

Einführung in SQL

Datenbanken bearbeiten

Jürgen Thomas

Wikibooks

Bibliografische Information

Detaillierte Daten zu dieser Publikation sind bei Wikibooks zu erhalten:

`HTTP://DE.WIKIBOOKS.ORG/`

Namen von Programmen und Produkten sowie sonstige Angaben sind häufig geschützt. Da es auch freie Bezeichnungen gibt, wird das Symbol ® nicht verwendet.

Erstellt am 9. März 2011.

Diese Publikation ist entstanden bei Wikibooks, einem Projekt für Lehr-, Sach- und Fachbücher unter den Lizenzen Creative Commons Attribution/Share-Alike (CC-BY-SA) und GFDL.

PDF- und Druckversion sind entstanden mit dem Programm wb2pdf unter GPL. Dabei wurde das Textsatzprogramm \LaTeX verwendet, das unter der LPPL steht. Einzelheiten und Quellen dazu stehen im Anhang.

Inhaltsverzeichnis

I.	EINFÜHRUNG	3
1.	EIN EINSTIEG	5
1.1.	DATENBANKEN ENTHALTEN INFORMATIONEN	5
1.2.	ABFRAGE NACH DEN MITARBEITERN	6
1.3.	NEUAUFNAHME BEI DEN MITARBEITERN	7
1.4.	SQL UND NATÜRLICHE SPRACHE	8
1.5.	ZUSAMMENFASSUNG	10
1.6.	SIEHE AUCH	10
2.	EINLEITUNG	11
2.1.	EINFÜHRUNG	11
2.2.	GESCHICHTE VON SQL	12
2.3.	ÜBERSICHT ÜBER DATENBANKMANAGEMENTSYSTEME	12
2.4.	REGELN FÜR DIE BEISPIELE	14
2.5.	SIEHE AUCH	16
3.	RELATIONALE DATENBANKEN	17
3.1.	GRUNDSTRUKTUR VON RELATIONALEN DATENBANKEN	17
3.2.	TABELLEN	20
3.3.	SPALTEN	21
3.4.	VERKNÜPFUNGEN UND SCHLÜSSEL	22
3.5.	SIEHE AUCH	25
4.	NORMALISIERUNG	27
4.1.	GRUNDGEDANKEN	27
4.2.	TABELLENKALKULATION ALS AUSGANGSPUNKT	28
4.3.	DIE 1. NORMALFORM	30
4.4.	DIE 2. NORMALFORM	32
4.5.	DIE 3. NORMALFORM	35
4.6.	ZUSÄTZLICHE MASSNAHMEN	37
4.7.	ZUSAMMENFASSUNG	39
4.8.	SIEHE AUCH	40

5.	BEISPIELDATENBANK	43
5.1.	SACHVERHALT	43
5.2.	SCHEMATISCHE DARSTELLUNG	45
5.3.	TABELLENSTRUKTUR UND DATENBANK	46
5.4.	ANMERKUNGEN	47
II.	GRUNDLAGEN	51
6.	SQL-BEFEHLE	53
6.1.	ALLGEMEINE HINWEISE	53
6.2.	DML – DATA MANIPULATION LANGUAGE	55
6.3.	DDL – DATA DEFINITION LANGUAGE	58
6.4.	TCL – TRANSACTION CONTROL LANGUAGE	58
6.5.	DCL – DATA CONTROL LANGUAGE	59
6.6.	ZUSAMMENFASSUNG	59
6.7.	ÜBUNGEN	59
6.8.	LÖSUNGEN	61
6.9.	SIEHE AUCH	62
7.	DML (1) – DATEN ABFRAGEN	63
7.1.	SELECT – ALLGEMEINE HINWEISE	63
7.2.	DIE EINFACHSTE ABFRAGE	64
7.3.	EINGRENZEN DER SPALTEN	65
7.4.	DISTINCT – KEINE DOPPELTEN ZEILEN	65
7.5.	WHERE – EINGRENZEN DER ERGEBNISMENGE	66
7.6.	ORDER BY – SORTIEREN	67
7.7.	FROM – MEHRERE TABELLEN VERKNÜPFEN	69
7.8.	AUSBlick AUf KOMPLEXE ABFRAGEN	71
7.9.	ZUSAMMENFASSUNG	72
7.10.	ÜBUNGEN	72
7.11.	LÖSUNGEN	73
8.	DML (2) – DATEN SPEICHERN	75
8.1.	INSERT – DATEN EINFÜGEN	75
8.2.	UPDATE – DATEN ÄNDERN	78
8.3.	DELETE – DATEN LÖSCHEN	81
8.4.	TRUNCATE – TABELLE LEEREN	82
8.5.	ZUSAMMENFASSUNG	82
8.6.	ÜBUNGEN	83
8.7.	LÖSUNGEN	84

9.	DDL – STRUKTUR DER DATENBANK	87
9.1.	ALLGEMEINE SYNTAX	87
9.2.	HAUPTTEILE DER DATENBANK	88
9.3.	ERGÄNZUNGEN ZU TABELLEN	91
9.4.	PROGRAMMIEREN MIT SQL	93
9.5.	ZUSAMMENFASSUNG	94
9.6.	ÜBUNGEN	95
9.7.	LÖSUNGEN	95
9.8.	SIEHE AUCH	96
10.	TCL – ABLAUFSTEUERUNG	97
10.1.	BEISPIELE	97
10.2.	TRANSAKTIONEN	98
10.3.	TRANSAKTION ERFOLGREICH BEENDEN	99
10.4.	SICHERUNGSPUNKTE	100
10.5.	TRANSAKTION RÜCKGÄNGIG MACHEN	100
10.6.	ZUSAMMENFASSUNG	100
10.7.	ÜBUNGEN	101
10.8.	LÖSUNGEN	102
11.	DCL – ZUGRIFFSRECHTE	105
11.1.	GRANT – ZUGRIFF GEWÄHREN	105
11.2.	REVOKE – ZUGRIFF VERWEIGERN	105
12.	DATENTYPEN	107
12.1.	VORDEFINIERTE DATENTYPEN	107
12.2.	KONSTRUIERTE UND BENUTZERDEFINIERTE DATENTYPEN	112
12.3.	SPEZIALISIERTE DATENTYPEN	113
12.4.	NATIONALE UND INTERNATIONALE ZEICHENSÄTZE	114
12.5.	ZUSAMMENFASSUNG	116
12.6.	ÜBUNGEN	116
12.7.	LÖSUNGEN	117
12.8.	SIEHE AUCH	119
13.	FUNKTIONEN	121
13.1.	ALLGEMEINE HINWEISE	121
13.2.	FUNKTIONEN FÜR ZAHLEN	122
13.3.	FUNKTIONEN FÜR ZEICHENKETTEN	125
13.4.	FUNKTIONEN FÜR DATUMS- UND ZEITWERTE	128
13.5.	FUNKTIONEN FÜR LOGISCHE UND NULL-WERTE	129
13.6.	KONVERTIERUNGEN	130
13.7.	SPALTENFUNKTIONEN	133

13.8.	ZUSAMMENFASSUNG	137
13.9.	ÜBUNGEN	137
13.10.	LÖSUNGEN	139
13.11.	SIEHE AUCH	140
 III. MEHR ZU ABFRAGEN		143
 14. AUSFÜHRLICHE SELECT-STRUKTUR		145
14.1.	ALLGEMEINE SYNTAX	145
14.2.	SET QUANTIFIER – MENGENQUANTIFIZIERER	146
14.3.	SELECT LIST – AUSWAHLLISTE	146
14.4.	TABLE REFERENCE LIST – TABELLEN-VERWEISE	148
14.5.	WHERE CLAUSE – WHERE-KLAUSEL	149
14.6.	GROUP BY CLAUSE – GROUP BY-KLAUSEL	151
14.7.	HAVING CLAUSE – HAVING-KLAUSEL	151
14.8.	UNION CLAUSE – UNION-KLAUSEL	152
14.9.	ORDER BY CLAUSE – ORDER BY-KLAUSEL	152
14.10.	ZUSAMMENFASSUNG	152
14.11.	ÜBUNGEN	153
14.12.	LÖSUNGEN	155
 15. FUNKTIONEN (2)		157
15.1.	ALLGEMEINE HINWEISE	157
15.2.	FUNKTIONEN FÜR ZAHLEN	157
15.3.	FUNKTIONEN FÜR ZEICHENKETTEN	160
15.4.	FUNKTIONEN FÜR DATUMS- UND ZEITWERTE	164
15.5.	FUNKTIONEN FÜR LOGISCHE UND NULL-WERTE	166
15.6.	VERSCHIEDENE FUNKTIONEN	167
15.7.	ZUSAMMENFASSUNG	168
15.8.	ÜBUNGEN	168
15.9.	LÖSUNGEN	170
15.10.	SIEHE AUCH	171
 16. WHERE-KLAUSEL IM DETAIL		173
16.1.	ALLGEMEINE HINWEISE	173
16.2.	EINE EINZELNE BEDINGUNG	173
16.3.	MEHRERE BEDINGUNGEN VERKNÜPFEN	179
16.4.	ZUSAMMENFASSUNG	182
16.5.	ÜBUNGEN	182
16.6.	LÖSUNGEN	183
16.7.	SIEHE AUCH	185

17. MEHRERE TABELLEN	187
17.1. SCHREIBWEISEN BEI MEHREREN TABELLEN	187
17.2. VERKNÜPFUNG ÜBER WHERE – DER TRADITIONELLE WEG . .	188
17.3. JOINS - DER MODERNE WEG	189
17.4. OUTER JOIN - AUCH NULL-WERTE ZURÜCKGEBEN	189
17.5. WEITERE MÖGLICHKEITEN VON JOIN	189
17.6. ZUSAMMENFASSUNG	189
17.7. ÜBUNGEN	190
17.8. LÖSUNGEN	191
18. EINFACHE TABELLENVERKNÜPFUNG	193
18.1. ALLE KOMBINATIONEN ALLER DATENSÄTZE	193
18.2. ZWEI TABELLEN EINFACH VERBINDEN	194
18.3. MEHRERE TABELLEN VERBINDEN	195
18.4. VERKNÜPFUNGS- UND ABFRAGEBEDINGUNGEN	196
18.5. ZUSAMMENFASSUNG	196
18.6. ÜBUNGEN	197
18.7. LÖSUNGEN	198
18.8. SIEHE AUCH	200
19. ARBEITEN MIT JOIN	201
19.1. DIE SYNTAX VON JOIN	201
19.2. INNER JOIN VON ZWEI TABELLEN	202
19.3. WHERE-KLAUSEL BEI JOINS	203
19.4. INNER JOIN MEHRERER TABELLEN	205
19.5. ZUSAMMENFASSUNG	205
19.6. ÜBUNGEN	206
19.7. LÖSUNGEN	207
19.8. SIEHE AUCH	210
20. OUTER JOIN	211
20.1. DIE SYNTAX VON OUTER JOIN	211
20.2. ALLGEMEINE HINWEISE ZU OUTER JOIN	211
20.3. LEFT OUTER JOIN	212
20.4. RIGHT OUTER JOIN	213
20.5. FULL OUTER JOIN	214
20.6. VERKNÜPFUNG MEHRERER TABELLEN	216
20.7. ZUSAMMENFASSUNG	219
20.8. ÜBUNGEN	220
20.9. LÖSUNGEN	222

21. MEHR ZU JOIN	225
21.1. WELCHER JOIN PASST WANN?	225
21.2. SELF JOIN – VERKNÜPFUNG MIT SICH SELBST	226
21.3. CROSS JOIN – DAS KARTESISCHE PRODUKT	232
21.4. WITH – INLINE-VIEW	233
21.5. ZUSAMMENFASSUNG	234
21.6. ÜBUNGEN	234
21.7. LÖSUNGEN	235
21.8. SIEHE AUCH	236
22. NÜTZLICHE ERWEITERUNGEN	237
22.1. DISTINCT – KEINE DOPPELTEN EINTRÄGE	237
22.2. BESCHRÄNKUNG AUF EINE ANZAHL ZEILEN	238
22.3. MEHRERE ABFRAGEN ZUSAMMENFASSEN	247
22.4. CASE WHEN – FALLUNTERSCHIEDUNGEN	250
22.5. ZUSAMMENFASSUNG	256
22.6. ÜBUNGEN	256
22.7. LÖSUNGEN	259
23. BERECHNETE SPALTEN	263
23.1. ALLGEMEINE REGEL	263
23.2. ERGEBNIS VON BERECHNUNGEN	264
23.3. ZEICHENKETTEN VERBINDEN UND BEARBEITEN	264
23.4. ERGEBNIS VON FUNKTIONEN	265
23.5. UNTERABFRAGEN	266
23.6. ZUSAMMENFASSUNG	267
23.7. ÜBUNGEN	267
23.8. LÖSUNGEN	268
24. GRUPPIERUNGEN	271
24.1. SYNTAX VON GROUP BY	271
24.2. GRUPPIERUNG BEI EINER TABELLE	272
24.3. GRUPPIERUNG ÜBER MEHRERE TABELLEN	273
24.4. EINSCHRÄNKUNGEN	274
24.5. ERWEITERUNGEN	276
24.6. ZUSAMMENFASSUNG	278
24.7. ÜBUNGEN	278
24.8. LÖSUNGEN	279
24.9. SIEHE AUCH	281
25. UNTERABFRAGEN	283
25.1. ERGEBNIS ALS EINZELNER WERT	283

