitarbeiter_On_Delete for Mitarbeiter
ACTIVE BEFORE DELETE POSITION 10
as
    DECLARE VARIABLE ltr_id INTEGER;
BEGIN
    ltr_id = null;
    -- hole die ID des Abteilungsleiters in die Variable
    select ID
    from Mitarbeiter
    where Abteilung_id = old.Abteilung_ID
    and Ist_Leiter = 'J'
    into :ltr_id;

    -- ändere die Mitarbeiter_ID für die Schadensfälle
    update Schadensfall
    set Mitarbeiter_ID = :ltr_id
    where Mitarbeiter_ID = old.ID;

    -- ändere die Mitarbeiter_ID für die Verträge
    update Versicherungsvertrag
    set Mitarbeiter_ID = :ltr_id
    where Mitarbeiter_ID = old.ID;
END
```

```

-- mysql-Version
-- what you write
delimiter //


create trigger Mitarbeiter_On_Delete2 after DELETE
  on Mitarbeiter
  for each row
BEGIN
    DECLARE ltr_id INTEGER;

    select ID
      from Mitarbeiter
     where Abteilung_id = old.Abteilung_ID
       and Ist_Leiter = 'J'
      into ltr_id;

    update Schadensfall
      set Mitarbeiter_ID = ltr_id
     where Mitarbeiter_ID = old.ID;

    update Versicherungsvertrag
      set Mitarbeiter_ID = ltr_id
     where Mitarbeiter_ID = old.ID;
END; //

delimiter;

-- is not what you get

CREATE DEFINER = 'root'@'localhost' TRIGGER `Mitarbeiter_On_Delete` BEFORE DELETE ON `mitarbeiter`
FOR EACH ROW
BEGIN
    DECLARE ltr_id INTEGER;

    select ID
      from Mitarbeiter
     where Abteilung_id = old.Abteilung_ID
       and Ist_Leiter = 'J'
      into ltr_id;

    update Schadensfall
      set Mitarbeiter_ID = ltr_id
     where Mitarbeiter_ID = old.ID;

    update Versicherungsvertrag
      set Mitarbeiter_ID = ltr_id
     where Mitarbeiter_ID = old.ID;
END;

```

**Aufgabe:**

Ein Trigger soll folgende Aufgabe erledigen:

- Wenn ein neuer Fahrzeugtyp aufgenommen wird, ist der Hersteller zu überprüfen.
- Ist der angegebene Hersteller (d. h. die Hersteller\_ID) in der Tabelle Fahrzeugherrsteller gespeichert? Wenn ja, dann ist nichts weiter zu erledigen.
- Wenn nein, dann ist ein Fahrzeugherrsteller mit dem Namen „unbekannt“ zu suchen.
- Wenn dieser vorhanden ist, ist dessen ID als Hersteller\_ID zu übernehmen.
- Andernfalls ist er mit der angegebenen Hersteller\_ID neu als „unbekannt“ zu registrieren.

**Lösung**

```

1. CREATE Fahrzeugtyp_Check_Hersteller as trigger
2. to Fahrzeugtyp
3. for INSERT
4. AS
5. DECLARE VARIABLE INTEGER fh_id
6. BEGIN
7. -- Initialisierung
8. fh_id = null;
9. -- prüfen, ob ID vorhanden ist
10. select ID
11. from Fahrzeugherrsteller
12. where ID = new.Hersteller_ID
13. -- wenn sie nicht gefunden wurde
14. if (fh_id is null) THEN
15. -- suche stattdessen den Hersteller 'unbekannt'
16. select ID
17. from Fahrzeugherrsteller
18. where Name = 'unbekannt'
19. -- wenn auch dazu die ID nicht gefunden wurde
20. if (fh_id is null) THEN
21. fh_id = new.Hersteller_ID;
22. insert into Fahrzeugherrsteller
23. values ( :fh_id, 'unbekannt', 'unbekannt' )
24. END
25. END
26. new.Hersteller_ID = :fh_id
27. END

```

## Beispiel

Zur praktischen Veranschaulichung habe ich ein MySQL-Skript mit drei Tabellen und einem Trigger geschrieben. In der Tabelle „users“ können Benutzer mit einem Vor- und Nachnamen eingetragen werden. Der im Skript definierte Trigger führt nach dem Eintragen in die „users“-Tabelle einen Stringvergleich aus und schreibt alle Benutzer, deren Nachname nicht mit einem Buchstaben größer „N“ anfängt in die „users\_a\_m“-Tabelle. Alle Benutzer, deren Nachname mit einem „N“ oder größer beginnt, werden in die Tabelle „users\_n\_z“ kopiert.

Mein Beispiel demonstriert neben dem definierten After-Trigger auch noch eine Möglichkeit zur Datenbank-Skalierung. Man könnte nämlich die Tabellen „users\_a\_m“ und „users\_n\_z“ auf verschiedene Datenbank-Server auslagern und beim Lesen von Datensätzen die entsprechende Datenbank anfragen (je nachdem mit welchem Anfangsbuchstaben der Nachname des gesuchten Benutzers anfängt). Dies hätte den Vorteil, dass die Datenbank-Anfragen verteilt werden und die Anfrage-Last nicht auf eine einzige Datenbank trifft.

```
-- Settings
DELIMITER $

-- Create database
DROP DATABASE IF EXISTS testdb_trigger$;
CREATE DATABASE testdb_trigger DEFAULT CHARACTER SET utf8
COLLATE utf8_general_ci$;
USE testdb_trigger$;

-- Create tables
CREATE TABLE IF NOT EXISTS users
(
    user_id      INT(10)      NOT NULL AUTO_INCREMENT
    ,surname     VARCHAR(255)  NOT NULL
    ,prename     VARCHAR(255)  NOT NULL
    ,PRIMARY KEY(user_id)
) ENGINE = InnoDB CHARACTER SET utf8 COLLATE utf8_general_ci$;

CREATE TABLE IF NOT EXISTS users_a_m
(
    id          INT(10)      NOT NULL AUTO_INCREMENT
    ,user_id    INT(10)      NOT NULL
    ,surname   VARCHAR(255)  NOT NULL
    ,prename   VARCHAR(255)  NOT NULL
    ,PRIMARY KEY(id)
    ,FOREIGN KEY(user_id) REFERENCES users(user_id) ON
DELETE CASCADE ON UPDATE CASCADE
) ENGINE = InnoDB CHARACTER SET utf8 COLLATE utf8_general_ci$;

CREATE TABLE IF NOT EXISTS users_n_z
```

```

(
    id          INT(10)      NOT NULL AUTO_INCREMENT
    ,user_id    INT(10)      NOT NULL
    ,surname    VARCHAR(255) NOT NULL
    ,prename    VARCHAR(255) NOT NULL
    ,PRIMARY KEY(id)
    ,FOREIGN KEY(user_id) REFERENCES users(user_id) ON
DELETE CASCADE ON UPDATE CASCADE
) ENGINE = InnoDB CHARACTER SET utf8 COLLATE utf8_general_ci$


-- Create Trigger
CREATE TRIGGER split_users AFTER INSERT
ON users FOR EACH ROW
BEGIN
    IF(NEW.surname BETWEEN 'A' AND 'N')
    THEN
        INSERT INTO users_a_m VALUES (NULL, NEW.user_id,
        NEW.surname, NEW.prename);
    ELSE
        INSERT INTO users_n_z VALUES (NULL, NEW.user_id,
        NEW.surname, NEW.prename);
    END IF;
END$


-- Insert 500 people
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Quander','Nadja')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Labuhn','Milan')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Ebisch','Alena')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Paffenholz','Ellen')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Haas','Vin')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Jaber','Chris')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Radicke','Yannik')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Abadre','Sean')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Nähr','Friedrich')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Waagner','Lilian')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Näger','Amelia')$
INSERT INTO users(user_id,surname,prename) VALUES (NULL,'Sach','Marit')$
```

## Funktionen



# 10 Stored Procedures

<http://www.postgresqltutorial.com/plpgsql-block-structure/>

## 10.1 Introduction

Summary: in this tutorial, you will learn about PostgreSQL stored procedures for developing functions in PostgreSQL.

PostgreSQL allows you to extend the database functionality with user-defined functions by using various procedural languages, which often referred to as stored procedures.

The store procedures define functions for creating triggers or custom aggregate functions. In addition, stored procedures also add many procedural features e.g., control structures and complex calculation. These allow you to develop custom functions much easier and more effective.

It is possible to call a procedural code block using the DO command without defining a function.

PostgreSQL divides the procedural languages into two main groups:

- Safe languages can be used by any users. SQL and PL/pgSQL are safe languages.
- Sand-boxed languages are only used by superusers because sand-boxed languages provide the capability to bypass security and allow access external sources. C is an example of a sand-boxed language or unsafe language.

By default, PostgreSQL supports 3 procedural languages: SQL, PL/pgSQL, and C. You can also load other procedural languages e. g. ,Perl, Python, and TCL into PostgreSQL using extensions.

### Advantages of using PostgreSQL stored procedures

The stored procedures bring many advantages as following:

- Reduce the number of round trips between application and database servers.

All SQL statements are wrapped inside a function stored in the PostgreSQL database server so the application only has to issue a function call to get the result back instead of sending multiple SQL statements and wait for the result between each call.

- Increase application performance  
because user-defined functions pre-compiled and stored in the PostgreSQL database server. Be able to reuse in many applications. Once you develop a function, you can reuse it in any applications.

### Disadvantages of using PostgreSQL stored procedures

Besides those advantages of using stored procedures, there are some caveats as follows:

- Slow in software development because it requires specialized skills that many developers do not possess.
- Make it difficult to manage versions and hard to debug.
- May not be portable to other database management systems e.g., MySQL or Microsoft SQL Server.

In the next tutorial, we will show you how to develop user-defined functions using PL/pgSQL, which comes with PostgreSQL by default.

In addition, PL/pgSQL is similar to PL/SQL (more on PL/SQL Tutorial), which is easy to learn.

In this tutorial, we have introduced you to PostgreSQL stored procedures and discussed about the pros and cons of using stored procedures in PostgreSQL.

Let's move to the next tutorial to develop the first user-defined function using PL/pgSQL procedural language.

## 10.2 Block Structure

Summary: in this tutorial, you will learn about the block structure of PL/pgSQL. You will write and execute the first PL/pgSQL block.

A PL/pgSQL function is organized into blocks. The following illustrates the syntax of a complete block in PL/pgSQL:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements;
...
END [ label ];
```