25.2.	ERGEBNIS ALS LISTE MEHRERER WERTE	286
25.3.	ERGEBNIS IN FORM EINER TABELLE	289
25.4.	VERWENDUNG BEI BEFEHLEN ZUM SPEICHERN	291
25.5.	ZUSAMMENFASSUNG	298
25.6.	ÜBUNGEN	298
25.7.	LÖSUNGEN	300
26.	ERSTELLEN VON VIEWS	305
26.1.	ALLGEMEINE HINWEISE	305
26.2.	EINE VIEW ANLEGEN UND BENUTZEN	306
26.3.	EINE VIEW ÄNDERN ODER LÖSCHEN	311
26.4.	ZUSAMMENFASSUNG	312
26.5.	ÜBUNGEN	312
26.6.	LÖSUNGEN	314
26.7.	SIEHE AUCH	318
IV.	ERWEITERUNGEN	319
27.	DDL – EINZELHEITEN	321
27.1.	DEFINITION EINER TABELLE	321
27.2.	DEFINITION EINER EINZELNEN SPALTE	323
27.3.	TABELLE ÄNDERN	328
27.4.	CONSTRAINTS – EINSCHRÄNKUNGEN	331
27.5.	ZUSAMMENFASSUNG	339
27.6.	ÜBUNGEN	339
27.7.	LÖSUNGEN	341
27.8.	SIEHE AUCH	342
28.	FREMDSCHLÜSSEL-BEZIEHUNGEN	343
28.1.	PROBLEMSTELLUNG	343
28.2.	GRUNDSÄTZE DER LÖSUNG	344
28.3.	SYNTAX UND OPTIONEN	345
28.4.	BEISPIELE	349
28.5.	KOMBINATION VON FREMDSCHLÜSSELN	351
28.6.	REKURSIVE FREMDSCHLÜSSEL	352
28.7.	REIHENFOLGE DER MASSNAHMEN BEACHTEN	354
28.8.	ZUSAMMENFASSUNG	356
28.9.	ÜBUNGEN	357
28.10.	LÖSUNGEN	359
28.11.	SIEHE AUCH	361

29. SQL-PROGRAMMIERUNG	363
29.1. ALLGEMEINE HINWEISE	363
29.2. ROUTINEN OHNE FESTE SPEICHERUNG	364
29.3. PROGRAMMIEREN INNERHALB VON ROUTINEN	365
29.4. SQL-PROGRAMMIERUNG MIT FIREBIRD	367
29.5. SQL-PROGRAMMIERUNG MIT MS-SQL	371
29.6. SQL-PROGRAMMIERUNG MIT MYSQL	375
29.7. SQL-PROGRAMMIERUNG MIT ORACLE	381
29.8. ZUSAMMENFASSUNG	385
29.9. ÜBUNGEN	386
29.10. LÖSUNGEN	387
29.11. SIEHE AUCH	388
30. EIGENE FUNKTIONEN	389
30.1. FUNKTION DEFINIEREN	389
30.2. BEISPIELE	391
30.3. ZUSAMMENFASSUNG	393
30.4. ÜBUNGEN	394
30.5. LÖSUNGEN	396
30.6. SIEHE AUCH	398
31. PROZEDUREN	399
31.1. EIN ÜBERBLICK	399
31.2. PROZEDUR DEFINIEREN	400
31.3. BEISPIELE	402
31.4. ZUSAMMENFASSUNG	412
31.5. ÜBUNGEN	413
31.6. LÖSUNGEN	415
31.7. SIEHE AUCH	418
32. TRIGGER	421
32.1. EIN ÜBERBLICK	421
32.2. TRIGGER DEFINIEREN	422
32.3. BEISPIELE	424
32.4. ZUSAMMENFASSUNG	426
32.5. ÜBUNGEN	427
32.6. LÖSUNGEN	429
32.7. SIEHE AUCH	431
33. TIPPS UND TRICKS	433
33.1. DIE LETZTE ID ABFRAGEN	433
33.2. TABELLENSTRUKTUR AUSLESEN	436

33.3.	SIEHE AUCH	441
34.	ÄNDERUNG DER DATENBANKSTRUKTUR	443
34.1.	SPALTEN HINZUFÜGEN UND ÄNDERN	443
34.2.	EINSCHRÄNKUNGEN AUF SPALTEN	445
34.3.	INDIZES	447
34.4.	FREMSCHLÜSSEL	448
34.5.	WEITERE ANPASSUNGEN	448
34.6.	ZUSAMMENFASSUNG	452
34.7.	SIEHE AUCH	452
35.	TESTDATEN ERZEUGEN	455
35.1.	ALLGEMEINE HINWEISE	455
35.2.	NEUE FAHRZEUGE REGISTRIEREN	456
35.3.	NEUE VERSICHERUNGSVERTRÄGE REGISTRIEREN	456
35.4.	PROBLEME MIT TESTDATEN	463
35.5.	ZUSAMMENFASSUNG	464
35.6.	SIEHE AUCH	465
V.	ANHANG	467
36.	TABELLENSTRUKTUR DER BEISPIELDATENBANK	469
37.	DOWNLOADS	475
37.1.	DIE DOWNLOAD-SEITE	475
37.2.	VERBINDUNG ZU DEN DATENBANKSYSTEMEN	476
37.3.	DIE VOLLSTÄNDIGE BEISPIELDATENBANK	479
37.4.	ERSTELLEN DER BEISPIELDATENBANK	480
37.5.	SKRIPTE FÜR NACHTRÄGLICHE ÄNDERUNGEN	481
38.	BEFEHLSREFERENZ	483
38.1.	DDL (DATA DEFINITION LANGUAGE)	483
38.2.	DML – DATA MANIPULATION LANGUAGE	488
38.3.	TCL – TRANSACTION CONTROL LANGUAGE	492
38.4.	DCL – DATA CONTROL LANGUAGE	493
39.	SCHLÜSSELWÖRTER	495
40.	LITERATURVERZEICHNIS	501
41.	WEBLINKS	503
42.	AUTOREN	507

Dieses Buch richtet sich an:

Schüler, Studenten und Andere, die sich mit relationalen Datenbanken beschäftigen wollen bzw. müssen

Was dieses Buch erreichen will:

- Einführung in SQL anhand einer Beispieldatenbank.
- Der Leser soll die Beispiele anhand von Übungen auf seinem eigenen Datenbankmanagementsystem nachvollziehen können.

Die Schwerpunkte liegen hierbei auf folgenden Themen:

- Abfragen von Daten
- Manipulieren von Daten
- Einfaches Ändern der Datenbankstruktur

Was dieses Buch nicht sein soll:

- Keine Einführung in die Grundkonzepte relationaler Datenbankmodelle.
- Keine Einführung in die Modellierung von relationalen Datenbanken.
- Keine Einführung in die Administration von Datenbankmanagementsystemen.
- Keine Einführung in das Performance-Tuning von relationalen Datenbanken.
- Keine Einführung in prozedurale (z.B. PL/SQL) oder objektorientierte Sprachen, die innerhalb der Datenbank gespeichert und genutzt werden können.

Teil I.

Einführung

1. Ein Einstieg

Nehmen wir einmal an, dass ein Versicherungsunternehmen einen neuen Geschäftsführer bekommt. Dieser sammelt zunächst Informationen, um sich mit dem Unternehmen vertraut zu machen.

1.1. Datenbanken enthalten Informationen

Der Geschäftsführer findet in seinem Büro keine Akten, sondern nur einen PC. Nach dem Einschalten und Anmelden bekommt er folgende Begrüßung:

```
Wikibooks-Beispieldatenbank
(C) Wiki-SQL
Bitte geben Sie einen SQL-Befehl ein:
sql >
```

Damit wird der Geschäftsführer auf mehrere Punkte der Organisation in seinem Unternehmen hingewiesen.

- Die Daten sind in einer Datenbank zusammengefasst mit dem Namen *Beispieldatenbank*.
- Diese Datenbank (DB) stammt von der Firma Wikibooks.
- Um etwas mit dieser DB zu machen, soll er hinter "sql >" einen SQL-Befehl eingeben.

Die Inhalte dieser Datenbank sind in Beispieldatenbank beschrieben, Einzelheiten in Tabellenstruktur der Beispieldatenbank erläutert.

Das Datenbanksystem (DBMS) der Firma Wikibooks wird als Wiki-SQL bezeichnet. Dies bedeutet, dass die Beispieldatenbank zu Wiki-SQL passen muss – in Bezug auf das Dateiformat, die interne Struktur und den Aufbau der Befehle.

SQL ist die Abkürzung für **Structured Query Language**, also strukturierte Abfragesprache. Der Geschäftsführer hat schon einmal davon gehört, dass dies eine standardisierte Form ist, um mit Datenbanken zu arbeiten, und probiert es aus.

1.2. Abfrage nach den Mitarbeitern

Als erstes möchte er eine Liste seiner Mitarbeiter haben. Er überlegt, wie die Abfrage lauten könnte:

```
Hole Name und Vorname von den Mitarbeitern
```

Auf Englisch probiert er (mit ein paar Synonymen) also so etwas aus:

```
Get Name, Vorname From Mitarbeiter  
Fetch Name, Vorname From Mitarbeiter  
Find Name, Vorname From Mitarbeiter  
Search Name, Vorname From Mitarbeiter
```

Und schließlich bekommt er keine Fehlermeldung, sondern ein Ergebnis:

```
Select Name, Vorname From Mitarbeiter
```

Die Liste ist ihm zu lang und unübersichtlich. Er will sie zunächst einmal sortieren und probiert *Sortierung, Reihenfolge* aus, bis es passt:

```
Select Name, Vorname From Mitarbeiter order by Name
```

Dann möchte er die Auswahl einschränken, nämlich auf die Mitarbeiter mit Anfangsbuchstaben 'A'. Wieder nach ein paar Versuchen weiß er, dass nicht WITH, sondern WHERE die Lösung liefert.

```
Select Name, Vorname From Mitarbeiter where Name < 'B'
```

Jetzt möchte er beide Abfragen verbinden:

```
Quelltext
```

```
Falsch
```

```
Select Name, Vorname From Mitarbeiter order by Name where Name < 'B'
```

```
SQL error code = -104. Token unknown - line 1, column 53. WHERE.
```

Das kann doch nicht sein?! WHERE ist doch das richtige Verfahren für eine solche Einschränkung?! Kommt es etwa auf die Reihenfolge der Zusätze an?

```
Select Name, Vorname From Mitarbeiter where Name < 'B' order by Name
```

NAME	VORNAME
Aagenau	Karolin
Aliman	Zafer

Welche Informationen sind denn sonst gespeichert? Er weiß auch (z. B. vom DIR-Befehl des Betriebssystems), dass ein Sternchen anstelle von *alles* gesetzt werden kann. Und siehe da, es klappt:

```
Select * From Mitarbeiter where Name < 'B' order by Name
```

ID	PERSONALNUMMER	NAME	VORNAME	GEBURTSDATUM	TELEFON
(und noch mehr)					
13	60001	Aagenau	Karolin	02.01.1950	0234/66006001
usw.					
18	80002	Aliman	Zafer	12.11.1965	0201/4012161
usw.					

Prima, damit ist klar, wie Informationen aus der Datenbank geholt werden:

```
SELECT <Liste von Teilinformationen>  
  FROM <Teil der Datenbank>  
 WHERE <Bedingung>  
 ORDER BY <Sortierung>
```

1.3. Neuaufnahme bei den Mitarbeitern

Als nächstes möchte der Geschäftsführer sich selbst als Mitarbeiter speichern. Schnell kommt er auf das „Grundgerüst“ des Befehls:

```
INSERT INTO Mitarbeiter VALUES
```

Wenn er danach seinen Namen schreibt, bekommt er wieder eine Fehlermeldung mit "token unknown". Er hat aber schon von der Benutzung von Klammern in der EDV gehört.

Quelltext

Falsch

```
Insert into Mitarbeiter VALUES ('Webern', 'Anton')
```

```
SQL error code = -804.  
Count of read-write columns does not equal count of values.
```

Na gut, dann wird eben ausdrücklich angegeben, dass erstmal nur Name und Vorname einzutragen sind.

Quelltext

Falsch

```
Insert into Mitarbeiter (Name, Vorname) VALUES ('Webern', 'Anton')
```

```
validation error for column PERSONALNUMMER, value "*** null ***".
```

Ach so, die Personalnummer muss angegeben werden, und vermutlich alles andere auch. Aber die ID ist doch gar nicht bekannt? Nun, immerhin sind wir auf diese Grundstruktur des Befehls gekommen:

```
INSERT INTO <Teil der Datenbank>  
    [ ( <Liste von Teilinformationen> ) ]  
VALUES ( <Liste von Werten> )
```

1.4. SQL und natürliche Sprache

Offensichtlich sind die Befehle von SQL der natürlichen englischen Sprache nachempfunden. (Die englische Sprache hat wegen der klaren Satzstruktur und Grammatik insofern natürlich Vorteile gegenüber der komplizierten deutschen Syntax.)

1.4.1. SELECT für Abfragen

Um Daten abzufragen, steht der **SELECT**-Befehl zur Verfügung mit folgenden Details:

SELECT	wähle aus
[DISTINCT ALL]	verschiedene alle
<Liste von Teilinformationen>	
FROM <Teil der Datenbank>	aus
[WHERE <Bedingungen>]	wobei
[GROUP BY <Sortierung>]	gruppiert durch
[HAVING <Bedingungen>]	wobei
[ORDER BY <Liste von Spalten>]	sortiert durch

1.4.2. INSERT für Neuaufnahmen

Um Daten neu zu speichern, gibt es den **INSERT**-Befehl mit folgenden Details:

INSERT INTO <Teil der Datenbank>	einfügen in
[<Liste von Teilinformationen>]	
VALUES (<Liste von Werten>)	Werte
/* oder */	
INSERT INTO <Teil der Datenbank>	einfügen in
[<Liste von Teilinformationen>]	
SELECT <Ergebnis einer Abfrage>	durch eine Auswahl

1.4.3. UPDATE für Änderungen

Um Daten zu ändern, gibt es den **UPDATE**-Befehl mit folgenden Details:

UPDATE <Teil der Datenbank>	aktualisiere
SET <spalte1> = <wert1> [,	setze fest
<spalte2> = <wert2> , usw.	
<spalten> = <wertn>]	
[WHERE <bedingungsliste>];	wobei

1.4.4. DELETE für Löschungen

Um Daten zu löschen, gibt es den **DELETE**-Befehl mit folgenden Details:

DELETE FROM <Teil der Datenbank>	lösche aus
[WHERE <bedingungsliste>];	wobei

1.4.5. CREATE TABLE bei der Struktur einer Tabelle

Um eine neue Tabelle zu erstellen, gibt es den **CREATE TABLE**-Befehl mit folgenden Einzelheiten:

```
CREATE TABLE <Teil der Datenbank>           erzeuge Tabelle
( <Spaltenliste>,
  <weitere Angaben> )
```

So einfach kann es gehen? Dann kann man doch auch eigene Daten erzeugen, speichern, abfragen und auswerten.