Let's examine the block structure in more detail:

Each block has two sections called declaration and body. The declaration section is optional while the body section is required. The block is ended with a semicolon (;) after the END keyword. A block may have optional labels at the beginning and at the end. The label at the beginning and at the end must be the same. The block label is used in case you want to use the block in EXIT statement or you want to qualify the names of variables declared in the block. The declaration section is where you declare all variables used within the body section. Each statement in the declaration section is terminated with a semicolon (;). The body section is where you put the logic of the block. It contains any valid statements. Each statement in the body section is also terminated with a semicolon (;).

### PL/pgSQL block structure example

The following example illustrates a very simple block.

```
1 DO $$  
2 <<first_block>>  
3 DECLARE  
4   counter integer := 0;  
5 BEGIN  
6   counter := counter + 1;  
7   RAISE NOTICE 'The current value of counter is %', counter;  
8 END first_block $$;
```

```
1 NOTICE: The current value of counter is 1
```

Notice that the DO statement does not belong to the block. It is used to execute an anonymous block. PostgreSQL introduced the DO statement since version 9.0.

In the declaration section, we declared a variable named counter and set its value to 0. Inside the body section, we increased the counter to 1 and output its value using RAISE NOTICE statement. The first\_block label is just for demonstration purpose. It does nothing in this example. PL/pgSQL Subblock

You can put a block inside the body of another block. This block nested inside another is called subblock. The block that contains the subblock is referred to as an outer block.

## 10.3 PL/pgSQL Block Structure



You often use subblocks for grouping statements so that a large block can be divided into smaller and more logical subblocks. The variables in the subblock can have the names as the ones in the outer block, even though it is not a good practice.

When you define a variable within subblock with the same name as the one in the outer block, the variable in the outer block is hidden in the subblock. In case you want to access a variable in the outer block, you use block label to qualify its name; see the following example:

```
DO $$ <<outer_block>> DECLARE
    counter integer := 0;
BEGIN
    counter := counter + 1;
    RAISE NOTICE 'The current value of counter is %', counter;
DECLARE
    counter integer := 0;
BEGIN
    counter := counter + 10;
    RAISE NOTICE 'The current value of counter in the subblock is %',
        counter;
    RAISE NOTICE 'The current value of counter in the outer block is
%', outer_block.counter;
END;
RAISE NOTICE 'The current value of counter in the outer block is
%', counter;
END outer_block $$;
```

In this example, we first declared a variable named counter in the outer\_block.

Next, in the subblock, we also declared a variable with the same name.

Then, before entering into the subblock, the value of counter is 1. In the subblock, we increased the value of the counter to 10 and print it out. Notice that the change only affects the counter variable in the subblock.

After that, we referred to the counter variable in the outer block using block label to qualify its name outer\_block.counter.

Finally, we print out the value counter variable in the outer block, its value remains intact.

In this tutorial, you have learned about the block structure of PL/pgSQL and how to execute a block using DO statement.

## 10.4 Errors and Messages

Summary: in this tutorial, we will show you how to report messages and raise errors using RAISE statement. In addition, we will introduce you to the ASSERT statement for inserting debugging checks into PL/pgSQL block. PL/pgSQL reporting messages

To raise a message, you use the RAISE statement as follows:

RAISE level format;
---------------------

Let's examine the components of the RAISE statement in more detail.

Followed the RAISE statement is the level option that specifies the error severity. There are following levels in PostgreSQL:

- DEBUG
- LOG
- NOTICE
- INFO
- WARNING
- EXCEPTION

If you don't specify the level, by default, the RAISE statement will use EXCEPTION level that raises an error and stops the current transaction. We will discuss about the RAISE EXCEPTION later in the next section.

The format is a string that specifies the message. The format uses percentage ( % ) placeholders that will be substituted by the next arguments. The number of placeholders must match the number of arguments, otherwise PostgreSQL will report the following error message:

```
[Err] ERROR: too many parameters specified for RAISE
```

The following example illustrates the RAISE statement that reports different messages at the current time.

Notice that not all messages are reported back to client, only INFO, WARNING, and NOTICE level messages are reported to client. This is controlled by the client\_min\_messages and log\_min\_messages configuration parameters. PL/pgSQL raising errors

To raise errors, you use the EXCEPTION level after the RAISE statement. Note that RAISE statement uses EXCEPTION level by default.

Besides raising an error, you can add more detailed information using the following clause with the RAISE statement:

```
USING option = expression
```

The option can be:

MESSAGE: to set error message text  
HINT: to provide the hint message so that the root cause of the error is easier to be discovered.  
DETAIL: to give detailed information about the error.  
ERRCODE: to identify the error code, which can be either by condition name or directly five-character SQLSTATE code. Please refer to the table of error codes and condition names.

The expression is a string-valued expression.

The following example raises a duplicate email error message:

The following examples illustrate how to raise a SQLSTATE and its corresponding condition:

```
DO $$  
BEGIN  
--...  
RAISE SQLSTATE '2210B';  
END $$;  
  
DO $$  
BEGIN  
--...  
RAISE invalid_regular_expression;  
END $$;
```

PL/pgSQL putting debugging checks using ASSERT statement

Notice that PostgreSQL introduces the ASSERT statement since version 9.5. Check your PostgreSQL version before using it.

Sometimes, a PL/pgSQL function is so big that make it more difficult to detect the bugs. To facilitate this, PostgreSQL provides you with the ASSERT statement for adding debugging checks into a PL/pgSQL function.

The following illustrates syntax of the ASSERT statement:

```
ASSERT condition [, message];
```

The condition is a boolean expression. If the condition evaluates to TRUE, ASSERT statement does nothing. If the condition evaluates to FALSE or NULL, the ASSERT\_FAILURE is raised.

If you don't provide the message, PL/pgSQL uses "assertion failed" message by default. If the message is provided, the ASSERT statement will use it to replace the default message.

```
DO $$  
DECLARE  
counter integer := -1;  
BEGIN  
ASSERT counter < 0  
MESSAGE 'Expect counter starts with 0';  
END $$;
```

It is important to note that the ASSERT statement is used for debugging only.

Now you can use RAISE statement to either raise a message or report an error.

---

# 11 Indizes



# 12 grant, revoke - UserManagement

Ein Datenbankserver verwaltet unter Umständen eine Vielzahl von Datenbanken, die von vielen Applikationen genutzt werden. Applikationen werden von vielen Benutzern genutzt, die unterschiedliche Berechtigungen besitzen sollten. Manchmal ist es nicht wünschenswert, dass Benutzer die Möglichkeit haben, andere Datenbanken als ihre eigenen zu sehen, insbesondere wenn sie per Administrationswerkzeugen Zugriff auf den Datenbankserver haben.

## 12.1 Aufgabe des Rechtesystems

Die Aufgabe des Rechtesystems besteht darin,

- einen User, der von einem bestimmten Host auf die Datenbank connected, zu authentifizieren.
- dem User für eine spezielle Datenbank mit seinen jeweiligen Rechten für beispielsweise INSERT, UPDATE, DELETE und SELECT zu versehen.
- ihn mit speziellen Rechten zu versorgen wie z.B. LOAD DATA INFILE und administrative Aufgaben

## 12.2 Funktionsweise des Rechtesystems

MySQL benutzt immer die Kombination aus Usernamen und Clienthost, um einen Anwender zu identifizieren.

	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Host	localhost	%	%	localhost
User	root	root		pma
Password				
Select_priv	Y	N	N	Y
Insert_priv	Y	N	N	Y
Update_priv	Y	N	N	Y
Delete_priv	Y	N	N	Y
Create_priv	Y	N	N	Y
Drop_priv	Y	N	N	Y
Reload_priv	Y	N	N	Y
Shutdown_priv	Y	N	N	Y
Process_priv	Y	N	N	Y
File_priv	Y	N	N	Y
Grant_priv	Y	N	N	Y
References_priv	Y	N	N	Y
Index_priv	Y	N	N	Y
Alter_priv	Y	N	N	Y
Show_db_priv	Y	N	N	Y
Super_priv	Y	N	N	Y
Create_tmp_table_priv	Y	N	N	Y
Lock_tables_priv	Y	N	N	Y
Execute_priv	Y	N	N	Y
Repl_slave_priv	Y	N	N	Y
Repl_client_priv	Y	N	N	Y
ssl_type				
ssl_cipher	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]
x509_issuer	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]
x509_subject	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]	[BLOB - 0 Bytes]
max_questions	0	0	0	0
max_updates	0	0	0	0
max_connections	0	0	0	0

Die Zugriffskontrolle in MySQL läuft in 2 verschiedenen Phasen ab.

1. Der Server überprüft, ob es dem User erlaubt ist, sich auf den Datenbankserver zu verbinden. Der Server benutzt dazu die Tabelle **user**.

Diese Tabelle benutzt alle Host/User Kombinationen, welche den MySQL-Server connecten dürfen. Alle Berechtigungen die ein Benutzer in dieser Tabelle enthält gelten für alle Datenbanken, sofern keine erweiterten Berechtigungen für den jeweiligen Benutzer in der Tabelle **db** definiert wurden. Man kann diese Berechtigungen auch als grundlegende Einstellungen ansehen und ein datenbankabhängiges Fein-Tuning in der Tabelle db festlegen.

	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Host	%	%
Db	test	test, %
User		
Select_priv	Y	Y
Insert_priv	Y	Y
Update_priv	Y	Y
Delete_priv	Y	Y
Create_priv	Y	Y
Drop_priv	Y	Y
Grant_priv	N	N
References_priv	Y	Y
Index_priv	Y	Y
Alter_priv	Y	Y
Create_tmp_table_priv	Y	Y
Lock_tables_priv	Y	Y

2. Wenn Phase 1 erfolgreich abgelaufen ist, wird vom Server jedes Statement daraufhin geprüft, ob der jeweilige User genügend Rechte hat, um dieses Statement auszuführen. Der Server benutzt dabei die Tabelle db.

### host-Tabelle

	<input type="checkbox"/>
Host	localhost
Db	
Select_priv	Y
Insert_priv	Y
Update_priv	Y
Delete_priv	Y
Create_priv	Y
Drop_priv	Y
Grant_priv	Y
References_priv	Y
Index_priv	Y
Alter_priv	Y
Create_tmp_table_priv	Y
Lock_tables_priv	Y

Die host-Tabelle ist in großen Netzwerken als Nachschlagetabelle für leere Host-Einträge in der db-Tabelle sinnvoll. Möchte man, daß ein Benutzer von jedem Host in dem Netzwerk auf den DB-Server zugreifen kann, sollte man den Host-Eintrag in der db-Tabelle auslassen und alle Host des Netzwerkes in der host-Tabelle eintragen. Sie wird in neuen MySQL-Versionen nicht mehr genutzt.

Sollten diese Rechte zur Laufzeit einer Verbindung geändert werden, so bedeutet es nicht, dass diese Änderungen automatisch geändert sind. Da der Server die Rechte beim Start aus den jeweiligen Tabellen liest, muss man ihn dazu auffordern, diese Rechte neu laden.

### Fein-Tuning per tables\_priv und columns\_priv

Feld	Typ	Funktion	Null	
Host	char(60)			
Db	char(64)			
User	char(16)			
Table_name	char(64)			
Column_name	char(64)			
Timestamp	timestamp	NOW	<input type="checkbox"/>	CURRENT_TIME
Column_priv	set	--		Select Insert Update References

In der 2. Phase können zusätzlich die Tabellen **tables\_priv** und **columns\_priv** abgefragt werden, wenn sich der Request auf Tabellen bezieht.

Field	Type	
Host	char(60)	
Db	char(64)	
User	char(16)	
Table_name	char(64)	
Grantor	char(77)	
Timestamp	timestamp	
		set('Select', 'Insert', 'Update', 'Delete', 'Create', 'Drop', 'Grant', 'References', 'Index', 'Alter')
Table_priv		
Column_priv		set('Select', 'Insert', 'Update', 'References')

## 12.3 Definitionen

Die HOST und DB Spalten können Strings mit Wildcards % und \_ beinhalten. Wird für diese Spalten kein Wert eingetragen entspricht dies dem Wert %. Ein HOST kann sein:

- localhost
- ein Hostname
- eine IP-Nummer
- ein String mit Wildcards

Ein leerer HOST-Eintrag in der db-Tabelle bedeutet -kein Host- aus der host-Tabelle. Ein leerer HOST-Eintag in der host- oder user-Tabelle bedeutet -kein Host-.

Die Spalte DB beinhaltet den Namen einer Datenbank oder einer SQL Regexp.

Ein leerer Benutzereintrag bedeutet -kein Benutzer-. In dieser Spalte können keine Wildcards verwendet werden.

Die Berechtigungen der user-Tabelle werden ge-OR-d mit den Berechtigungen aus der db-Tabelle. Dies Bedeutet, daß ein Superuser nur in der Tabelle user mit allen Berechtigungen festgelegt auf Y eingetragen werden muß.

Wenn man sich nun den Aufbau der Tabellen näher betrachtet, wird man feststellen, daß die user-Tabelle zusätzlich zu den Zugriffsberechtigungen auf die jeweilige Datenbank auch administrative Berechtigungen regelt. Dadurch sollte klar sein, dass diese Tabelle die grundlegenden Berechtigungen regelt.

## 12.4 Benutzerkonten mit GRANT und REVOKE erstellen.

```
GRANT priv_type [(column_list)] [, priv_type [(column_list)]]  
...  
    ON {tbl_name | * | *.* | db_name.*}  
    TO user [IDENTIFIED BY [PASSWORD] 'password']  
        [, user [IDENTIFIED BY [PASSWORD] 'password']] ...  
    [REQUIRE  
        NONE |  
        [{SSL| X509}]  
        [CIPHER 'cipher' [AND]]  
        [ISSUER 'issuer' [AND]]  
        [SUBJECT 'subject']]  
    [WITH [GRANT OPTION | MAX_QUERIES_PER_HOUR count |  
          MAX_UPDATES_PER_HOUR count |  
          MAX_CONNECTIONS_PER_HOUR count]]  
  
REVOKE priv_type [(column_list)] [, priv_type [(column_list)]]  
...  
    ON {tbl_name | * | *.* | db_name.*}  
    FROM user [, user] ...  
  
    REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

### 12.4.1 Beispiel für die Vergabe von Benutzerrechten

```
GRANT SELECT ON testgrant.* TO 'jim'@'localhost' IDENTIFIED BY
'Abc123'
```

**SQL-Befehl:**

```
SELECT *
FROM user
LIMIT 0 , 30
```

[Bearbeiten] [SQL erklären] [PHP-Code erzeugen] [Aktualisieren]

Zeige : 30 Datensätze, beginnend ab 0  
nebeneinander  angeordnet und wiederhole die Kopfzeilen nach 100  
Datensätzen.

Nach Schlüssel sortieren: keine

Host	localhost	%	localhost	%	localhost
User	root	root	jim	pma	
Password	66e77e8a4cb25c1b				
Select_priv	Y	N	N	N	Y
Insert_priv	Y	N	N	N	Y

Erzeugt ein **SELECT-Privileg** für alle Tabellen in der Datenbank **testgrant** für eine User namens **jim**. jim muss sich von **localhost** verbinden und das Passwort **Abc123** benutzen:

**SQL-Befehl:**

```
SELECT *
FROM `db`
LIMIT 0 , 30
```

[Bearbeiten] [SQL erklären] [PHP-Code erzeugen] [Aktualisieren]

Zeige : 30 Datensätze, beginnend ab 0  
untereinander  angeordnet und wiederhole die Kopfzeilen nach 100  
Datensätzen.

Nach Schlüssel sortieren: keine

Host	Db	User	Select_priv	Insert_priv	Update_priv
localhost	test		Y	Y	Y
localhost	test\_%		Y	Y	Y
localhost	testgrant	jim	Y	N	N

```
GRANT SELECT, INSERT, DELETE, UPDATE ON world.* TO 'jim'@'localhost'
IDENTIFIED BY 'Abc123';
```

Mehrere Rechte werden vergeben

```
GRANT SELECT (ID, Name, CountryCode), UPDATE (Name, CountryCode)
ON world.City TO 'jim'@'localhost' IDENTIFIED BY 'Abc123';
```

Die **IDENTIFIED BY** -Klausel ist optional. Ist sie vorhanden, weist sie einem User ein Passwort zu. Hat der User bereits ein Passwort, so wird es überschrieben. Wenn der User bereits existiert und die Klausel weggelassen wird, so bleibt das gegenwärtige Passwort des Users wie es ist. Rechte auf spezielle Spalten einer Tabelle.

```
GRANT SELECT ON world.* TO ''@'localhost';
```

Anonymer Zugriff auf alle Tabellen der Tabelle world

```
GRANT SELECT ON world.* TO 'jim'@'localhost' IDENTIFIED BY  
'Abc123' WITH GRANT OPTION;
```

SELECT-Rechte auf die Datenbank world mit der Möglichkeit, dieses Recht weiterzugeben.