1.5. Zusammenfassung

Die einzelnen Teile der SQL-Befehle sind leicht verständlich; und es scheint nur wenige Befehle zu geben, die man als „Anfänger“ wirklich lernen muss. Natürlich kann man nicht sofort alle Möglichkeiten erfassen. Aber angesichts des begrenzten Umfangs und der klaren Struktur lohnt es sich, sich näher damit zu befassen. Dies will dieses Buch erleichtern.

1.6. Siehe auch

Wikipedia hat einen Artikel zum Thema SQL¹.

Weitere Informationen gibt es in folgenden Kapiteln:

- BEISPIELDATENBANK²
- TABELLENSTRUKTUR DER BEISPIELDATENBANK³

1 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SQL](http://de.wikipedia.org/wiki/SQL)

2 Kapitel 5 auf Seite 43

3 Kapitel 36 auf Seite 469

2. Einleitung

In diesem Kapitel erhalten Sie Informationen über Inhalt und Aufbau des Buches.

2.1. Einführung

Die Abfragesprache SQL ist die etablierte Sprache für die Arbeit mit relationalen Datenbankmanagementsystemen (DBMS). Es existieren verschiedene Standards, und jeder Hersteller von DBMS hat seine eigenen Erweiterungen und Besonderheiten zu den Standards.

Das Buch soll eine Einführung in die Sprache SQL bieten. Ziel ist es, dass der Leser nach dem Durcharbeiten folgende Aufgaben selbständig lösen kann:

- Eigene Abfragen für relationale Datenbanken erstellen
- Manipulieren von Daten (Einfügen, Ändern und Löschen)
- Eigene einfache relationale Datenbanken aufbauen
- Bestehende Datenbankstrukturen erweitern

Um die Ziele zu erreichen, wird SQL anhand praxisnaher Beispiele erläutert.

Die Beispiele im Buch wurden unter mindestens einem der folgenden DBMS getestet:

- Firebird
- MS-SQL Server 2005 oder 2008
- MySQL 4.1 oder 5
- Oracle 9, 10 oder 11

Vorzugsweise werden allgemeingültige Schreibweisen nach dem SQL-Standard (siehe unten) benutzt. Deshalb sollten die Befehle in aller Regel auf allen gängigen DBMS funktionieren und höchstens kleinere Änderungen benötigen. Dort, wo eine spezielle Schreibweise wesentlich abweicht, wird das ausdrücklich erwähnt.

i Achtung:

Wegen der vielen unterschiedlichen Schreibweisen der SQL-Befehle ist eine vollständige Prüfung leider nicht möglich. Sie müssen in allen Zweifelsfällen in der Dokumentation Ihres DBMS die passende Schreibweise nachlesen.

2.2. Geschichte von SQL

SQL ist aus IBM's SEQUEL in den siebziger Jahren entstanden. Der Erfolg der Sprache SQL ist sicherlich auch darin begründet, dass die Sprache einfach aufgebaut ist und sich an der englischen Umgangssprache orientiert. Von SQL gibt es verschiedene Standards. Diese sind:

- Erster SQL-Standard (1986 ANSI)
- SQL2 bzw. SQL-92 (1992)
- SQL3 bzw. SQL:1999 (1999 ISO)
- SQL:2003 (2003)
- SQL:2008 (2008)

Hierbei ist zu beachten, dass die meisten Datenbankmanagementsysteme SQL2 unterstützen. Die neueren Versionen sind in der Regel nur teilweise oder gar nicht in den einzelnen Datenbankmanagementsystemen umgesetzt.

Alles, was in diesem Buch als **SQL-Standard**, also als „offizielle SQL-Feststellung“ angegeben wird, bezieht sich auf die **SQL-Dokumente von 2003**

2.3. Übersicht über Datenbankmanagementsysteme

2.3.1. Allgemein

Es gibt verschiedene Konzepte von Datenbanken:

- Relationale DBMS
- Objektorientierte DBMS
- Objektorientierte DBMS

Bei Wikipedia gibt es eine Liste der Datenbankmanagementsysteme.

Da SQL die Abfragesprache für relationale Datenbanken ist, bezieht sich das Buch nur auf diese Art von Datenbanken. Das Konzept hinter den relationalen Datenbanken wird im nächsten Kapitel erläutert.

2.3.2. Kommerzielle Datenbankmanagementsysteme

- DB2
- Informix
- Interbase
- Microsoft SQL Server
- Oracle
- Sybase

Microsoft und Oracle bieten auch kostenlose Express-Versionen mit eingeschränkten Möglichkeiten oder Nutzungsrechten an.

2.3.3. Freie Datenbankmanagementsysteme

- Firebird
- MySQL
- PostgreSQL
- SQLite

Bei MySQL ist das duale Lizenzsystem zu beachten: je nach Nutzungsbedingungen frei oder kostenpflichtig.

Die Unterscheidung zwischen „frei“ und „kommerziell“ ist nicht korrekt, sondern wird nur der Einfachheit halber gemacht. Bei den „freien“ DBMS steht die freie Verfügbarkeit im Vordergrund, auch wenn Kosten anfallen können oder das DBMS nicht als *Open Source*-Projekt entwickelt wird. Bei den „kommerziellen“ DBMS steht das gewerbliche Interesse des Anbieters im Vordergrund, auch wenn es kostenlose Lizenzen gibt.

2.3.4. Weitere Systeme zur Datenverwaltung

Die folgenden Dateisysteme enthalten keine Datenbanken im eigentlichen Sinne, sondern Dateien für strukturierte Daten. Auch diese können (je nach ver-

wendetem Programm) in eingeschränktem Umfang mit SQL-Befehlen umgehen.

- dBASE und seine Varianten
- MS-Access
- das Datenbankmodul *base* von OpenOffice
- Paradox

Auf diese Systeme gehen wir nicht ein. Sie müssen in der jeweiligen Programm-Dokumentation nachlesen, welche Befehle und Optionen möglich sind.

2.4. Regeln für die Beispiele

Das Buch ist grundsätzlich schrittweise aufgebaut. Aber nicht immer können in einem Beispiel nur bereits bekannte Begriffe verwendet werden. Wenn in seltenen Fällen Bestandteile erst in einem späteren Kapitel erläutert werden, dann gibt es ausdrückliche Hinweise, beispielsweise hier:

- Der INSERT-Befehl in DML (2) – DATEN SPEICHERN¹ muss korrekt mit DATENTYPEN² umgehen und benutzt dazu auch FUNKTIONEN³.
- Auch für das Erstellen einer Tabelle in DDL – STRUKTUR DER DATENBANK⁴ muss genau auf die Datentypen geachtet werden.

Wenn Sie die SQL-Begriffe aus dem Englischen ins Deutsche übersetzen, sollten Sie den Zusammenhang auch ohne Hin- und Herblättern verstehen.

Die Beispiele für SQL-Befehle werden nach den folgenden Regeln geschrieben.

1. Alle SQL-Befehle und Schlüsselwörter, wie z. B. SELECT, INSERT, DELETE, WHERE, ORDER BY, werden vorzugsweise groß geschrieben. SQL selbst verlangt das nicht, sondern arbeitet ohne Berücksichtigung von Groß- und Kleinschreibung (*case insensitive*); dort werden SELECT, select und sogar sElEcT gleich behandelt.⁵
2. Eigentlich sollten Tabellen- und Spaltennamen vorzugsweise ebenfalls groß geschrieben werden, und zwar ohne Anführungszeichen. Aber in der Praxis werden solche Namen meistens „gemischt“ geschrieben.

1 Kapitel 8 auf Seite 75

2 Kapitel 12 auf Seite 107

3 Kapitel 13 auf Seite 121

4 Kapitel 9 auf Seite 87

5 In der PDF-Version verhindern zurzeit technische Einschränkungen, dass alle diese Begriffe automatisch großgeschrieben werden.

3. String-Literale werden mit einfachen Anführungszeichen gekennzeichnet.
4. SQL-Befehle werden mit einem Semikolon abgeschlossen.
5. Optionale Argumente (d. h. solche, die nicht unbedingt erforderlich sind) werden in [] eingeschlossen.
6. Variable Argumente (d. h. solche, die mit unterschiedlichem Inhalt vorkommen) werden in < > eingeschlossen.
7. Wahlmöglichkeiten werden durch das Pipe-Zeichen | (den senkrechten Strich) getrennt.
8. Listen werden gekennzeichnet durch <inhaltliste>, wobei dies eine Kurzform ist für <inhalt1, inhalt2, ... inhaltn>.
9. Sofern das Ergebnis einer Abfrage im Ausgabefenster aufgeführt wird, handelt es sich überwiegend nur um einen Teil des Ergebnisses, gleichgültig ob darauf hingewiesen wird oder nicht.

Die Struktur eines Befehls wird dargestellt durch einen einfachen Rahmen mit Courier-Schrift:

```
SELECT <spaltenliste>
      FROM <tabellenname>
[ WHERE <bedingungsliste> ]
;
```

► **Aufgabe:** So wird eine Aufgabenstellung angezeigt, die mit dem danach folgenden Beispiel erledigt werden soll.

Ein konkretes Beispiel wird mit einem komplexen Rahmen und unterschiedlichen Inhalten (farblicher Hinweis bei Fehlern, mit oder ohne Kopf- und Fußzeile, mit oder ohne Ausgabefenster) dargestellt:

```
select * from Beispieltabelle
where Spalte1 = 'Abc';
```

Hier steht ggf. eine Meldung.

2.5. Siehe auch

Unter [WEBLINKS](#)⁶ stehen viele zusätzliche Hinweise.

[MoWES](#)⁷ bietet eine kostenlose PHP-MySQL-Umgebung für den USB-Stick, die sich gut eignet, MySQL zu lernen und zu testen.

<references />

6 Kapitel 41 auf Seite 503

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/MoWES](http://de.wikipedia.org/wiki/MoWES)

3. Relationale Datenbanken

Um mit SQL auf relationalen Datenbanken arbeiten zu können, muss der Anwender ein Grundverständnis für relationale Datenbanken haben. Dieses soll in diesem Kapitel vermittelt werden.

3.1. Grundstruktur von relationalen Datenbanken

Bevor man mit der Sprache SQL beginnt, muss das Grundprinzip von relationalen Datenbanken geklärt werden. Relationale Datenbanken versuchen einen Bestandteil der Realität in einem Datenmodell abzubilden.

Für diese Datenmodelle gibt es verschiedene Abstraktionsebenen. In der Regel unterscheidet man zwischen **Entitätenmodell** und **Tabellenmodell**. Da es sich hier um eine *Einführung* handelt, beschränken wir uns auf das Tabellenmodell, das weniger Theorie voraussetzt.

Grundsätzlich sollen dabei Objekte der Realität betrachtet werden, welche zueinander in Beziehung stehen. Zum einen werden die Objekte mit ihren Eigenschaften untersucht: Objekte mit gleichen Eigenschaften werden zusammengefasst; Objekte mit verschiedenen Eigenschaften werden getrennt. Zum anderen werden die Beziehungen zwischen unterschiedlichen Objekten behandelt. Außerdem geht es darum, möglichst keine Informationen unnötigerweise doppelt zu speichern.

3.1.1. Beispielhafte Struktur

In unserer `BEISPIELDATENBANK`¹ simulieren wir dazu eine **Versicherungsgesellschaft**. Unter anderem werden die Verträge mit den dazugehörigen Kunden betrachtet:

¹ Kapitel 5 auf Seite 43

- Ein Versicherungsvertrag ist durch die Vertragsnummer, das Datum des Abschlusses, den Versicherungsnehmer, die Art und die Höhe der Police gekennzeichnet.
- Ein Versicherungsnehmer kann bei der Versicherung einen oder mehrere Verträge haben. Es kann Kunden geben, die aktuell keinen Vertrag haben; aber es kann keinen Vertrag ohne zugehörigen Kunden geben.
- Ein Versicherungsnehmer ist gekennzeichnet durch seinen Namen und Anschrift und bei Personen einen Vornamen und ein Geburtsdatum. Außerdem verfügt er „üblicherweise“ über eine Kundennummer, die ihn eindeutig kennzeichnet.