```
mysql> SHOW GRANTS FOR jim@localhost
mysql> Grants for jim@localhost
      GRANT USAGE ON *.* TO 'jim'@'localhost' IDENTIFIED...
      GRANT SELECT ON `testgrant`.* TO 'jim'@'localhost'
mysql> SHOW GRANTS FOR jim@192.168.0.140
#1141 - There is no such grant defined for user 'jim' on host '192.168.0.140'
```

Zeigt alle Rechte für den jeweiligen User laut Eintrag in User-Tabelle an.

## 12.4.2 REVOKE - Wegnahme von Benutzerrechten

Das REVOKE statement nimmt die Privilegien eines Accounts. Seine Syntax hat die folgenden Abschnitte:

- Das Schlüsselwort **REVOKE**, gefolgt von der Liste der wegzunehmenden Rechte
- Die **ON** - Klausel, die den Level angibt, an der die Rechte entzogen werden.
- Die **FROM**-Klausel, die den Usernamen spezifiziert

```
REVOKE DELETE, INSERT, UPDATE ON world.* FROM 'jim'@'localhost';
REVOKE GRANT OPTION ON world.* FROM 'jill'@'localhost';

mysql> SHOW GRANTS FOR 'jen'@'myhost.example.com';

+-----+
| Grants for jen@myhost.example.com |
+-----+
| GRANT FILE ON *.* TO 'jen'@'myhost.example.com' |
| GRANT SELECT ON `mydb`.* TO 'jen'@'myhost.example.com' |
| GRANT UPDATE ON `test`.`mytable` TO 'jen'@'myhost.example.com' |
+-----+

mysql> REVOKE FILE ON *.* FROM 'jen'@'myhost.example.com';
mysql> REVOKE SELECT ON mydb.* FROM 'jen'@'myhost.example.com';
mysql> REVOKE UPDATE ON test.mytable FROM 'jen'@'myhost.example.com';

mysql> SHOW GRANTS FOR 'jen'@'myhost.example.com';

+-----+
| Grants for jen@myhost.example.com |
+-----+
| GRANT USAGE ON *.* TO 'jen'@'myhost.example.com' |
+-----+

mysql> USE mysql;
mysql> DELETE FROM user WHERE User = 'jen' AND Host = 'myhost.example.com';
mysql> FLUSH PRIVILEGES;
```

Bearbeiten Sie folgende **Aufgabe**.

# 13 Aufgaben zu SQL

## 13.1 LUNA

1. Verschaffen sie sich einen Überblick über die Tabelle `tblabteilung`
2. Lassen sie sich alle Mitarbeiter mit Namen und Vorname anzeigen
3. Finden Sie den Namen und Nummer aller Abteilungen mit Sitz in München.
4. Nennen Sie die Namen und Vornamen aller Mitarbeiter, deren Personalnummer mindestens 15000 ist
5. Finden Sie alle Projekte, deren Finanzmittel mehr als 6000 US\$ betragen. Der derzeitige Kurs beträgt: 1 US\$ = 0,9875 €
6. Gesucht werden Personalnummer, Projektnummer und Aufgabe der Mitarbeiter, die im Projekt p2 Sachbearbeiter sind.
7. Gesucht wird die Personalnummer der Mitarbeiter, die entweder im Projekt p1 oder p2 oder in beiden tätig sind
8. Gesucht wird die Personalnummer der Mitarbeiter, die entweder im Projekt p1 oder p2 oder in beiden tätig sind. Jeder Treffer soll nur einmal erscheinen.
9. Nennen Sie Personalnummer und Nachnamen der Mitarbeiter, die nicht in der Abteilung a2 arbeiten
10. Finden Sie alle Mitarbeiter mit Personalnummer 29346, 28559 oder 25348.
11. Nennen Sie alle Mitarbeiter, deren Personalnummer weder 10102 noch 9031 ist
12. Nennen Sie Namen und Mittel aller Projekte, deren Etat zwischen 95000,00 € und 120000,00 € liegt.
13. Nennen Sie die Personalnummer aller Mitarbeiter, die Projektleiter sind und vor oder nach 1988 eingestellt worden sind. (d.h. nicht im Jahr 1988)
14. Finden Sie die Personal- und Projektnummer aller Mitarbeiter, die im Projekt p1 arbeiten und deren Aufgabe noch nicht festgelegt ist.
15. Finde Namen u. Personalnummer der Mitarbeiter, deren Name mit "K" beginnt.
16. Finden Sie Namen, Vornamen und Personalnummer aller Mitarbeiter, deren Vornamen als zweiten Buchstaben ein "a" hat
17. Finden Sie die Abteilungsnummer und Standorte aller Abteilungen, die sich in den Orten, die mit einem Buchstaben zwischen E und N beginnen, befinden.
18. Finde Namen, Vornamen und Personalnr aller Mitarbeiter, deren Namen nicht mit den Buchstaben M, N, O und P, und deren Vornamen nicht mit H beginnt.
19. Nennen Sie alle Mitarbeiter, deren Name nicht mit "mann" endet
20. Nennen Sie die Abteilungsnummer des Mitarbeiters, dessen Personalnummer 2581 ist und der im Projekt p3 arbeitet.
21. Nennen Sie die Nummern aller Projekte, in welchen Mitarbeiter arbeiten, deren Personalnummer kleiner als die Nummer des Mitarbeiters Müller ist.
22. Nennen Sie die Daten aller Mitarbeiter, die in München arbeiten.

23. Nennen Sie die Namen aller Mitarbeiter, die im Projekt Apollo arbeiten
24. Gruppieren Sie die Mitarbeiter nach Projektnummer und Aufgabe
25. Nennen Sie die kleinste Personalnummer eines Mitarbeiters
26. Nenne Personalnr. und Namen des Mitarbeiters mit der kleinsten Personalnr.
27. Finden Sie die Personalnummer des Projektleiters, der in dieser Position als letzter eingestellt wurde
28. Berechnen Sie die Summe der finanziellen Mittel aller Projekte
29. Berechne das arithmetische Mittel der Geldbeträge, die höher als 100000 € sind.
30. Wie viele verschiedenen Aufgaben werden in jedem Projekt ausgeübt.
31. Finden Sie heraus, wie viele Mitarbeiter in jedem Projekt arbeiten
32. Erstellen Sie eine Übersicht, wie viele Mitarbeiter welcher Tätigkeit nachgehen.
33. Nennen Sie alle Projekte, in denen weniger als vier Mitarbeiter beschäftigt sind.
34. Finden Sie für jeden Mitarbeiter, zusätzlich zu seiner Personalnummer, Namen und Vornamen, auch die Abteilungsnummer und den Standort der Abteilung
35. Finden Sie die Daten der Mitarbeiter, die im Projekt Gemini arbeiten.
36. Nenne die Abteilungsnummern aller am 15.10.1989 eingestellten Mitarbeiter.
37. Nennen Sie Namen und Vornamen aller Projektleiter, deren Abteilung den Standort Stuttgart hat.
38. Nennen Sie die Namen der Projekte, in denen Mitarbeiter tätig sind, die zur Abteilung Diagnose gehören.
39. Finden Sie alle Abteilungen, an deren Standorten sich weitere Abteilungen befinden.
40. Betrachten Sie folgende DDL- und SELECT-Statements. Welches Ergebnis erhalten Sie für die SELECT-Statements

```
drop table if exists nulltest ;
create table nulltest (
c1 int null
);

insert into nulltest(c1) values(1);
insert into nulltest(c1) values(null);
insert into nulltest(c1) values(3);

select avg(c1) from nulltest;

select sum(c1)/count(*) from nulltest;
select sum(c1)/count(c1) from nulltest;
```

### 13.1.1 Luna\_Lösung

1. Verschaffen sie sich einen Überblick über die Tabelle tblabteilung

```
SELECT * from tblabteilung
```

2. Lassen sie sich alle Mitarbeiter mit Namen und Vorname anzeigen

```
SELECT M_Name, M_Vorname  
FROM tblmitarbeiter
```

3. Finden Sie den Namen und Nummer aller Abteilungen, die ihren Sitz in München haben.

```
SELECT A_Name, A_Stadt  
FROM tblabteilung  
WHERE A_Stadt = 'München'
```

4. Nennen Sie die Namen und Vornamen aller Mitarbeiter, deren Personalnummer mindestens 15000 ist

```
SELECT M_Name, M_Vorname  
FROM tblmitarbeiter  
WHERE M_Nr >= 15000
```

5. Finden Sie alle Projekte, deren Finanzmittel mehr als 6000 US\$ betragen. Der derzeitige Kurs beträgt: 1 US\$ = 0,9875 €

```
SELECT P_Name  
FROM tblprojekt  
WHERE P_Mittel * 0.9875 > 60000
```

6. Gesucht werden Personalnummer, Projektnummer und Aufgabe der Mitarbeiter, die im Projekt p2 Sachbearbeiter sind.

```
SELECT M_Nr, P_Nr, T_Taetigkeit  
FROM tbltaetigkeit  
WHERE P_Nr = 'p1'  
      AND T_Taetigkeit = 'Sachbearbeiter'
```

7. Gesucht wird die Personalnummer der Mitarbeiter, die entweder im Projekt p1 oder p2 oder in beiden tätig sind

```
SELECT M_Nr
FROM tbltaetigkeit
WHERE P_Nr = 'p1'
OR P_Nr = 'p2'
```

8. Gesucht wird die Personalnummer der Mitarbeiter, die entweder im Projekt p1 oder p2 oder in beiden tätig sind. Jeder Treffer soll nur einmal erscheinen.

```
SELECT DISTINCT M_Nr
FROM tbltaetigkeit
WHERE P_Nr = 'p1'
OR P_Nr = 'p2'
```

9. Nennen Sie Personalnummer und Nachnamen der Mitarbeiter, die nicht in der Abteilung a2 arbeiten

```
SELECT M_nr, M_Name
FROM tblmitarbeiter
WHERE A_Nr <> 'a2'
```

10. Finden Sie alle Mitarbeiter, deren Personalnummer entweder 29346, 28559 oder 25348 ist

```
SELECT *
FROM tblmitarbeiter
WHERE M_Nr in (29346, 28559, 25348)
```

11. Nennen Sie alle Mitarbeiter, deren Personalnummer weder 10102 noch 9031 ist

```
SELECT *
FROM tblmitarbeiter
WHERE M_Nr NOT IN (10102, 9031)
```

12. Nennen Sie Namen und Mittel aller Projekte, deren Etat zwischen 95000,00 € und 120000,00 € liegt.