Nun könnte man alle Verträge wie folgt in einer einzigen Datei, z. B. einem Arbeitsblatt einer Tabellenkalkulation, speichern:

NUMMER	ABSCHLUSSDATUM	ART	NAME	ANSCHRIFT	BETREUER	TELEF
DG-01	03.05.1974	TK	Heckel Obsthandel GmbH	46282		
Dorsten			Pohl, Helmut	0201/4014186 Mobil (0171) 4123456		
DG-02	11.06.1975	TK	Heckel Obsthandel GmbH	46282		
Dorsten			Pohl, Helmut	0201/4014186 Mobil (0171) 4123456		
DG-03	25.02.1977	TK	Heckel Obsthandel GmbH	46282		
Dorsten			Pohl, Helmut	0201/4014186 Mobil (0171) 4123456		
XC-01	07.08.1974	HP	Antonius, Bernhard	45892		
Gelsenkirchen			Braun, Christian	0201/4014726 Mobil (0170) 8351647		
RH-01	11.12.1976	VK	Cornelsen, Dorothea	44577		
Castrop-Rauxel			Braun, Christian	0201/4014726 Mobil (0170) 8351647		

Offensichtlich werden die persönlichen Daten eines Versicherungsnehmers und seines Betreuers „zu oft“ gespeichert. Es ist also sinnvoll, dies zu trennen – zum einen die Verträge:

NUMMER	ABSCHLUSSDATUM	ART	KUNDE	BETREUER
DG-01	03.05.1974	TK	1	9
DG-02	11.06.1975	TK	1	9
DG-03	25.02.1977	TK	1	9
XC-01	07.08.1974	HP	2	10
RH-01	11.12.1976	VK	3	10

Zum anderen die Kunden:

NUMMER	NAME	ANSCHRIFT
1	Heckel Obsthandel GmbH	46282 Dorsten
2	Antonius, Bernhard	45892 Gelsenkirchen
3	Cornelsen, Dorothea	44577 Castrop-Rauxel

Und schließlich die zuständigen Sachbearbeiter (Betreuer):

NUMMER	NAME	TELEFON	MOBIL
9	Pohl, Helmut	0201/4014186	(0171) 4123456
10	Braun, Christian	0201/4014726	(0170) 8351647

Durch die Angabe der Nummer (Kunde bzw. Betreuer) in den Aufstellungen ist eine klare Verbindung hergestellt. Außerdem zeigt die Wiederholung des Wortes „Mobil“ an, dass dieser Wert in einer eigenen Spalte eingetragen werden sollte.

Diese Trennung von Informationen sind Schritte bei der NORMALISIERUNG² einer Datenbank.

3.1.2. Eigenschaften der Objekte

Vor allem müssen wir uns Gedanken über die Eigenschaften der verschiedene Objekte machen. Dabei gibt es solche, die ein Objekt eindeutig kennzeichnen, andere, die immer anzugeben sind, und weitere, die nur unter manchen Umständen von Bedeutung sind.

Für einen **Versicherungsnehmer** gibt es u. a. folgende Eigenschaften:

- NUMMER ist eindeutig und eine Pflichtangabe.
- NAME, PLZ, ORT sind Pflichtangaben, ihre Inhalte können aber bei mehreren Versicherungsnehmern vorkommen.
- VORNAME und GEBURTSDATUM sind bei natürlichen Personen Pflicht, aber bei juristischen Personen (Firmen) irrelevant.

Für einen **Versicherungsvertrag** gibt es u. a. folgende Eigenschaften:

- NUMMER ist eindeutig und eine Pflichtangabe.
- Auch die anderen bisher genannten Eigenschaften sind Pflicht, aber sie sind nicht eindeutig.

Die verschiedenen Objekte stehen über die Kundennummer miteinander in Beziehung. Im Beispiel geht es um die Verknüpfung: „Ein Kunde kann einen oder mehrere Verträge oder auch keinen haben.“ *Der letzte Fall „keinen Vertrag“ kommt erst am Schluss des Buches vor, wenn wir weitere TESTDATEN ERZEUGEN*³.

In den relationalen Datenbanksystemen (DBMS) werden die Objekte als Tabellen dargestellt. Die Eigenschaften werden über die Spalten der Tabelle

² Kapitel 4 auf Seite 27

³ Kapitel 35 auf Seite 455

abgebildet. Eine Zeile (wahlweise als Datensatz bezeichnet) in der Tabelle entspricht genau einem Objekt in der Realität. Die Beziehungen zwischen Tabellen werden über Fremdschlüssel abgebildet.

3.2. Tabellen

Tabellen sind zweidimensional gegliederte Informationen. Anzahl, Bezeichnung und Typ der Spalten (auch Felder oder Attribute genannt) werden durch die Definition der Tabelle festgelegt. Die Zeilen (Anzahl und Inhalte) sind variabel und entsprechen jeweils einem wirklichen Objekt des Typs, der in der Tabelle gesammelt wird.

So sieht ein Ausschnitt aus der Tabelle *Abteilung* der Beispieldatenbank aus:

Spaltenname	ID	KURZBEZEICHNUNG	BEZEICHNUNG	ORT
Datentyp	integer	varchar(10)	varchar(30)	varchar(30)
Zeilen	1	Fibu	Finanzbuchhaltung	Dortmund
	2	Albu	Anlagenbuchhaltung	Dortmund
	5	Vert	Vertrieb	Essen
	6	Lagh	Lagerhaltung	Bochum

Diese Tabelle enthält also 4 Spalten und 12 Zeilen, von denen hier 4 angezeigt werden.

Dabei handelt es sich um eine **Basistabelle** (TABLE), die tatsächlich Informationen speichert. Daneben gibt es „virtuelle“ Arten von Tabellen, nämlich die VIEW als Sichttabelle und die Ergebnismenge (Resultset) als Ergebnis einer SELECT-Abfrage.

Eine **View** enthält eine fest vordefinierte Abfrage, die sich auf eine oder mehrere Tabellen bezieht. Aus Sicht des Anwenders sieht sie wie eine Basistabelle aus, ist aber nur eine Abbildung realer Tabellen. Ein Beispiel wäre ein Ausschnitt aus einer View *Mitarbeiter_Bochum*, nämlich der Mitarbeiter, die zu einer der Abteilungen in Bochum gehören:

PERSNR	NAME	VORNAME	BEZEICHNUNG
60001	Aagenau	Karolin	Lagerhaltung
60002	Pinkart	Petra	Lagerhaltung
70001	Olschewski	Pjotr	Produktion
70002	Nordmann	Jörg	Produktion
120001	Carlsen	Zacharias	Forschung und Entwicklung
120002	Baber	Yvonne	Forschung und Entwicklung

Näheres zu Sichttabellen steht im Kapitel ERSTELLEN VON VIEWS⁴.

Jede **Ergebnismenge** hat zwangsläufig die Struktur einer Tabelle.

Ergänzend sei darauf hingewiesen, dass auch das DBMS selbst sämtliche Schemata in Systemtabellen speichert. Beispielsweise stehen bei Interbase und Firebird die Definition von TABLES und VIEWS in der Tabelle RDB\$RELATIONS und die dazugehörigen Felder (Spalten) in RDB\$RELATION_FIELDS.

3.3. Spalten

Spalten bezeichnen die Elemente einer Tabellenstruktur. Sie werden eindeutig gekennzeichnet durch ihren **Namen**; diese Eindeutigkeit gilt innerhalb einer Tabelle, verschiedene Tabellen dürfen Spalten mit gleichem Namen (z. B. *ID*) haben. Außerdem gehört zur Definition einer Spalte der **Datentyp**; dies wird im Kapitel DATENTYPEN⁵ behandelt.

Die Spalten (innerhalb einer Tabelle) werden intern nach **Position** geordnet. Spalten an verschiedenen Positionen können denselben Datentyp haben, aber niemals denselben Namen. Auf eine bestimmte Spalte wird fast immer über den Namen zugegriffen, nur äußerst selten über die Position.

Eine Spalte hat also einen Namen und einen Datentyp. Jede Zeile in einer Tabelle hat genau einen Wert für jede Spalte; wenn mehrere gleichartige Werte eingetragen werden sollen, werden mehrere Spalten benötigt. Jeder Wert in einer Zeile entspricht dem Datentyp der Spalte.

Hinweis: In dieser Hinsicht unterscheiden sich Datenbank-Tabellen ganz wesentlich von denjenigen einer Tabellenkalkulation, bei der der Datentyp einzelner Zellen abweichen kann von der Spaltendefinition und einzelne Zellen zusammengezogen werden können.

Die Eigenschaft **NULL** für einen Wert ist eine Besonderheit, die vor allem Einsteiger gerne verwirrt. Dies bedeutet, dass einer Zelle (noch) kein Wert zugeordnet worden ist. *Eine bessere Bezeichnung wäre etwas wie UNKNOWN; aber es heißt nun leider NULL.* Bitte beachten Sie deshalb:

- Für ein Textfeld werden folgende Werte unterschieden:

4 Kapitel 26 auf Seite 305

5 Kapitel 12 auf Seite 107

1. Der Wert '' ist ein leerer Text.
 2. Der Wert ' ' ist ein Text, der genau ein Leerzeichen enthält.
 3. Der Wert NULL enthält nichts.
- Für ein logisches Feld (Datentyp *boolean*) wird dies unterschieden:
 1. Der Wert TRUE bedeutet „wahr“.
 2. Der Wert FALSE bedeutet „falsch“.
 3. Der Wert NULL bedeutet „unbekannt“.
 - Für ein Zahlenfeld wird dies unterschieden:
 1. Der Wert 0 ist eine bestimmte Zahl, genauso gut wie jede andere.
 2. Der Wert NULL bedeutet „unbekannt“.

Hinweis

Der Wert NULL steht nicht für einen bestimmten Wert, sondern kann immer als „unbekannt“ interpretiert werden.

Dies kann bei jeder Spalte allgemein festgelegt werden: Die Eigenschaft „NOT NULL“ bestimmt, dass in dieser Spalte der NULL-Wert nicht zulässig ist; wenn Daten gespeichert werden, muss immer ein Wert eingetragen werden (und sei es ein leerer Text). Wenn dies nicht festgelegt wurde, muss kein Wert eingetragen werden; der Feldinhalt ist dann NULL.

Bei SELECT-Abfragen (vor allem auch bei Verknüpfungen mehrerer Tabellen) gibt es unterschiedliche Ergebnisse je nachdem, ob NULL-Werte vorhanden sind und ob sie berücksichtigt oder ausgeschlossen werden sollen.

3.4. Verknüpfungen und Schlüssel

Mit diesen Verfahren werden die Tabellen in Beziehung zueinander gebracht. Auch dies folgt der Vorstellung, dass die Wirklichkeit abgebildet werden soll.

3.4.1. Verknüpfungen

Diese, nämlich die Beziehungen zwischen den Tabellen, sind ein Kern eines relationalen Datenbanksystems. In der BEISPIELDATENBANK⁶ bestehen unter anderem folgende Beziehungen:

6 Kapitel 36 auf Seite 469

- Die Tabelle *Mitarbeiter* verweist auf folgende Tabelle:
 1. Jeder Mitarbeiter gehört zu einem Eintrag der Tabelle *Abteilung*.
- Die Tabelle *Zuordnung_SF_FZ* verbindet Schadensfälle und Fahrzeuge und verweist auf folgende Tabellen:
 1. Jedes beteiligte Fahrzeug gehört zu einem Eintrag der Tabelle *Fahrzeug*.
 2. Jeder Schadensfall muss in der Tabelle *Schadensfall* registriert sein.
- Die Tabelle *Versicherungsvertrag* verweist auf folgende Tabellen:
 1. Jeder Vertrag wird von einer Person aus der Tabelle *Mitarbeiter* bearbeitet.
 2. Zu jedem Vertrag gehört ein Eintrag der Tabelle *Fahrzeug*.
 3. Zu jedem Vertrag gehört ein Eintrag der Tabelle *Versicherungsnehmer*.

Durch diese Verknüpfungen werden mehrere Vorteile erreicht:

- Informationen werden nur einmal gespeichert.
Beispiel: Der Name und Sitz einer Abteilung muss nicht bei jedem Mitarbeiter notiert werden.
- Änderungen werden nur einmal vorgenommen.
Beispiel: Wenn die Abteilung umzieht, muss nur der Eintrag in der Tabelle *Abteilung* geändert werden und nicht die einzelnen Angaben bei jedem Mitarbeiter.
- Der Zusammenhang der Daten wird gewährleistet.
Beispiel: Ein Versicherungsnehmer kann nicht gelöscht werden, solange er noch mit einem Vertrag registriert ist.

Damit dies verwirklicht werden kann, werden geeignete Maßnahmen benötigt:

1. Jeder Datensatz muss durch einen Schlüssel eindeutig identifiziert werden können.
2. Die Schlüssel der verschiedenen miteinander verknüpften Datensätze müssen sich zuordnen lassen.

3.4.2. Schlüssel

PrimaryKey (PK): Der Primärschlüssel ist eine Spalte in der Tabelle, durch die eindeutig jede Zeile identifiziert wird (gerne mit dem Namen *ID*). Es kann auch eine Kombination von Spalten als eindeutig festgelegt werden; das ist aber selten sinnvoll. In der Regel sollte diese Spalte auch keine andere „inhaltliche“ Bedeutung haben als die ID.

Beispiele: Die Kombination Name/Vorname kann bei kleinen Datenmengen zwar praktisch eindeutig sein, aber niemals theo-

retisch; irgendwann kommt ein zweiter „Hans Müller“, und dann? Bei einem Mehrbenutzersystem werden häufig mehrere Einträge „gleichzeitig“ gespeichert; es ist besser, wenn das DBMS die Vergabe der ID selbst steuert, als dass die Benutzer sich absprechen müssen. In der Beispieldatenbank wird deshalb in der Tabelle *Mitarbeiter* zwischen der automatisch vergebenen *ID* und der ebenfalls eindeutigen *Personalnummer* unterschieden.

ForeignKey (FK): Über Fremdschlüssel werden die Tabellen miteinander verknüpft. Einem Feld in der einen Tabelle wird ein Datensatz in einer anderen Tabelle zugeordnet; dieser wird über den Primärschlüssel bereitgestellt. Es kann auch eine Kombination von Spalten verwendet werden; da sich der Fremdschlüssel aber auf einen Primärschlüssel der anderen Tabelle beziehen muss, ist dies ebenso selten sinnvoll. Die „Datenbank-Theorie“ geht sogar soweit, dass die Schlüsselfelder dem Anwender gar nicht bekannt sein müssen.

Beispiele stehen in der obigen Aufstellung. Nähere Erläuterungen sind im Kapitel FREMDSCHLÜSSEL-BEZIEHUNGEN⁷ zu finden.

Index: Ein Suchbegriff dient zum schnellen Zugriff auf Datensätze innerhalb einer Tabelle. *Die Mehrzahl lautet nach Duden ‚Indizes‘, auch ‚Indexe‘ ist möglich; in der EDV wird oft auch der englische Plural ‚Indexes‘ verwendet.* Dies gehört zwar nicht zum „Kernbereich“ eines relationalen DBMS, passt aber (auch wegen der umgangssprachlichen Bedeutung des Wortes „Schlüssel“) durchaus hierher.

- Der Primärschlüssel ist ein Suchbegriff, mit dem eindeutig ein Datensatz gefunden werden kann.
- Mit einem Index kann die Suche nach einem bestimmten Datensatz oder einer Datenmenge beschleunigt werden.

Beispiel: die Suche nach PLZ

- Die Werte einer Spalte oder einer Kombination von Spalten sollen eindeutig sein.

Beispiel: die Personalnummer

Nähere Erläuterungen sind in den Kapiteln im Teil *Erweiterungen* ab DDL – EINZELHEITEN⁸ zu finden.

7 Kapitel 28 auf Seite 343

8 Kapitel 27 auf Seite 321

3.5. Siehe auch

Über Wikipedia sind weitere Informationen zu finden:

- RELATIONALE DATENBANK⁹
- ENTITÄTENMODELL¹⁰ und ENTITY-RELATIONSHIP-MODELL¹¹
- NORMALISIERUNG¹²
- TABELLENKALKULATION¹³ im Gegensatz zu Datenbank-Tabellen
- NULLWERT¹⁴

9 [HTTP://DE.WIKIPEDIA.ORG/WIKI/RELATIONALE%20DATENBANK](http://de.wikipedia.org/wiki/Relationale%20Datenbank)

10 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ENTIT%E4T%20%28INFORMATIK%29](http://de.wikipedia.org/wiki/Entit%C3%A4tenmodell)

11 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ENTITY-{}RELATIONSHIP-{}MODELL](http://de.wikipedia.org/wiki/Entity-Relationship-Modell)

12 [HTTP://DE.WIKIPEDIA.ORG/WIKI/NORMALISIERUNG%20%28DATENBANK%29](http://de.wikipedia.org/wiki/Normalisierung%20%28Datenbank%29)

13 [HTTP://DE.WIKIPEDIA.ORG/WIKI/TABELLENKALKULATION](http://de.wikipedia.org/wiki/Tabellenkalkulation)

14 [HTTP://DE.WIKIPEDIA.ORG/WIKI/NULLWERT](http://de.wikipedia.org/wiki/Nullwert)

4. Normalisierung

In diesem Kapitel werden einige Überlegungen angestellt, wie eine Datenbank konzipiert werden soll.

Dieses Kapitel geht zwar über die Anforderungen einer *Einführung* hinaus. Denn es richtet sich weniger an „einfache“ Anwender, die eine vorhandene Datenbank benutzen wollen, sondern an (künftige) Programmentwickler, die eine Datenbank entwerfen und erstellen. Aber wann soll man darüber sprechen, wenn nicht am Anfang?

4.1. Grundgedanken

Schon das einfache Muster einer unzureichenden Tabelle wie im vorigen Kapitel weist auf ein paar Forderungen hin, die offensichtlich an eine sinnvolle Struktur gestellt werden sollten:

Redundanz vermeiden: Eine Information, die an mehreren Stellen benötigt wird, soll nur einmal gespeichert werden. Dadurch müssen auch sämtliche Änderungen, die solche Informationen betreffen, nur an einer Stelle erledigt werden.

Im Beispiel waren das u. a. die Angaben zum Fahrzeughalter und zum Sachbearbeiter (mit seinen Telefonnummern), die bei jedem Vertrag angegeben werden.

Wiederholungen trennen: Die Zusammenfassung gleichartiger Informationen ist nicht sehr übersichtlich. Im vorigen Kapitel waren zunächst Festnetz- und Mobil-Rufnummer in einer Spalte untergebracht. Das ist praktisch, wenn man sich keine Gedanken darüber machen will, welche und wie viele Kontaktnummern es gibt. Es ist ziemlich unpraktisch, wenn man gezielt einzelne Nummern suchen will oder von einer Nummer auf den Sachbearbeiter schließen will. Sinnvoller ist es, Festnetz- und Mobilnummer in getrennten Spalten zu speichern.

Primärschlüssel verwenden: Jeder Datensatz muss eindeutig identifiziert werden, damit die Informationen verwendet werden können.

In der ungenügenden Tabelle wurde wegen der Zeilennummern darauf verzichtet. Aber selbst wenn man die Suche in zigtausend Zeilen für machbar hielte, spätestens bei der Aufteilung in mehrere Tabellen braucht man Werte, um die Zusammenhänge eindeutig darzustellen.

Bei diesen Überlegungen handelt es sich nur um ein paar Gedanken, die sich einem aufdrängen. Das wollen wir nun besser strukturieren.

4.2. Tabellenkalkulation als Ausgangspunkt

Überlegen wir uns zunächst, welche Informationen benötigt werden.

Grundlegende Daten

Für die Verträge müssen die folgenden Angaben gespeichert werden.

- die **Vertragsdaten** selbst, also Nummer, Abschlussdatum und Art des Vertrags (HP = Haftpflicht, TK = Teilkasko, VK = Vollkasko) und der Prämienberechnung
- der **Vertragspartner** mit Name, Anschrift, Telefonnummern, dazu bei natürlichen Personen Geschlecht und Geburtsdatum sowie bei juristischen Personen der Eintrag im Handelsregister o. ä.
- das versicherte **Fahrzeug** mit Kennzeichen, Hersteller und Typ (die Farbe ist eine Zusatzinformation, die bei manchen Suchvorgängen hilfreich sein kann)
- der zuständige **Sachbearbeiter** mit Name, Abteilung und Telefonnummern

Ein Teil der Angaben muss immer vorhanden sein, ein anderer Teil je nach Situation (Geschlecht bei Personen) und ein weiterer Teil nur bei Bedarf (Mobiltelefon).

Schnell stellen wir fest, dass ein Versicherungsnehmer auch **mehrere Fahrzeuge** versichern kann. Andererseits kann ein Fahrzeug zu mehreren Verträgen gehören, wenn Haftpflicht und Vollkasko getrennt abgeschlossen werden oder wenn zur Haftpflicht vorübergehend Vollkasko fürs Ausland hinzukommt. Jeder Vertrag ist ein eigener Datensatz; also stehen bei jedem Datensatz die

Angaben des Vertragspartners und des Fahrzeugs. Um alle Verträge eines Kunden gemeinsam anzuzeigen, brauchen wir folglich ein paar Angaben zur Organisation:

- Kundennummer und laufende Nummer seiner Verträge als eindeutiger Suchbegriff (vielleicht als Primärschlüssel)
- Vertragsnummer als eindeutiger Wert
- Fahrzeug-Kennzeichen als Wert, der meistens (aber nicht immer) nur einmal vorkommt

Schadensfälle

Nun sollen zu einem Vertrag auch die Schadensfälle gespeichert werden. Wir benötigen also eine oder mehrere Spalten mit den erforderlichen Angaben (Datum und Art des Schadens, Sachbearbeiter der Schadensabwicklung, andere beteiligte Fahrzeuge); aber wie wird das am besten gespeichert?

- eine gemeinsame Spalte für alle diese Angaben für alle Schadensfälle (denn die Mehrzahl der Fahrzeuge fährt schadensfrei)
- je eine Spalte für alle Angaben eines Schadensfalls (werden dann zwei oder drei Spalten benötigt, oder muss man für „Mehrfach-Sünder“ gleich zehn Spalten vorsehen?)
- oder einzelne Datensätze für jeden Schaden eines Fahrzeugs (sodass sich alle Fahrzeug- und Vertragsdaten in diesen Datensätzen wiederholen und nur die Schadensangaben unterscheiden)

Ungeklärt bleibt dabei noch dieses Problem: Wie viele versicherte Schadensgegner gibt es denn – keine (wenn ein Reh angefahren wird), einen (z. B. beim Parken) oder viele (bei einem Auffahrunfall auf der Autobahn)?

Entscheiden wir uns deshalb provisorisch für ein eigenes **Arbeitsblatt Schadensfälle** mit folgender Struktur:

- je eine Spalte für die Angaben, die direkt zum Schadensfall gehören
- vorläufig 5 Gruppen für maximal 5 beteiligte Fahrzeuge
- jede dieser Gruppen enthält in einzelnen Spalten Fahrzeug-Kennzeichen und die Halter-Angaben (Name, Anschrift)

Damit stehen die Fahrzeugdaten in beiden Arbeitsblättern. Als praktisches Problem kommt hinzu: Für die Schadenshäufigkeit eines bestimmten Fahrzeugs muss man es in fünf Spalten herausuchen. *Die Beschränkung auf fünf beteiligte Fahrzeuge wird im nächsten Schritt aufgelöst, machen wir uns dazu keine weiteren Gedanken.*

Auf diesem Weg kann jedenfalls keine sinnvolle Struktur erreicht werden.

„Update-Anomalien“

Mit diesem Begriff werden die folgenden Unklarheiten bezeichnet. Dabei handelt es sich um weitere Probleme bei einer ungenügenden Struktur.

Einfügen-Anomalie: In der ersten Tabelle sind u. a. die Sachbearbeiter enthalten. Es wäre sinnvoll, auch alle anderen Mitarbeiter der Gesellschaft hier einzutragen. Aber bei einem Vertrag werden zwingend Vertragsnummer und Abschlussdatum benötigt; dieser Zwang verhindert das Speichern der Teilinformation *Sachbearbeiter* ohne Bezug auf einen Vertrag.

Löschen-Anomalie: Wenn ein Vertrag gekündigt und abgelaufen ist und deshalb gelöscht wird, dann sind in der ersten Tabelle auch alle Angaben zum Vertragspartner verloren. Wenn er drei Tage später einen neuen Vertrag abschließt, müssen alle seine Angaben neu aufgenommen werden. Und was soll mit den Schadensfällen geschehen, die zu diesem Vertrag gespeichert sind?

Ändern-Anomalie: Wenn der Sachbearbeiter wechselt, muss sein Name bei allen von ihm betreuten Verträgen geändert werden, obwohl eine einzige Änderung ausreichend sein sollte. (Dies ist auf die Datenredundanz zurückzuführen, dass nämlich dieser Hinweis vielfach gespeichert ist statt einmalig.)

All diesen Problemen wollen wir nun durch eine deutlich verbesserte Datenstruktur begegnen. Nehmen wir dazu zunächst an, dass alle benötigten Informationen in den beiden Arbeitsblättern *Verträge* und *Schadensfälle* der Tabellenkalkulation stehen, und beginnen wir, dies sinnvoll zu strukturieren.

4.3. Die 1. Normalform

Am „grausamsten“ für den Aufbau der Tabelle und die praktische Arbeit ist die vielfache Wiederholung gleicher Informationen.

Sowohl beim Namen des Fahrzeughalters als auch bei den Mitarbeitern steht der Name bisher in einer Spalte in der Form „Nachname, Vorname“, was als Suchbegriff geeignet ist. Nun benötigen wir aber auch eine persönliche Anrede (Name mit Titel) und eine Briefanschrift (Vorname, Name). Soll dies in weiteren

Spalten gespeichert werden? Nein, denn all dies kann aus den Einzelangaben „Name“ und „Vorname“ zusammengesetzt werden.

Ein Schadensfall ist bisher auf fünf beteiligte Fahrzeuge beschränkt, auch wenn selten mehr als zwei Beteiligungen benötigt werden. Wenn nun ein sechstes, zehntes oder zwanzigstes Fahrzeug beteiligt ist, muss dann jedesmal die Tabellenstruktur (und jedes Makro, das auf diese Spalten zugreift) geändert werden?!

Damit haben wir bereits zwei Regeln, die als **Definition** der ersten Normalform gelten. Hinzu kommt eine dritte Regel, die zwar formal nicht erforderlich ist; aber aus praktischen Gründen gibt es keine sinnvolle Lösung ohne diese Regel.

i Die 1. Normalform

1. Jede Spalte enthält nur unteilbare (atomare, atomische) Werte.
2. Spalten, die gleiche oder gleichartige Informationen enthalten, sind in eigene Tabellen (Relationen) auszulagern.
3. Jede Tabelle enthält einen Primärschlüssel.

Verletzung der 1. Normalform

Unsere Ausgangstabelle verstößt an vielen Stellen gegen diese Regeln:

- **Zusammengesetzte Werte** befinden sich z. B. an folgenden Stellen:
 - Fahrzeughalter: Name und Vorname, PLZ und Ort, ggf. Straße und Hausnummer
 - Sachbearbeiter: Name und Vorname, Festnetz- und Mobilnummer
- **Wiederholungen** finden sich vor allem hier:
 - mehrere Fahrzeuge bei einem Schadensfall

Vorgehen zur Herstellung der 1. Normalform

Es werden also zwei Schritte benötigt:

1. Zusammengesetzte Werte werden in Einzelinformationen aufgeteilt: je eine Spalte für jeden unteilbaren Wert.
2. Wiederholungen werden in getrennte Tabellen ausgelagert; welche Daten zusammengehören, wird durch eine laufende Nummer angegeben.

Eine erste Verbesserung

An jeder Stelle, die Namen oder Anschrift enthält, werden getrennte Spalten verwendet, beispielsweise im Arbeitsblatt *Verträge*:

Kundennummer	Vertrag	Abschluss	Halter_Name	Halter_PLZ	Halter_-
Ort	Sachbearbeiter_N	Sachbearbeiter_V	Telefon		
1	DG-01	03.05.1974	Heckel Obsthandel GmbH	46282	
Dorsten	Pohl		Helmut	0201/4014186	
1	DG-02	04.07.1974	Heckel Obsthandel GmbH	46282	
Dorsten	Pohl		Helmut	0201/4014186	

Die Tabelle *Schadensfälle* wird auf die eigentlichen Daten beschränkt; die beteiligten Fahrzeuge stehen in einer eigenen Tabelle *Zuordnungen*.