```
SELECT P_Name, P_Mittel
FROM tblprojekt
WHERE P_Mittel BETWEEN 95000 AND 1200000
```

13. Nennen Sie die Personalnummer aller Mitarbeiter, die Projektleiter sind und vor oder nach 1988 eingestellt worden sind. (d.h. nicht im Jahr 1988)

```
SELECT M_Nr
FROM tbltaetigkeit
WHERE T_Taetigkeit = 'Projektleiter'
AND T_Einstellungsdatum NOT BETWEEN '01.01.1988' and
'31.12.1988'
```

14. Finden Sie die Personal- und Projektnummer aller Mitarbeiter, die im Projekt p1 arbeiten und deren Aufgabe noch nicht festgelegt ist.

```
SELECT M_nr, P_Nr
FROM tbltaetigkeit
WHERE P_Nr = 'p1'
AND T_Taetigkeit is NULL
```

15. Finden Sie Namen und Personalnr. aller Mitarbeiter, deren Name mit "K" beginnt.

```
SELECT M_Name, M_Vorname, M_Nr
FROM tblmitarbeiter
WHERE M_Name LIKE 'K%'
```

16. Finden Sie Namen, Vornamen und Personalnummer aller Mitarbeiter, deren Vornamen als zweiten Buchstaben ein "a" hat.

```
SELECT M_Name, M_Vorname, M_Nr
FROM tblmitarbeiter
WHERE M_Vorname LIKE '_a%'
```

17. Finden Sie die Abteilungsnummer und Standorte aller Abteilungen, die sich in den Orten, die mit einem Buchstaben zwischen E und N beginnen, befinden.

```
SELECT A_Nr, A_Stadt
FROM tblabteilung
WHERE A_Stadt regexp '^ [E-N] '
```

18. Finden Sie Namen, Vornamen und Personalnr. aller Mitarbeiter, deren Namen nicht mit den Buchstaben M,N,O und P, und deren Vornamen nicht mit H beginnt.

```
SELECT M_Name, M_Vorname, M_Nr
FROM tblmitarbeiter
WHERE M_Name regexp '[^M-P]%'
AND M_Vorname regexp '[^H]%'
```

19. Nennen Sie alle Mitarbeiter, deren Name nicht mit "mann" endet.

```
SELECT *
FROM tblmitarbeiter
WHERE M_Name NOT LIKE '%mann'
```

20. Nennen Sie die Abteilungsnummer des Mitarbeiters, dessen Personalnummer 2581 ist und der im Projekt p3 arbeitet.

```
SELECT A_Nr
FROM tblmitarbeiter
WHERE M_Nr =
(SELECT M_Nr
FROM tbltaetigkeit
WHERE M_Nr = 2581
AND P_Nr = 'p3')
```

21. Nennen Sie die Nummern aller Projekte, in welchen Mitarbeiter arbeiten, deren Personalnummer kleiner als die Nummer des Mitarbeiters Müller ist.

```
SELECT DISTINCT P_Nr
FROM tbltaetigkeit
WHERE M_Nr <
(SELECT M_Nr
FROM tblmitarbeiter
WHERE M_Name = 'Müller')
```

22. Nennen Sie die Daten aller Mitarbeiter, die in München arbeiten.

```
SELECT *
FROM tblmitarbeiter
WHERE A_Nr IN
    (SELECT A-Nr
     FROMtblabteilung
     WHERE A_Stadt = 'München'
```

23. Nennen Sie die Namen aller Mitarbeiter, die im Projekt Apollo arbeiten.

```
SELECT M_Name
FROMtblmitarbeiter
WHERE M_Nr IN
    (SELECT M_Nr
     FROMtbltaetigkeit
     WHERE P_Nr =
        (SELECT P_Nr
         FROMtblprojekt
         WHERE P_Name = 'Apollo'
        )
    )
```

24. Gruppieren Sie die Mitarbeiter nach Projektnummer und Aufgabe.

```
SELECT P_Nr, T_Taetigkeit
FROMtbltaetigkeit
GROUP BY P_Nr, T_Taetigkeit
```

25. Nennen Sie die kleinste Personalnummer eines Mitarbeiters.

```
SELECT MIN(M_Nr) as Kleinster
FROMtblmitarbeiter
```

26. Nennen Sie Personalnr und Namen des Mitarbeiters mit der kleinsten Personalnr.

```
SELECT M_Nr, M_Name
FROMtblmitarbeiter
WHERE M_Nr = (
    SELECT MIN(M_Nr)
    FROMtblmitarbeiter
)
```

27. Finden Sie die Personalnummer des Projektleiters, der in dieser Position als letzter eingestellt wurde

```
SELECT M_Nr
FROM tblmitarbeiter
WHERE M_Nr = (
    SELECT MAX(T_Einstellungsdatum)
    FROM tbltaetigkeit
    WHERE T_Taetigkeit = 'Projektleiter'
)
```

28. Berechnen Sie die Summe der finanziellen Mittel aller Projekte.

```
SELECT SUM(P_Mittel)
FROM tblprojekt
```

29. Berechnen Sie das arithmetische Mittel der Geldbeträge, die höher als 100000 € sind.

```
SELECT AVG(P_Mittel)
FROM tblprojekt
WHERE P_Mittel > 100000
```

30. Finden Sie heraus, wie viele verschiedene Aufgaben in jedem Projekt ausgeübt werden.

```
SELECT P_Nr, COUNT(DISTINCT T_Taetigkeit) as Anzahl
FROM tbltaetigkeit
GROUP BY P_Nr
```

31. Finden Sie heraus, wie viele Mitarbeiter in jedem Projekt arbeiten.

```
SELECT P_Nr, COUNT(*)
FROM tbltaetigkeit
GROUP BY P_Nr

-- Hinweis: (*) berücksichtigt auch NULL-Werte --
```

32. Erstellen Sie eine Übersicht, wie viele Mitarbeiter welcher Tätigkeit nachgehen.

```
SELECT T_Taetigkeit, COUNT(*) as Mitarbeiteranzahl  
FROM tbltaetigkeit  
GROUP BY T_Taetigkeit
```

33. Nennen Sie alle Projekte, in denen weniger als vier Mitarbeiter beschäftigt sind.

```
Group by .... Having count ... < 4
```

34. Finden Sie für jeden Mitarbeiter, zusätzlich zu seiner Personalnummer, Namen und Vornamen, auch die Abteilungsnummer und den Standort der Abteilung.

```
SELECT tblmitarbeiter.*,tblabteilung.*  
FROM tblmitarbeiter,tblabteilung  
WHERE tblmitarbeiter.A_Nr =tblabteilung.A_Nr
```

35. Finden Sie die Daten der Mitarbeiter, die im Projekt Gemini arbeiten.

```
SELECT *  
FROM tblmitarbeiter,tbltaetigkeit,tblprojekt  
WHERE tblmitarbeiter.M_Nr =tbltaetigkeit.M_Nr  
AND tbltaetigkeit.P_Nr =tblprojekt.P_Nr  
and P_Name = 'Gemini'
```

36. Nennen Sie die Abteilungsnummern aller Mitarbeiter, die am 15.10.1989 eingestellt wurden.

```
SELECT A_Nr  
FROM tblmitarbeiter,tbltaetigkeit  
WHERE tbltaetigkeit.M_Nr =tblmitarbeiter.M_Nr  
AND T_Einstellungsdatum = '1989-10-15'
```

37. Nennen Sie Namen und Vornamen aller Projektleiter, deren Abteilung den Standort Stuttgart hat.

```
SELECT M_Name, M_Vorname
FROM tblmitarbeiter, tbltaetigkeit,tblabteilung
WHERE tbltaetigkeit.M_Nr = tblmitarbeiter.M_Nr
AND tblmitarbeiter.A_Nr =tblabteilung.A_Nr
AND T_TAetigkeit = 'Projektleiter'
AND A_Stadt = 'Stuttgart'
```

38. Nennen Sie die Namen der Projekte, in denen Mitarbeiter tätig sind, die zur Abteilung Diagnose gehören.

```
SELECT DISTINCT P_Name
FROMtblabteilung,tblmitarbeiter,tblprojekt,tbltaetigkeit
WHEREtblprojekt.P_Nr =tbltaetigkeit.P_Nr
ANDtbltaetigkeit.M_Nr =tblmitarbeiter.M_Nr
ANDtblmitarbeiter.A_Nr =tblabteilung.A_Nr
ANDA_Name = 'Diagnose'
```

39. Finden Sie alle Abteilungen, an deren Standorten mit weiteren Abteilungen.

```
SELECT Temp1.A_Nr, Temp2.A_Name, Temp1.A_Stadt
FROMtblabteilung Temp1,tblabteilung Temp2
WHERETemp1.A_Stadt = Temp2.A_Stadt
ANDTemp1.A_Nr <> Temp2.A_Nr
```

40. Betrachten Sie folgende DDL- und SELECT-Statements. Welches Ergebnis erhalten Sie für die SELECT-Statements

```
drop table if exists nulltest ;
create table nulltest (
    c1 int null);

insert into nulltest(c1) values(1);
insert into nulltest(c1) values(null);
insert into nulltest(c1) values(3);
select avg(c1) from nulltest;
select sum(c1)/count(*) from nulltest;
select sum(c1)/count(c1) from nulltest;
```

Während die **AVG**-Funktion **NULL**-Werte **in** Feldern berücksichtigt, gibt es bei der **COUNT()**-Funktion zwei unterschiedliche Verhaltensweisen.

**COUNT(\*)** berücksichtigt auch **NULL**-Werte und gibt als Ergebnis die Anzahl **3** zurück (**4/3 = 1.33**)

**COUNT(c1)** klammert **NULL**-Werte aus und gibt als Ergebnis die Anzahl **2** zurück (**4/2 = 2**)

## 13.2 Nordwind



1. Erstellen Sie eine Liste aller Mitarbeiter, die jünger als 40 Jahre sind
2. Erstellen Sie eine Liste aller Mitarbeiterinnen aus London
3. Erstellen Sie eine Liste aller Kunden, wo der Inhaber gleichzeitig Kontaktperson ist
4. Erstellen Sie eine Liste aller Kunden nach Ländern angeordnet, die Städte in alphabetischer Reihenfolge.
5. Erstellen Sie eine Liste aller Lieferanten, die noch keine Homepage haben.
6. Erstellen Sie eine Liste aller Kunden, die irgendeine Art von Camembert kauften
7. Ermitteln Sie unsere drei umsatzstärksten Kunden. Beginnen Sie zunächst mit dem umsatzstärksten Kunden.
8. Mit welcher Lieferfirma machen wir den geringsten Umsatz
9. Wer hat Gnocchi gekauft, deren Preis über dem durchschnittlichen Verkaufspreis lag.
10. Welcher Kunde musste am längsten auf seine Lieferung warten. Wie lange ?
11. Wie lange dauert der Versand im Schnitt bei unseren Versandfirmen
12. Erstellen Sie eine Liste, wie hoch die Frachtkosten unserer Versandfirmen 1996 waren
13. Erstellen Sie eine Liste der Kunden, deren Waren noch nicht versendet wurden
14. Ermitteln Sie, wie oft unsere Kunden 1998 im Mittel bestellt haben.
15. Stellen Sie fest, wie oft LILA Supermercado 1996 über 1000,00 € bestellt hat
16. Stellen Sie fest, wie viele Sendungen in die USA gingen
17. Ermitteln Sie, wie viele verschiedenen Länder wir beliefern
18. Berechnen Sie, wie viel Umsatz wir mit skandinavischen Lieferanten machen
19. Erstellen Sie eine Liste, welche Kunden 1995 länger als 10 Tage auf ihre Lieferung mit SpeedExpress warten mussten.
20. Erstellen Sie eine Liste mit dem Umsatz je Kategorie, die mit dem Minimum beginnt

### 13.2.1 Lösung

1. Erstellen Sie eine Liste aller Mitarbeiter, die jünger als 40 Jahre sind

```
SELECT
`personal`.`Geburtsdatum`, `personal`.`Vorname`,
`personal`.`Nachname`
FROM `personal`
WHERE NOW() <= DATE_ADD(`personal`.`Geburtsdatum`, INTERVAL
"40" YEAR);

=====
oder etwas unflexibler

SELECT .....
FROM Personal
WHERE Personal.Geburtsdatum > YYYY/MM/DD -- (Jahr/Monat/Tag
einsetzen)
```

2. Erstellen Sie eine Liste aller Mitarbeiterinnen aus London

```
SELECT Vorname, Nachname
FROM Personal
where Personal.Ort = 'London'
and Perdsonal.Anrede LIKE 'Frau'
```

3. Erstellen Sie eine Liste aller Kunden, wo der Inhaber gleichzeitig Kontaktperson ist

```
SELECT Firma from Kunden where Position LIKE 'Inhaber'
```

4. Erstellen Sie eine Liste aller Kunden nach Ländern angeordnet, die Städte in alphabetischer Reihenfolge.

```
Select Firma, Land, Ort from Kunden order by Land, Ort
```

5. Erstellen Sie eine Liste aller Lieferanten, die noch keine Homepage haben.

```
Select Firma, Homepage FROM LIEFERANTEN
WHERE Homepage is null
or Hoempage = ''
```

6. Erstellen Sie eine Liste aller Kunden, die irgendeine Art von Camembert gekauft haben

```
SELECT DISTINCT Firma, Artikelname
FROM Kunden, Bestellungen, Bestelldetails, Artikel
WHERE Kunden.KundenCode = Bestellungen.KundenCode
AND Bestellungen.BestellNr = Bestelldetails.BestellNr
AND bestelldetails.ArtikelNr = Artikel.ArtikelNr
AND Artikelname like '%Camembert%'

===== inner join
SELECT DISTINCT Firma, Artikelname
from Kunden
INNER JOIN bestellungen on (kunden.kundencode = bestellungen.kundencode)
INNER JOIN bestelldetails on (bestellungen.bestellnr = bestelldetails.bestellnr)
INNER JOIN artikel on bestelldetails.artikelnr = artikel.artikelnr
WHERE artikel.artikelname LIKE '%Camembert%'
```

7. Ermitteln Sie unsere drei umsatzstärksten Kunden. Beginnen Sie zunächst mit dem umsatzstärksten Kunden.

```
SELECT
kunden.firma, SUM(bestelldetails.Einzelpreis * bestelldetails.Anzahl) as Umsatz
FROM
bestelldetails
INNER JOIN bestellungen ON (bestelldetails.bestellnr = bestellungen.bestellnr)
INNER JOIN kunden ON (bestellungen.KundenCode = kunden.KundenCode)
GROUP BY Kunden.Firma
ORDER BY Umsatz DESC
LIMIT 0,3
```

8. Mit welcher Lieferfirma machen wir den geringsten Umsatz

```
SELECT
lieferanten.Firma,
SUM(bestelldetails.einzelpreis * bestelldetails.anzahl) as
Umsatz
FROM bestelldetails
INNER JOIN bestellungen ON (bestelldetails.BestellNr = be-
stellungen.BestellNr)
INNER JOIN artikel ON (bestelldetails.Artikelnr = arti-
kel.artikelnr)
INNER JOIN lieferanten ON (Artikel.lieferantenNr = Lieferan-
ten.LieferantenNr)
GROUP BY lieferanten.Firma
ORDER BY Umsatz
LIMIT 0,1
```

9. Wer hat Gnocchi gekauft, deren Preis über dem durchschnittlichen Verkaufspreis lag.

```
Select distinct Firma, Artikelname,bestelldetails.einzelpreis
from Kunden, Bestellungen, Bestelldetails, Artikel
where Kunden.KundenCode = Bestellungen.KundenCode
and Bestellungen.BestellNr = Bestelldetails.BestellNr
and bestelldetails.ArtikelNr = Artikel.ArtikelNr
and Artikelname like 'Gnocchi%'
and bestelldetails.einzelpreis > (select avg(bestelldetails.ein-
zelpreis) from bestelldetails, artikel
where bestelldetails.artikelnr = artikel.artikelnr
and Artikelname like 'Gnocchi%')
#####
Lösung geht nicht
SELECT
AVG(bestelldetails.einzelpreis) as Durchschnittspreis,
artikel.artikelname, kunden.firma
FROM bestellungen
INNER JOIN bestelldetails
ON (bestellungen.bestellnr=bestelldetails.bestellnr)
INNER JOIN artikel
ON (bestelldetails.artikelnr=artikel.Artikelnr)
INNER JOIN kunden ON (bestellungen.kundencode=kunden.kundencode)
WHERE artikel.artikelname LIKE 'Gnocchi%'
AND (SELECT AVG(bestelldetails.eizelpreis AS Durchschnitt2) >
(bestelldetails.einzelpreis)
GROUP BY artikel.artikelname
```

10. Welcher Kunde musste am längsten auf seine Lieferung warten. Wie lange ?

```
SELECT distinct Firma, Lieferdatum-Bestelldatum as wartezeit
FROM Kunden, bestellungen
WHERE Kunden.Kundencode = bestellungen.kundencode
AND lieferdatum-bestelldatum =
      (SELECT MAX(Lieferdatum - bestelldatum) FROM Kunden,
       bestellungen
      where kunden.kundencode = bestellungen.kundencode)
```

11. Wie lange dauert der Versand im Schnitt bei unseren Versandfirmen

```
select firma, avg(Lieferdatum-Versanddatum)
from versandfirmen, bestellungen
where versandfirmen.firmennr = bestellungen.versandueber
group by firma
```

12. Erstellen Sie eine Liste, wie hoch die Frachtkosten der Versandfirmen 1996 waren

```
Select firma, sum(frachtkosten)
from versandfirmen, bestellungen
where versandfirmen.firmennr = bestellungen.versandueber
and year(Lieferdatum) = 1996
group by firma
```

13. Erstellen Sie eine Liste der Kunden, deren Waren noch nicht versendet wurden

```
select distinct firma
from kunden, bestellungen
where kunden.kundencode = bestellungen.kundencode
and versanddatum is null
```

14. Ermitteln Sie, wie oft unsere Kunden 1998 im Mittel bestellt haben.

```
select Firma, count(bestellnr) from kunden, bestellungen
where kunden.kundencode = bestellungen.kundencode
and year(bestelldatum) = 1994
group by Firma
```

15. Stellen Sie fest, wie oft LILA Supermercado 1996 über 1000,00 € bestellt hat

```
select sum(einzelpreis*anzahl) - (einzelpreis*anzahl*rabatt)
as bestellwert
from bestelldetails, bestellungen, kunden
where bestelldetails.bestellnr = bestellungen.bestellnr
and bestellungen.kundencode = kunden.kundencode
and firma like 'lila%'
and year(Bestelldatum) = 1998
group by bestellungen.bestellnr
having sum((einzelpreis*anzahl) - (einzelpreis*anzahl*Rabatt)) > 1000
```

16. Stellen Sie fest, wie viele Sendungen in die USA gingen

```
select count(Bestimmungsland) as Lieferungen_USA from bestellungen
where Bestimmungsland= 'USA'
```

17. Ermitteln Sie, wie viele verschiedenen Länder wir beliefern

```
select distinct bestimmungsland as länderliste from bestellungen
```

18. Berechnen Sie, wie viel Umsatz wir mit skandinavischen Lieferanten machen

```
SELECT SUM(Einzelpreis*Anzahl) - (Einzelpreis*Anzahl*Rabatt)
as Umsatz_Skandinavien
FROM bestelldetails, bestellungen, kunden
WHERE bestelldetails.bestellnr = bestellungen.bestellnr
AND bestellungen.kundencode = kunden.kundencode
AND kunden.land in ('Schweden','Finnland','Dänemark')
```

19. Erstellen Sie eine Liste, welche Kunden 1995 länger als 10 Tage auf ihre Lieferung mit SpeedExpress warten mussten.