Nummer	Datum	Schadensort	Beschreibung	Sachbearbeiter_-
N	Sachbearbeiter_V	Telefon		
1	02.03.2007	Recklinghausen, Bergknappenstr. 144	Gartenzaun gestreift	
		Schindler Christina	0201/4012151	
2	11.07.2007	Haltern, Hauptstr. 46	beim Ausparken hat ...	
		Aliman Zafer	0201/4012161	

Die Anzahl der beteiligten Fahrzeuge an einem Schadensfall wird durch eine eigene Tabelle *Zuordnungen* berücksichtigt:

Nummer	Fahrzeug	Hersteller	Typ	Halter_Name	Halter_Vorname	Halter_-
PLZ	Halter_Ort	Halter_Straße				
1	RE-LM 902	Opel	Corsa	Heckel	Obsthandel GmbH	
46282	Dorsten		Gahlener Str. 40			
2	BO-GH 102	Volvo	C30	Geissler	Helga	
44809	Bochum		Steinbankstr. 15			
2	RE-CD 456	Renault	Twingo	Cornelsen	Dorothea	
44577	Castrop-Rauxel		Kiefernweg 9			

Die Zusatzbedingung eines Primärschlüssels wurde gleichzeitig erfüllt; die betreffenden Spalten wurden unterstrichen.

4.4. Die 2. Normalform

Eine weitere Wiederholung in den „Monster-Tabellen“ sind die Angaben zum Halter bei den *Verträgen* und den *Zuordnungen* oder die Fahrzeugdaten sowohl bei den *Verträgen* als auch bei den *Zuordnungen*. Auch diese werden in

eigene Tabellen ausgelagert; das ergibt sich als **Definition** aus der folgenden Regel:

i Die 2. Normalform

1. Die Tabelle erfüllt die 1. Normalform.
2. Alle Informationen in den Spalten, die nicht Teil des Primärschlüssels sind, müssen sich auf den gesamten Primärschlüssel beziehen.

Man sagt dazu auch, dass die Informationen funktional abhängig sind von der Gesamtheit der Schlüsselwerte. Umgekehrt formuliert bedeutet es: Wenn eine Spalte nur zu einem einzelnen Schlüsselfeld gehört, ist die 2. Normalform nicht erfüllt.

Während sich die 1. Normalform auf die einzelnen Spalten und Wiederholungen innerhalb eines Datensatzes bezieht, befasst sich die 2. Normalform mit Wiederholungen bei verschiedenen Datensätzen.

Verletzung der 2. Normalform

Unsere derzeitigen Tabellen verstoßen mehrfach gegen diese Regel:

- Bei den *Verträgen* beziehen sich Name und Anschrift des Vertragspartners nur auf die Kundennummer, aber nicht auf die Vertragsnummer.
- Bei den *Verträgen* stehen auch die Fahrzeugdaten. (Dies wurde bei der Beschreibung der grundlegenden Daten erwähnt, fehlt aber in den bisherigen Beispielen.) Diese beziehen sich auf die Vertragsnummer, haben aber nur indirekt etwas mit Name und Anschrift des Vertragspartners zu tun.
- Bei den *Zuordnungen* der *Schadensfälle* gehören sowohl die Fahrzeug- als auch die Halterdaten nur zu einem Fahrzeug (also dem Kennzeichen), aber nicht zu einem Schadensfall.

Vorgehen zur Herstellung der 2. Normalform

Es müssen alle „unpassenden“ Informationen in eigene Tabellen ausgelagert werden. Der Primärschlüssel wird vereinfacht, sodass er sich nur auf die „eigentlichen“ Informationen einer Tabelle bezieht.

- Aus den *Verträgen* werden alle Angaben des Vertragspartners entfernt und in eine Tabelle *Versicherungsnehmer* übertragen. Der Primärschlüssel besteht

nur noch aus der Spalte *Vertrag*. Die Spalte *Kundennummer* ist nur noch ein Fremdschlüssel als Verweis auf die neue Tabelle *Versicherungsnehmer*.

- Aus den *Verträgen* werden alle Angaben des Fahrzeugs entfernt und in eine neue Tabelle *Fahrzeuge* übertragen. Das Fahrzeug-Kennzeichen ist hier nur noch ein Fremdschlüssel als Verweis auf die Tabelle *Fahrzeuge*.
- Aus den *Zuordnungen* werden alle Angaben der Fahrzeuge entfernt und in eine Tabelle *Fahrzeuge* übertragen. Die Tabelle *Zuordnungen* besteht nur noch aus den Spalten des Primärschlüssels.

Damit stehen die Fahrzeugdaten nur noch in einer Tabelle – sowohl für die Verträge als auch für die Schadensfälle (genauer: die Zuordnungen).

Eine zweite Verbesserung

Die ursprüngliche Tabelle *Verträge* beschränkt sich jetzt auf diese Angaben:

Vertrag	Abschluss	Typ	Kundennummer	Fahrzeug	Sachbearbeiter_
N	Sachbearbeiter_V	Telefon			
DG-01	03.05.1974	HP	1	RE-LM 901	Pohl
Helmut		0201/4014186			
DG-02	04.07.1974	HP	1	RE-LM 902	Pohl
Helmut		0201/4014186			

Die neue Tabelle *Versicherungsnehmer* umfasst die Daten, die sich auf den Vertragspartner beziehen:

Kundennummer	Name	Vorname	Geburtsdatum	PLZ	Ort	Straße	Hausnummer
1	Heckel	Obsthandel GmbH		46282	Dorsten		
	Gahlener Str.	40					
5	Geissler	Helga	13.01.1953	44809	Bochum		
	Steinbankstr.	15					

Die neue Tabelle *Fahrzeuge* umfasst nur noch die Daten, die sich auf das Fahrzeug selbst beziehen. Name und Anschrift des Fahrzeughalters werden durch die Kundennummer, also den Verweis auf die Tabelle *Versicherungsnehmer* ersetzt.

Fahrzeug	Hersteller	Typ	Farbe	Halter
RE-LM 902	Opel	Corsa	ocker	1
BO-GH 102	Volvo	C30	rot	5
RE-CD 456	Renault	Twingo	ocker	3

Die Tabelle *Schadensfälle* muss nicht angepasst werden. Die Tabelle *Zuordnungen* vereinfacht sich radikal:

Nummer	Fahrzeug
1	RE-LM 902
2	BO-GH 102
2	RE-CD 456

Die 2. Normalform kann ganz einfach dadurch gewährleistet werden, dass sich der Primärschlüssel nur auf eine Spalte bezieht.

4.5. Die 3. Normalform

Beseitigen wir noch die übrigen Wiederholungen, nämlich die Sachbearbeiter bei den Verträgen und den Schadensfällen sowie die Hersteller bei den Fahrzeugen. Diese werden ebenfalls in eigene Tabellen ausgelagert gemäß der **Definition** nach der folgenden Regel:

i Die 3. Normalform

1. Die Tabelle erfüllt die 2. Normalform.
2. Informationen in den Spalten, die nicht Teil des Primärschlüssels sind, dürfen funktional nicht voneinander abhängen.

Die 3. Normalform befasst sich also mit Wiederholungen bei verschiedenen Datensätzen, die nur zusätzliche Informationen bereitstellen.

Verletzung der 3. Normalform

Unsere derzeitigen Tabellen verstoßen in folgender Hinsicht gegen diese Regel:

- Name, Vorname und Telefonnummer eines Sachbearbeiters hängen voneinander ab. Sie haben aber nur insgesamt etwas mit dem Vertrag bzw. dem Schadensfall zu tun, nicht als einzelne Information.
- Hersteller und Typ eines Fahrzeugs hängen voneinander ab. Sie haben aber nur insgesamt etwas mit dem Fahrzeug zu tun, nicht als einzelne Information.

Eine andere Erklärung dafür ist, dass die Zusatzinformation auch ohne Bezug zum eigentlichen Datensatz gültig bleibt. Der Sachbearbeiter gehört zum Un-

ternehmen unabhängig von einem bestimmten Vertrag. Der Fahrzeughersteller existiert unabhängig davon, ob ein bestimmtes Fahrzeug noch fährt oder inzwischen verschrottet worden ist.

Vorgehen zur Herstellung der 3. Normalform

Alle „unpassenden“ Informationen kommen wieder in eigene Tabellen. Ihre Spalten werden ersetzt durch einen Fremdschlüssel als Verweis auf die neue Tabelle.

- Aus den *Verträgen* und den *Schadensfällen* werden alle Angaben zum Sachbearbeiter entfernt und in eine Tabelle *Mitarbeiter* übertragen. Die Spalte *Sachbearbeiter* verweist nur noch als Fremdschlüssel auf die neue Tabelle *Mitarbeiter*.

Dies löst automatisch auch das oben erwähnte Problem: Wir können nun alle Mitarbeiter in einer gemeinsamen Tabelle speichern.

- Aus den *Fahrzeugen* werden alle Angaben zu Hersteller und Typ entfernt und in eine neue Tabelle *Fahrzeugtypen* übertragen. Die Spalten *Hersteller* und *Typ* werden ersetzt durch einen Fremdschlüssel als Verweis auf die Tabelle *Fahrzeugtypen*.

Um es korrekt zu machen, gehört der Hersteller in eine weitere Tabelle *Fahrzeughersteller*; in der Tabelle *Fahrzeugtypen* verweist er als Fremdschlüssel auf diese weitere Tabelle.

Eine weitere Verbesserung

Die Tabellen *Verträge* und *Schadensfälle* werden also nochmals vereinfacht:

Vertrag	Abschluss	Typ	Kundennummer	Fahrzeug	Sachbearbeiter
DG-01	03.05.1974	HP	1	RE-LM 901	9
DG-02	04.07.1974	HP	1	RE-LM 902	9

Nummer	Datum	Schadensort	Beschreibung	Sachbearb
1	02.03.2007	Recklinghausen, Bergknappe...	Gartenzaun	
gestreift	14			
2	11.07.2007	Haltern, Hauptstr. 46	beim Ausparken	
hat ...	15			

Hinzu kommt die neue Tabelle *Mitarbeiter*:

Nummer	Nachname	Vorname	Telefon
9	Pohl	Helmut	0201/4014186
14	Schindler	Christina	0201/4012151
15	Aliman	Zafer	0201/4012161

In gleicher Weise wird die Tabelle *Fahrzeuge* gekürzt; die Angaben werden in die neuen Tabellen *Fahrzeugtypen* und *Fahrzeughersteller* ausgelagert.

4.6. Zusätzliche Maßnahmen

In der Theorie gibt es noch eine 4. und eine 5. Normalform (und eine Alternative zur 3. Normalform). Dazu sei auf den Wikipedia-Artikel und die dortigen Hinweise (Quellen, Literatur, Weblinks) verwiesen. In der Praxis ist aber eine Datenbank, die den Bedingungen der 3. Normalform entspricht, sehr gut konzipiert. Weitere Normalisierungen bringen kaum noch Vorteile; stattdessen können sie die Übersichtlichkeit und die Geschwindigkeit beim Datenzugriff beeinträchtigen.

4.6.1. Verzicht auf Zusatztabelle

Beispielsweise wiederholen sich in der Tabelle *Versicherungsnehmer* die **Ortsangaben**. Dabei gehören die Kombinationen PLZ/Ort immer zusammen; auch wenn der Kunde umzieht, ist eine solche Kombination unverändert gültig. Also könnte man in der Tabelle die Adresse wie folgt speichern:

Kd-Nr	Name	Vorname	Geburtsdatum	PLZ	Alort	Straße	Hausnummer
1	Heckel	Obsthandel GmbH		46282	10884500		
	Gahlener Str.	40					
5	Geissler	Helga	13.01.1953	44809	05902500		
	Steinbankstr.	15					

Der Ortsname ist dann zu finden über die PL-Datei der Deutschen Post AG (mit PLZ/Alort als Primärschlüssel):

Dateiversion	Geltung	PLZ	Alort	PLZ-Arten	Ortsname
PL 0509 244	20010101	44809	05902500	06 2 2	Bochum
PL 0509 244	20050329	46282	10884500	06 2 2	Dorsten

Das gleiche Verfahren ist möglich für die Straßennamen. Es sorgt dafür, dass nur gültige Anschriften gespeichert sind und auch bei Eingemeindungen die neuen Angaben eindeutig übernommen werden. Aber selbst wenn man wegen der Datensicherheit mit *Alort* arbeiten will, wird man für die praktische Arbeit den Ortsnamen in der Adressendatei behalten wollen.

4.6.2. Primärschlüssel nur zur Identifizierung

Im vorigen Kapitel hatten wir kurz darauf hingewiesen, dass der Primärschlüssel nur die Bedeutung als ID haben sollte.

Eine Begründung liefert die obige Tabelle *Fahrzeuge*, bei der das Kennzeichen zur Identifizierung benutzt wurde. Wenn der Fahrzeughalter in einen anderen Kreis umzieht, bekommt er ein neues Kennzeichen. Dann müsste es an allen Stellen geändert werden, an denen es gespeichert ist. Nun darf die Vertragsabteilung nur die Verträge ändern, die Schadensabwicklung die Schadensfälle (und wer weiß, wo noch darauf Bezug genommen wird). Jede Ummeldung verursacht also erheblichen Mehraufwand.

Sorgen wir also für mehr Datensicherheit und Arbeitsvereinfachung:

- Der Primärschlüssel ist eine automatisch zugewiesene ID. Diese ID wird niemals geändert.
- Die Identifizierung, die der Benutzer kennt, ist eine einzelne Spalte in einer einzigen Tabelle.

Beispiele: Vertragsnummer, Fahrzeug-Kennzeichen, Personalnummer

- Die Verknüpfungen mit anderen Tabellen regelt die Datenbank mit Hilfe der ID selbständig.