```
SELECT DISTINCT kunden.firma
FROM kunden, bestellungen, versandfirmen
WHERE kunden.kundencode = bestellungen.kundencode
AND bestellungen.versandueber = versandfirmen.firmennr
AND lieferdatum-bestelldatum > 10
AND YEAR(Lieferdatum) = 1996
AND Versandfirmen.firma LIKE 'Speed%'
```

20. Erstellen Sie eine Liste mit dem Umsatz je Kategorie, die mit dem Minimum beginnt

```
SELECT Kategorienname, SUM((bestelldetails.einzelpreis * an-  
zahl) - (bestelldetails.einzelpreis * Anzahl * Rabatt)) AS  
Umsatz  
FROM bestellungen,bestelldetails, artikel, kategorien  
WHERE bestellungen.bestellnr = bestelldetails.bestellnr  
AND bestelldetails.artikelnr = artikel.artikelnr  
AND artikel.kategorienr = kategorien.kategorienr  
GROUP BY kategorienname  
ORDER BY 2
```

### 13.3 Fragen zum Thema JOINS

1. Gesucht werden alle Rechnungen, die mit Kreditkarte beglichen wurden.

```
SELECT
    RechnungsNr,
    KundenNr,
    Betrag,
    Rechnungen.Kartennummer,
    Firma,
    Inhaber,
    Ablaufdatum
FROM Kreditkarte, Rechnungen
WHERE Kreditkarte.Kartennummer = Rechnungen.Kartennummer

SELECT
    RechnungsNr,
    KundenNr,
    Betrag,
    Rechnungen.Kartennummer,
    Firma,
    Inhaber,
    Ablaufdatum
FROM Kreditkarte
INNER JOIN Rechnungen ON Kreditkarte.Kartennummer = Rechnungen.Kartennummer
```

2. Gesucht werden alle Rechnungen. Falls sie per Kreditkarte bezahlt wurden, so sollen die Kartendaten ebenfalls ausgegeben werden.

```
SELECT
    RechnungsNr,
    KundenNr,
    Betrag,
    Rechnungen.Kartennummer,
    Firma,
    Inhaber,
    Ablaufdatum
FROM Rechnungen
LEFT JOIN Kreditkarte ON Kreditkarte.Kartennummer = Rechnungen.Kartennummer
```

3. Gesucht werden alle Karteninformationen. Falls mit der entsprechenden Kreditkarte etwas bestellt wurde, sollen die Rechnungsinformationen beigefügt werden.

```
SELECT
    RechnungsNr,
    KundenNr,
    Betrag,
    Kreditkarte.Kartennummer,
    Firma,
    Inhaber,
    Ablaufdatum
FROM Rechnungen RIGHT JOIN Kreditkarte
ON Kreditkarte.Kartennummer = Rechnungen.Kartennummer
```

4. Gesucht werden sowohl alle Karteninformationen als auch alle Rechnungen. Sofern möglich sollen dabei Rechnungen und Karten kombiniert werden.

```
SELECT
    RechnungsNr, KundenNr,
    Betrag, Rechnungen.Kartennummer,
    Firma, Inhaber,
    Ablaufdatum
FROM Rechnungen
OUTER JOIN Kreditkarte ON Kreditkarte.Kartennummer = Rechnungen.Kartennummer
```

5. Alle Rechnungen zur Kreditkarte 12347

```
-- Ineffizient, da where erst nach dem join kommt
SELECT
    RechnungsNr,
    KundenNr,
    Betrag,
    Rechnungen.Kartennummer,
    Firma,
    Inhaber,
    Ablaufdatum
FROM Rechnungen
INNER JOIN Kreditkarte ON Kreditkarte.Kartennummer = Rechnungen.Kartennummer
WHERE Kreditkarte.Kartennummer = 12347
```

```
SELECT
```

```

RechnungsNr, KundenNr,
Betrag, Rechnungen.Kartennummer,
Firma, Inhaber, Ablaufdatum
FROM Kreditkarte
INNER JOIN Rechnungen ON
    Kreditkarte.Kartennummer = Rechnungen.Kartennummer
    AND Kreditkarte.Kartennummer = 12347

SELECT
    RechnungsNr, KundenNr,
    Betrag, Rechnungen.Kartennummer,
    Firma, Inhaber, Ablaufdatum
FROM
    (SELECT Kartennummer, Firma, Inhaber, Ablaufdatum
     FROM Kreditkarte
     WHERE Kartennummer = 12347) Karte
INNER JOIN Rechnungen
    ON Karte.Kartennummer = Rechnungen.Kartennummer

--Dies ist die Subselect-Variante, die nicht von allen DBMS
--unterstützt wird (vgl. z.B. ältere Versionen von MySQL oder MS-
--Access).
--Der Vorteil dieser Schreibweise gegenüber der vorherigen liegt
--in der Parametrisierbarkeit: Man stelle sich vor, man möchte
--eine Abfrage innerhalb einer Anwendung mehrfach verwenden, der
--Kriterienausdruck soll allerdings dynamisch bleiben. In PHP-
--Syntax könnte sodann der gesamte Teil SELECT ... FROM ... WHERE
--stehen bleiben, einzige und alleine hinter dem WHERE wird eine
--Variable $(kriterienausdruck) platziert, die beliebig komplex
--sein kann. Da es sich um ein gewöhnliches SELECT FROM WHERE
--handelt, können hier auch Gruppierungen o.ä. vorgenommen werden,
--die in der ON-Klausel unzulässig sind.

```

6. Sie möchten z.B. für eine spezielle Marketingaktion wissen, welche Kunden sowohl über eine Mastercard als auch eine Kreditkarte von American Express verfügen. Ein Selfjoin kann Ihnen helfen, die gewünschten Daten zu erhalten.

```

-- self join

SELECT
    KK1.KndNr, KK1.Firma, KK2.KndNr, KK2.Firma
FROM Kreditkarten KK1
INNER JOIN Kreditkarten KK2 ON KK1.KndNr = KK2.KndNr
WHERE KK1.Firma = 'Mastercard' AND KK2.Firma = 'American
Express'

```

7. Im vorhergehenden Abschnitt haben Sie für Ihre Marketingabteilung die Kundennummern der Kunden herausgefunden, die sowohl über eine 'Mastercard' als auch eine 'American Express' verfügen

```
SELECT
    KK1.KndNr, Nachname, Vorname,
    Strasse, PLZ, Ort
FROM (
    Kreditkarten KK1
INNER JOIN Kreditkarten KK2
    ON KK1.KndNr = KK2.KndNr
)
INNER JOIN Kunden
    ON KK1.KndNr = Kunden.KndNr
WHERE KK1.Firma = 'Mastercard' AND KK2.Firma = 'American
Express'
```

8. Diesmal benötigt sie die Adressen von allen Kunden, von den Kreditkarteninhabern zusätzlich die Kreditkarteninformationen aber nur, wenn sie Mitglied im Vorteilsclub sind. Ihnen ist klar, dass Sie für die Lösung dieser Aufgabe die Tabellen 'Kunden', 'Kreditkarten' und 'Vorteilsclub' miteinander verknüpfen müssen:

```
SELECT
    Kunden.KndNr, Nachname, Vorname,
    Strasse, PLZ, Ort,
    Firma, KartenNr, Ablaufdatum
FROM
    Kunden
LEFT JOIN (
    Kreditkarten
INNER JOIN
    Vorteilsclub
ON
        Kreditkarten.KndNr = Vorteilsclub.KndNr
)
ON
    Kunden.KndNr = Kreditkarten.KndNr
```

9. Ihr Chef will eine Übersicht über die im Monat Oktober bestellten Artikel haben, mit den Detailinformationen zu den Kunden, die diese Bestellungen getätigt haben. Die benötigten Daten verteilen sich auf die Tabellen ‘Kunden’, ‘Bestellungen\_Oktober’ und ‘Positionen’.

```
SELECT
    Artikel, Anzahl, Preis, Datum,
    Nachname, Vorname, Strasse,
    PLZ, Ort
FROM
    Positionen
INNER JOIN (
    Bestellungen_Oktober
INNER JOIN
    Kunden
ON
    Bestellungen_Oktober.KndNr = Kunden.KndNr
)
ON
    Positionen.BestellungsNr      =      Bestellungen_Oktober.BestellungsNr
```

10. Nun möchte Ihr Chef eine Übersicht über alle Kunden mit den Artikeln, die diese bestellt haben. Auch Kunden ohne Bestellungen sollen mit Vor- und Nachname aufgeführt werden; diese Spalten sollen vorne stehen.