4.6.3. Änderungsdaten

In vielen Fällen ist es sinnvoll, wenn in einer Tabelle nicht nur der aktuelle Zustand gespeichert ist, sondern der Verlauf.

- Beispiel mit **Änderung des Kennzeichens**: Wenn die Polizei nach einem Unfallverursacher sucht, der inzwischen umgezogen ist, wird für das alte (nicht mehr gültige) Kennzeichen kein Fahrzeug gefunden.
- **Adressänderungen auf Termin** legen: Es wäre viel zu aufwändig, wenn alle Änderungen gleichzeitig an dem Stichtag, an dem sie gelten sollen,

eingetragen werden müssten. Besser ist es, sie sofort zu speichern mit Angabe des Geltungsdatums; die Datenbank holt abhängig vom aktuellen und dem Geltungsdatum immer die richtige Version.

- **Aktueller Steuersatz:** Die Abrechnung einer Versandfirma muss immer mit dem richtigen Steuersatz rechnen, auch wenn er mitten im Jahr geändert wird. Als Steuersatz 1 für die Mehrwertsteuer wird dann bei einer „älteren“ Rechnung mit 16 % und bei einer Rechnung neueren Datums mit 19 % gerechnet.

Für alle solche Fälle erhalten Tabellen gerne zusätzliche Spalten *gültig von* und *gültig bis*. Anstelle einer Änderung werden Datensätze verdoppelt; die neuen Informationen ersetzen dabei die bisher gültigen.

Selbstverständlich erfordert eine solche Datenbankstruktur höheren Aufwand sowohl beim Entwickler der Datenbank als auch beim Programmierer der Anwendung. Der zusätzliche Nutzen rechtfertigt diesen Aufwand.

4.6.4. Reihenfolge ohne Bedeutung

Beim Aufbau einer Datenbank darf es nicht relevant sein, in welcher Reihenfolge die Zeilen und Spalten vorkommen: Die Funktionalität der Datenbank darf nicht davon abhängen, dass zwei Spalten in einer bestimmten Folge nacheinander kommen oder dass nach einem Datensatz ein bestimmter zweiter folgt.

- Ein Datensatz (also eine Zeile) steht nur über den Primärschlüssel bereit.
- Eine Spalte steht nur über den Spaltennamen zur Verfügung.

Nur ausnahmsweise wird mit der Nummer einer Zeile oder Spalte gearbeitet, z. B. wenn ein „Programmierer-Werkzeug“ nur indizierte Spalten kennt.

4.7. Zusammenfassung

Fassen wir diese Erkenntnisse zusammen zu einigen Regeln, die bei der Entwicklung einer Datenbank-Struktur zu beachten sind.

4.7.1. Allgemeine Regeln

Von der Realität ausgehen: Erstellen Sie Tabellen danach, mit welchen Dingen (Objekten) Sie arbeiten. Benutzen Sie aussagekräftige Namen für Tabellen und Spalten.

Einzelne Informationen klar trennen: Alle Informationen werden in einzelne Spalten mit unteilbaren Werten aufgeteilt.

Vollständigkeit: Alle möglichen Informationen sollten von vornherein berücksichtigt werden. Nachträgliche Änderungen sind zu vermeiden.

Keine berechneten Werte speichern: Eine Information, die aus vorhandenen Angaben zusammengestellt werden kann, wird nicht als eigene Spalte gespeichert.

Kleine Tabellen bevorzugen: Wenn eine Tabelle sehr viele Informationen umfasst, ist es besser, sie zu unterteilen und über einen einheitlichen Primärschlüssel zu verbinden.

Keine doppelten Speicherungen: Es darf weder ganze doppelte Datensätze geben noch wiederholte Speicherungen gleicher Werte (Redundanz von Daten). Doppelte Datensätze werden durch Primärschlüssel und Regeln zur Eindeutigkeit in Felder vermieden; Datenredundanz wird durch getrennte Tabellen ersetzt, sodass es nur die Fremdschlüssel mehrfach gibt.

Wichtig ist auch, dass der Aufbau der Datenbank beschrieben und ggf. begründet wird. Hilfreich sind dabei graphische Darstellungen, wie sie von vielen Datenbank-Programmen angeboten werden.

4.7.2. Abweichungen

Die vorstehenden Überlegungen werden nicht immer beachtet. Für fast jede Regel gibt es begründete Ausnahmen

Beispielsweise enthält eine Datenbank mit Adressen in der Regel die Adressen im Klartext und nicht nur als Verweise auf die Nummern von Orten, Ortsteilen, Straßen und Straßenabschnitten der Deutschen Post.

Entscheiden Sie erst nach bewusster Abwägung von Vor- und Nachteilen, wenn Sie von einer der Regeln abweichen wollen.

4.8. Siehe auch

Über Wikipedia sind weitere Informationen zu finden:

- NORMALISIERUNG¹
- RELATIONALE DATENBANK²
- EDGAR F. CODD³ und seine 12 REGELN⁴ für relationale Datenbanken
- UPDATE-ANOMALIEN⁵
- REDUNDANZ VON DATEN⁶

Für eine sorgfältige Planung einer Adressen-Datenbank hilft die Datenstruktur der Deutschen Post:

- BROSCHÜRE DATAFACTORY⁷ (360 kB) als Überblick
Unter POST DIREKT⁸ kann die Datei 20101230_HB_Leitdaten.pdf mit der Datensatzbeschreibung kostenfrei bestellt werden.

1 [HTTP://DE.WIKIPEDIA.ORG/WIKI/NORMALISIERUNG%20%28DATENBANK%29](http://de.wikipedia.org/wiki/Normalisierung%20%28Datenbank%29)

2 [HTTP://DE.WIKIPEDIA.ORG/WIKI/RELATIONALE%20DATENBANK](http://de.wikipedia.org/wiki/Relationale%20Datenbank)

3 [HTTP://DE.WIKIPEDIA.ORG/WIKI/EDGAR%20F.%20CODD](http://de.wikipedia.org/wiki/Edgar%20F.%20Codd)

4 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ONLINE_ANALYTICAL_PROCESSING%2312_REGELN_NACH_CODD](http://de.wikipedia.org/wiki/Online_Analytical_Processing%2312_Regeln_Nach_Codd)

5 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ANOMALIE%20%28INFORMATIK%29](http://de.wikipedia.org/wiki/Anomalie%20%28Informatik%29)

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/REDUNDANZ%20%28INFORMATIONSTHEORIE%29](http://de.wikipedia.org/wiki/Redundanz%20%28Informationstheorie%29)

7 [HTTP://WWW.DEUTSCHEPOST.DE/DOWNLOADSERVLET?TARGET=/MLM.NF/DPAG/IMAGES/D/DATAFACTORY/20100205_DATAFACTORY_INTERNET.PDF](http://www.deutschepost.de/downloadServlet?target=/mlm.nf/dpag/images/d/datafactory/20100205_datafactory_internet.pdf)

8 [MAILTO:INFO@POSTDIREKT.DE](mailto:info@postdirekt.de)

5. Beispieldatenbank

Dieses Kapitel bespricht die Grundlagen der Beispieldatenbank. Alle Einzelheiten der Tabellen stehen im Anhang unter TABELLENSTRUKTUR DER BEISPIELDATENBANK¹.

i Hinweis

Alle Beispiele, Erläuterungen und Aufgaben beziehen sich auf diese Datenstruktur. Bitte schlagen Sie (soweit erforderlich) immer dort nach.

5.1. Sachverhalt

Die Beispieldatenbank in diesem Buch versucht, eine Versicherungsgesellschaft für Kfz-Versicherungen abzubilden. Das Beispiel ist eine starke Vereinfachung der Realität; zum Beispiel fehlen alle Teile der laufenden Abrechnung der Prämien und Schadensfälle.

Die relevanten Begriffe sind für eine bessere Übersichtlichkeit fett gekennzeichnet. Folgender Sachverhalt wird in der Datenbank abgebildet:

Die Versicherungsgesellschaft *UnsereFirma* verwaltet mit dieser Datenbank ihre Kundendaten und die Schadensfälle. Wegen der Schadensfälle müssen auch Daten „fremder“ Kunden und Versicherungen gespeichert werden.

Jeder **Versicherungsvertrag** wird mit einem **Versicherungsnehmer** über genau ein **Fahrzeug** abgeschlossen. Der Versicherungsvertrag wird durch folgende Eigenschaften gekennzeichnet:

- Vertragsnummer (Pflicht, eindeutig)
- Datum des Abschlusses, Art des Vertrages (Pflicht); dabei gibt es die Arten Haftpflicht (HP), Haftpflicht mit Teilkasko (TK), Vollkasko (VK).

¹ Kapitel 36 auf Seite 469

- Verweis auf den Versicherungsnehmer (Pflicht)
- Verweis auf das Fahrzeug (Pflicht, eindeutig)
- Verweis auf den Mitarbeiter, der den Vertrag bearbeitet (Pflicht)

Der **Versicherungsnehmer** ist gekennzeichnet durch diese Eigenschaften:

- Kundennummer (Pflicht, eindeutig)
- Name und Anschrift: PLZ, Ort, Straße, Hausnummer (Pflicht)
- bei natürlichen Personen zusätzlich durch Vorname, Geburtsdatum, Datum des Führerscheins (optional)
- Verweis auf eine „Fremdversicherung“, wenn ein (fremdes) Fahrzeug an einem Unfall beteiligt ist (optional)

Das **Fahrzeug** ist gekennzeichnet durch diese Eigenschaften:

- polizeiliches Kennzeichen (Pflicht, eindeutig)
- Farbe (optional)
- Fahrzeugtyp und damit indirekt auch den Fahrzeughersteller (Pflicht)

Ein **Mitarbeiter** ist gekennzeichnet durch diese Eigenschaften:

- Name, Vorname, Geburtsdatum (Pflicht)
- Personalnummer (Pflicht, eindeutig)
- Verweis auf Abteilung, Vermerk, ob es sich um den Leiter der Abteilung handelt (Pflicht)
- Kontaktdaten wie Telefon, Mobiltelefon, Email, Raum (optional)

Die **Abteilung** ist gekennzeichnet durch diese Eigenschaften:

- Nummer (Pflicht, eindeutig)
- Kurzbezeichnung, Bezeichnung (Pflicht, eindeutig)
- Ort (optional)

Zusätzlich gibt es **Dienstwagen**. Dabei handelt es sich um eine Tabelle mit den gleichen Eigenschaften wie bei *Fahrzeug* und zusätzlich:

- Verweis auf den Mitarbeiter, zu dem ein Dienstwagen gehört (optional)
- denn es gibt auch Firmenwagen, die keinem Mitarbeiter persönlich zugeordnet sind

Ein **Schadensfall** ist gekennzeichnet durch diese Eigenschaften:

- Datum, Ort und Umstände des Unfalls (Pflicht)
- Vermerk, ob es Verletzte gab, sowie Höhe des Gesamtschadens (optional, denn die Angaben könnten erst später bekannt werden)
- Verweis auf den Mitarbeiter, der den Schadensfall bearbeitet (Pflicht)

An einem Schadensfall können mehrere Fahrzeuge unterschiedlicher Versicherungen beteiligt sein. (Unfälle mit Radfahrern und Fußgängern werden nicht betrachtet.) Deshalb gibt es eine weitere Tabelle **Zuordnung_SF_FZ**:

- Liste aller Schadensfälle und aller beteiligten Fahrzeuge (Pflicht)
- anteilige Schadenshöhe eines Fahrzeugs an dem betreffenden Unfall (optional)

Über die Verbindung *Schadensfall* → *Fahrzeug* → *Versicherungsvertrag* → *Versicherungsnehmer* → *Versicherungsgesellschaft* können alle beteiligten Gesellschaften festgestellt und in die Schadensabwicklung eingebunden werden.

Bitte beachten Sie auch die unten stehenden Hinweise über FEHLENDE SPALTEN UND EINSCHRÄNKUNGEN².

5.2. Schematische Darstellung

Die vorstehende Struktur kann in folgendem Diagramm dargestellt werden. Die Verweise, nämlich die Verknüpfungen zwischen den Tabellen sind daraus so abzulesen:

- Von jeder Tabelle gibt es einen Verweis auf einen Eintrag in der direkt darunter stehenden Tabelle.
- Zu jedem Vertrag gehört ein Sachbearbeiter; das wird durch den Pfeil nach links außen angedeutet, der in der Tabelle *Mitarbeiter* von rechts hereinkommt.
- Zu jedem Vertrag gehört genau ein Fahrzeug; zu jedem Fahrzeug gehört ein Vertrag.
- Jeder Eintrag in der Liste der Zuordnungen *Schadensfall*, *Fahrzeug* bezieht sich auf ein Fahrzeug und einen Schadensfall; dabei gilt:
 - Zu jedem Schadensfall gehören ein oder mehrere Fahrzeuge.
 - Nicht jedes Fahrzeug hat einen Schadensfall.
- Ein Dienstwagen kann einem Mitarbeiter zugeordnet sein, muss es aber nicht.

Tabellen und ihre Zusammenhänge

² [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23FEHLENDE%20SPALTEN%20UND%20EINSCHR%44NKUNGEN](http://de.wikibooks.org/wiki/%23FEHLENDE%20SPALTEN%20UND%20EINSCHR%44NKUNGEN)

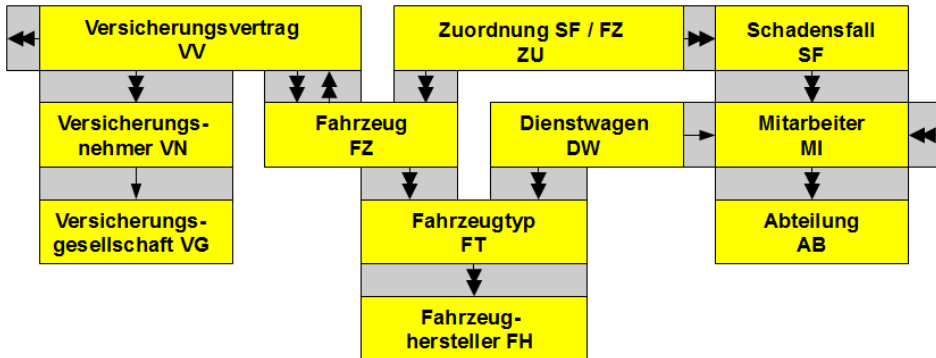


Abb. 1

Doppelte Pfeile zeigen eine Muss-Verknüpfung, einfache eine Kann-Verknüpfung.