```
SELECT
    Nachname, Vorname,
    Artikel, Anzahl, Preis, Datum
FROM
    Kunden
LEFT JOIN (
    Bestellungen_Oktober
INNER JOIN
    Positionen
ON Positionen.BestellungsNr = Bestellungen_Oktober.BestellungsNr
)
ON
    Kunden.KndNr = Bestellungen_Oktober.KndNr
```

11. Sie haben eine Tabelle mit Teams und möchten diese in einer Einfachrunde gegeneinander antreten lassen (Das Heimrecht sei irrelevant). Ein Thetajoin liefert Ihnen alle Begegnungen

```
SELECT
    h.Team AS Heim,
    g.Team AS Gast
FROM
    Teams h
INNER JOIN
    Teams g
ON
    h.T_ID > g.T_ID
```

12. Für eine Doppelrunde mit Hin- und Rückspiel nehmen Sie einfach

```
SELECT
    h.Team AS Heim,
    g.Team AS Gast
FROM
    Teams h
INNER JOIN
    Teams g
ON
    h.T_ID <> g.T_ID
```

### 13.3.1 SQL-DDL

```

# CREATE DATABASE IF NOT EXISTS `join2`;
# USE `selfhtml2`;

DROP TABLE IF EXISTS `Bestellungen_Oktober`;
CREATE TABLE `Bestellungen_Oktober` (
  `KndNr` INT(11) NOT NULL,
  `BestellungsNr` INT(11) NOT NULL AUTO_INCREMENT,
  `Datum` DATE NOT NULL,
  PRIMARY KEY (`BestellungsNr`),
  KEY `KndNr` (`KndNr`),
  KEY `Datum` (`Datum`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Bestellungen_Oktober` WRITE;
INSERT INTO `Bestellungen_Oktober` VALUES
(123456, 987654, '2014-10-15'),
(123456, 987655, '2014-10-16'),
(123457, 987656, '2014-10-16');
UNLOCK TABLES;

DROP TABLE IF EXISTS `Kreditkarten`;
CREATE TABLE `Kreditkarten` (
  `KartenNr` BIGINT(20) NOT NULL AUTO_INCREMENT,
  `Firma` tinytext COLLATE utf8_bin NOT NULL,
  `KndNr` BIGINT(20) NOT NULL,
  `Ablaufdatum` DATE NOT NULL,
  PRIMARY KEY (`KartenNr`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Kreditkarten` WRITE;
INSERT INTO `Kreditkarten` VALUES
(12345, 'VISA', 123457, '2019-05-01'),
(12346, 'Mastercard', 123459, '2020-01-01'),
(12348, 'American Express', 123459, '2019-05-01'),
(12349, 'Diners Club', 123458, '2022-02-01'),
(12350, 'VISA', 123458, '2017-03-01');
UNLOCK TABLES;

DROP TABLE IF EXISTS `Kunden`;
CREATE TABLE `Kunden` (
  `KndNr` BIGINT(20) NOT NULL,
  `Nachname` tinytext COLLATE utf8_bin NOT NULL,
  `Vorname` tinytext COLLATE utf8_bin NOT NULL,
  `Strasse` tinytext COLLATE utf8_bin NOT NULL,
  `PLZ` text COLLATE utf8_bin NOT NULL,
  `Ort` tinytext COLLATE utf8_bin NOT NULL,
  PRIMARY KEY (`KndNr`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;

```

```

LOCK TABLES `Kunden` WRITE;
INSERT INTO `Kunden` VALUES
(123456, 'Mustermann', 'Max', 'Musterweg 1', '12345', 'Musterstadt'),
(123457, 'Musterfrau', 'Katrin', 'Musterstraße 7', '12345', 'Musterstadt'),
(123458, 'Müller', 'Lieschen', 'Beispielweg 3', '23987', 'Irgendwo'),
(123459, 'Schmidt', 'Hans', 'Hauptstraße 2', '98765', 'Anderswo'),
(123460, 'Becker', 'Heinz', 'Mustergasse 4', '12543', 'Musterdorf');
UNLOCK TABLES;

DROP TABLE IF EXISTS `Positionen`;
CREATE TABLE `Positionen` (
  `PositionsNr` BIGINT(20) NOT NULL AUTO_INCREMENT,
  `BestellungsNr` BIGINT(20) NOT NULL,
  `Artikel` tinytext COLLATE utf8_bin NOT NULL,
  `Anzahl` INT(11) NOT NULL,
  `Preis` DECIMAL(10,2) NOT NULL,
  PRIMARY KEY (`PositionsNr`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Positionen` WRITE;
INSERT INTO `Positionen` VALUES
(10241, 987654, 'CD-Player', 2, 49.95),
(10242, 987654, 'DVD-Player', 3, 59.95),
(10243, 987654, 'CD xyz', 10, 15.95),
(10244, 987654, 'DVD abc', 5, 9.95),
(10245, 987655, 'CD-Player', 1, 51.20),
(10246, 987655, 'CD xyz extra ', 20, 16.25),
(10247, 987656, 'DVD-Player', 1, 64.95);
UNLOCK TABLES;

DROP TABLE IF EXISTS `Vorteilsclub`;
CREATE TABLE `Vorteilsclub` (
  `KndNr` BIGINT(20) NOT NULL,
  `ClubNr` BIGINT(20) NOT NULL AUTO_INCREMENT,
  `Kategorie` tinyint(4) NOT NULL,
  PRIMARY KEY (`ClubNr`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Vorteilsclub` WRITE;
INSERT INTO `Vorteilsclub` VALUES
(123458, 1414, 3),
(123456, 1415, 1),
(123460, 1416, 1);
UNLOCK TABLES;

DROP TABLE IF EXISTS `Teams`;

```

```

CREATE TABLE `Teams` (
  `T_ID` INT(3) NOT NULL AUTO_INCREMENT,
  `Team` tinytext COLLATE utf8_bin NOT NULL,
  PRIMARY KEY (`T_ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Teams` WRITE;
INSERT INTO `Teams` VALUES
(1, 'Hamburg'),
(2, 'München'),
(3, 'Berlin'),
(4, 'Köln');
UNLOCK TABLES;
# CREATE DATABASE IF NOT EXISTS `join1`;
# USE `selfhtml`;

DROP TABLE IF EXISTS `Kreditkarte`;
CREATE TABLE `Kreditkarte` (
  `Kartennummer` BIGINT(20) NOT NULL,
  `Firma` tinytext NOT NULL,
  `Inhaber` tinytext NOT NULL,
  `Ablaufdatum` DATE NOT NULL,
  KEY `Kartennummer` (`Kartennummer`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Kreditkarte` WRITE;
INSERT INTO `Kreditkarte` VALUES
(12345, 'VISA', 'Max Mustermann', '2017-05-01'),
(12346, 'Mastercard', 'Katrin Musterfrau', '2018-01-01'),
(12347, 'American Express', 'John Doe', '2015-02-01'),
(12348, 'Diners Club', 'John Doe', '2020-03-01');
UNLOCK TABLES;

DROP TABLE IF EXISTS `Rechnungen`;
CREATE TABLE `Rechnungen` (
  `RechnungsNr` BIGINT(20) NOT NULL,
  `KundenNr` tinytext NOT NULL,
  `Betrag` DECIMAL(10,2) NOT NULL,
  `Kartennummer` BIGINT(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin;
LOCK TABLES `Rechnungen` WRITE;
INSERT INTO `Rechnungen` VALUES
(98765, 'ABX039', 49.95, 12345),
(98766, 'ABX039', 12.95, NULL),
(98767, 'ABX040', 79.95, 12347),
(98768, 'ABX050', 59.99, 12347),
(98769, 'ABX050', 29.99, 12348),
(98770, 'ABX060', 99.99, NULL);
UNLOCK TABLES;

```

## 14 Links

[http://wwwiti.cs.uni-magdeburg.de/iti\\_db/lehre/db1/](http://wwwiti.cs.uni-magdeburg.de/iti_db/lehre/db1/)

## 15 Literatur- und Quellenverzeichnis

- [http://de.wikibooks.org/wiki/Einf%C3%BChrung\\_in\\_SQL](http://de.wikibooks.org/wiki/Einf%C3%BChrung_in_SQL) Für die wesentlichen Elemente der Sprache SQL und der Beispieldatenbank
- Adams

## 16 Index

Aggregatfunktion.....	75	<b>LEFT JOIN</b> .....	94, 103
<b>ALTER</b> .....	48	<b>LIKE</b> .....	62
<b>AND</b> .....	66	<b>MAX</b> .....	76
<b>AVG</b> .....	77	<b>MIN</b> .....	76
<b>BETWEEN</b> .....	62	<b>NOT</b> .....	65
<b>CHECK constraint</b> .....	44	<b>OR</b> .....	66
<b>CONTAINS</b> .....	67	<b>ORDER BY</b> .....	71
<b>COUNT</b> .....	75	<b>OUTER JOIN</b> .....	94, 103
<b>DELETE</b> .....	51	Referentielle Integrität .....	37
<b>DISTINCT</b> .....	60	<b>REPLACE</b> .....	50
<b>ESCAPE</b> .....	63	<b>RIGHT JOIN</b> .....	94
<b>EXISTS</b> .....	65	RIGHT OUTER JOIN.....	105
<b>FULL JOIN</b> .....	94	<b>SELECT</b> .....	57
<b>FULL OUTER JOIN</b> .....	106	<b>SUM</b> .....	76
<b>GROUP BY</b> .....	78	Transaktion .....	132
<b>HAVING</b> .....	82	<b>UNIQUE constraint</b> .....	45
<b>IN</b> 64		Unterabfragen.....	121
<b>INNER JOIN</b> .....	94, 95	<b>UPDATE</b> .....	51
<b>INSERT</b> .....	50	<b>WHERE</b> .....	60
<b>IS NULL</b> .....	63	<b>WHERE – EQUI JOIN</b> .....	87
<b>Join</b> .....	86	<b>XOR</b> .....	66
<b>JOIN</b> .....	73, 94		