Die Kürzel bei den Tabellennamen werden künftig immer wieder als „Alias“ anstelle des „Langnamens“ verwendet. In dieser Übersicht wurden sie wegen der Deutlichkeit großgeschrieben; künftig werden sie kleingeschrieben.

5.3. Tabellenstruktur und Datenbank

Im Anhang ist die Tabellenstruktur der Beispieldatenbank beschrieben, nämlich die Liste der Tabellen und aller ihrer Spalten. Innerhalb des Buches wird die Datenbankstruktur erweitert werden; dabei werden einige der nachstehenden Anmerkungen beachtet.

Im Anhang stehen auch Wege, wie Sie die Beispieldatenbank erhalten.

- Entweder Sie holen sich eine fertige Version der Datenbank:
 - Laden Sie die kompromierte Datei herunter.
 - Benennen Sie die Datei um, soweit erforderlich.
 - Extrahieren Sie die Daten und speichern Sie das Ergebnis an einer passenden Stelle.
- Oder Sie erzeugen die Datenbank mit einem Hilfsprogramm des von Ihnen verwendeten DBMS:
 - Erstellen Sie zunächst eine leere Datenbank, soweit erforderlich.
 - Laden bzw. kopieren Sie das passende Skript zum Erzeugen der Datenbank-Tabellen und Daten.

- Führen Sie das Skript mit einem Hilfsprogramm des von Ihnen verwendeten DBMS aus.

Eine kurze Beschreibung für die Installation steht bei den Hinweisen zu den DOWNLOADS³. Für Details nutzen Sie bitte die Dokumentation des jeweiligen Datenbankmanagementsystems; Links dazu stehen in der EINLEITUNG⁴ sowie unter WEBLINKS⁵.

Zum Abschluss werden auch ein paar Verfahren behandelt, schnell viele Datensätze als TESTDATEN⁶ zu erzeugen. Für das Verständnis der Inhalte des Buchs sind diese Verfahren nicht wichtig. Sie sind aber äußerst nützlich, wenn Sie weitere Möglichkeiten der SQL-Befehle ausprobieren wollen.

5.4. Anmerkungen

Die Beispieldatenbank wurde erstellt und eingefügt, als sich das Buch noch ziemlich am Anfang befand. Im Laufe der Zeit wurden bei der Konzeption und der Umsetzung in Beispielen Unstimmigkeiten und Mängel festgestellt. Auf diese soll hier hingewiesen werden; teilweise wurden sie beseitigt, teilweise werden wir die Datenstruktur in späteren Kapiteln ändern, teilweise belassen wir es dabei.

Namen von Tabellen und Spalten

Es ist üblicher, eine Tabelle im Singular zu bezeichnen.

- Es ist einfacher zu sagen: Ein *Versicherungsvertrag* hat diese Eigenschaften. . . Umständlicher wäre: Ein Datensatz der Tabelle *Versicherungsverträge* hat diese Eigenschaften. . .
- Vor allem beim Programmieren werden die Namen mit anderen Begriffen verknüpft; es käme dann etwas wie *VertraegePositionenRow* zustande – also die Positionen von Verträgen –, wo nur eine einzelne Zeile, nämlich eine Position eines Vertrags gemeint ist.

• Aus diesem Grund wurden die o. g. SQL-Skripte nochmals vollständig überarbeitet; alle Tabellennamen benutzen jetzt den Singular.

3 Kapitel 37 auf Seite 475

4 Kapitel 2 auf Seite 11

5 Kapitel 41 auf Seite 503

6 Kapitel 35 auf Seite 455

In einem Programmiererforum las der Autor dieser Zeilen eine kleine Diskussion über „Singular oder Plural“. Dort wurde das als uraltes Streitthema bezeichnet, das zu heftigsten Auseinandersetzungen führen kann. Im Ergebnis sollte es eher als Geschmackssache angesehen werden. Also hat der Autor in diesem Buch seinen Geschmack durchgesetzt. Wer mag, kann für sich alle Tabellennamen (im Skript für die Beispieldatenbank und in allen Beispielen und Übungen) ändern.

Die Spalten von Tabellen sollen den Inhalt klar angeben; das ist durchgehend berücksichtigt worden. Es ist nicht üblich, ein Tabellenkürzel als Teil des Spaltennamens zu verwenden; denn ein Spaltenname wird immer in Zusammenhang mit einer bestimmten Tabelle benutzt.

➤ Aus einer früheren Version der Beispieldatenbank wurden diese Kürzel entfernt.

Es ist häufig praktischer, englische Begriffe für Tabellen und Spalten zu verwenden. Programmierer müssen die Namen weiterverarbeiten; weil sowohl Datenbanken als auch Programmiersprachen mit Englisch arbeiten, entstehen ständig komplexe Namen. Dabei ist ein Begriff wie *CustomerNumberChanged* (also reines Englisch) immer noch besser als etwas wie *KundeNummerChanged*, also denglischer Mischmasch.

➤ Auf diese Änderung wird verzichtet, weil es ziemlich hohen Aufwand erfordert hätte. Außerdem spielt die Weiterverwendung durch Programmierer in diesem Buch eigentlich keine Rolle.

Aufteilung der Tabellen

Die Aufgaben der Versicherungsgesellschaft können nur eingeschränkt behandelt werden. Es fehlen Tabellen für Abrechnung und Zahlung der Prämien sowie Abrechnung der Kosten für Schadensfälle.

➤ Auf diese Aufgaben wird hier verzichtet.

Die Tarife und damit die Höhe der Basisprämie gehören in eine eigene Tabelle; ein Versicherungsvertrag müsste darauf verweisen.

➤ Auf diese Aufteilung wird wegen der Vereinfachung verzichtet.

Sowohl Mitarbeiter als auch Versicherungsnehmer sind Personen: Auch zu einem Versicherungsnehmer passen Kontaktdaten; auch bei einem Mitarbeiter

ist die Privatanschrift von Bedeutung. Es wäre deshalb vernünftig, eine gemeinsame Tabelle *Person* einzurichten; die Tabelle *Mitarbeiter* würde Verweise auf diese Tabelle erhalten.

• Auf diese Änderung wird wegen der Vereinfachung verzichtet.

Man kann darüber streiten, ob der optionale Verweis auf eine fremde Versicherungsgesellschaft besser zum Versicherungsvertrag oder zum Versicherungsnehmer gehört.

• Wegen dieser Unklarheit wird auf eine Änderung verzichtet.

Die Verknüpfungen zwischen den Tabellen fehlten in der Ursprungsversion der Beispieldatenbank vollständig. Sie werden in den Kapiteln FREMDSCHLÜSSEL-BEZIEHUNGEN⁷ besprochen und in ÄNDERUNG DER DATENBANKSTRUKTUR⁸ eingebaut.

Fehlende Spalten und Einschränkungen

Neben den bereits genannten Punkten sind **weitere Spalten** sinnvoll bzw. notwendig:

- Tabelle *Versicherungsvertrag*: Basisprämie, Prämienatz, letzte Änderung des Prämienatzes
- Tabelle *Zuordnung_SC_FZ*: Anteil am Verschulden
- Tabellen *Versicherungsnehmer* und *Mitarbeiter*: Geschlecht m/w wegen der Anrede in Briefen

Viele **Einschränkungen** fehlen in der ursprünglichen Version:

- Spalten, deren Werte eindeutig sein müssen
- Spalten, deren Werte nur bestimmte Werte annehmen dürfen
- Tabellen, bei denen Spaltenwerte in gewisser Beziehung stehen müssen

Diese Änderungen werden in DDL - EINZELHEITEN⁹ besprochen und in ÄNDERUNG DER DATENBANKSTRUKTUR¹⁰ eingebaut.

7 Kapitel 28 auf Seite 343

8 Kapitel 34 auf Seite 443

9 Kapitel 27 auf Seite 321

10 Kapitel 34 auf Seite 443

Teil II.

Grundlagen

6. SQL-Befehle

Manchen Einsteigern mag SQL sehr sperrig erscheinen. SQL ist weniger für improvisierte Einmalabfragen gedacht als vielmehr zur Entwicklung von stabilen, dauerhaft nutzbaren Abfragen. Wenn die Abfrage einmal entwickelt ist, wird sie meistens in eine GUI- oder HTML-Umgebung eingebunden, sodass der Benutzer mit dem SQL-Code gar nicht mehr in Berührung kommt.

Es gibt zwar einen SQL-Standard (siehe das Kapitel [EINLEITUNG¹](#)), der möglichst von allen Datenbanken übernommen werden soll, aber leider hält sich kein angebotenes DBMS (Datenbank-Management-System) vollständig daran. Es kommt also je nach DBMS zu leichten bis großen Abweichungen, und für Sie führt kein Weg daran vorbei, immer wieder in der Dokumentation ihrer persönlichen Datenbank nachzulesen. Das gilt natürlich besonders für Verfahren, die im SQL-Standard gar nicht enthalten sind, von Ihrem DBMS trotzdem angeboten werden.



Die Abweichungen eines bestimmten DBMS vom SQL-Standard werden sehr oft als **SQL-Dialekt** bezeichnet.

6.1. Allgemeine Hinweise

Die gesamte Menge an Befehlen ist recht überschaubar; Schwierigkeiten machen die vielen Parameter mit zahlreichen Varianten.

Der Übersicht halber wurde SQL in **Teilbereiche** gegliedert; allerdings gibt es auch für diese Aufteilung Unterschiede bei den DBMS, den Dokumentationen und in Fachbüchern. Diese Aufteilung (siehe Inhaltsverzeichnis dieses Kapitels) dient aber nicht nur der Übersicht, sondern hat auch praktische Gründe:

¹ Kapitel 2 auf Seite 11

- DML-Befehle werden vor allem von „einfachen“ Anwendern benutzt.
- DDL- und TCL-Befehle dienen Programmierern.
- DCL-Befehle gehören zum Aufgabenbereich von Systemadministratoren.

Dies sind die **Bestandteile** eines einzelnen Befehls:

- der Name des Befehls
- der Name des Objekts (Datenbank, Tabelle, Spalte usw.)
- ein Hinweis zur Maßnahme, soweit diese nicht durch den Befehl klar ist
- weitere Einzelheiten
- das Semikolon als Zeichen für den Abschluss eines SQL-Befehls

Damit dies alles eindeutig ist, gibt es eine Reihe von SCHLÜSSELWÖRTERN² (key words, reserved words, non-reserved words), anhand derer das DBMS die Informationen innerhalb eines Befehls erkennt.

Die **Schreibweise** eines Befehls ist flexibel.

- Groß- und Kleinschreibung der Schlüsselwörter werden nicht unterschieden.
- Ein Befehl kann beliebig auf eine oder mehrere Zeilen verteilt werden; der wichtigste Gesichtspunkt dabei ist die Lesbarkeit auch für Sie selbst beim Schreiben.
- Das Semikolon ist nicht immer erforderlich, wird aber empfohlen. Bei manchen DBMS wird der Befehl erst nach einem folgenden GO o. ä. ausgeführt.

Für eigene **Bezeichner**, d. h. die Namen von Tabellen, Spalten oder eigenen Funktionen gilt:

- Vermeiden Sie unbedingt, Schlüsselwörter dafür zu verwenden; dies führt schnell zu Problemen auch dort, wo es möglich wäre.
- Das Wort muss in der Regel mit einem Buchstaben oder dem Unterstrich a . . . z A . . . Z _ beginnen. Danach folgen beliebig Ziffern und Buchstaben.
- Inwieweit andere Zeichen und länderspezifische Buchstaben (Umlaute) möglich sind, hängt vom DBMS ab.

Der Autor dieser Zeilen rät von solchen Zeichen ab, aber das mag Geschmackssache sein. Bei den Erläuterungen zur BEISPIELDATEN-BANK³ (Namen von Tabellen und Spalten) wird sowieso für englische Bezeichnungen plädiert.

2 Kapitel 39 auf Seite 495

3 Kapitel 5 auf Seite 43

Dieses Buch geht davon aus, dass Schlüsselwörter nicht als Bezeichner dienen und auch Umlaute nicht benutzt werden.

Kommentare können in SQL-Befehle fast beliebig eingefügt werden (nur die Schlüsselwörter dürfen natürlich nicht „zerrissen“ werden). Es gibt zwei Arten von Kommentaren:

```
-- (doppelter Bindestrich, am besten mit Leerzeichen dahinter)
```

Alles, was dahinter kommt (einschließlich der beiden Striche) bis zum Ende dieser Zeile gilt als Kommentar und nicht als Bestandteil des Befehls.

```
/* (längerer Text, gerne auch über mehrere Zeilen) */
```

Alles, was zwischen `'/*'` und `'*/'` steht (einschließlich dieser Begrenzungszeichen), gilt als Kommentar und nicht als Bestandteil des Befehls.

6.2. DML – Data Manipulation Language

DML beschäftigt sich mit dem **Inhalt** des Datenbestandes. „Manipulation“ ist dabei nicht nur im Sinne von „Manipulieren“ zu verstehen, sondern allgemeiner im Sinne von „in die Hand nehmen“ (lat. *manus* = Hand).

Der SELECT-Befehl dient der Abfrage von Daten und wird in DML (1) - DATEN ABFRAGEN⁴ behandelt.

- Gelegentlich finden Sie dafür auch den Begriff *Data Query Language (DQL)*. Diese Einführung fasst ihn als Teilgebiet der DML auf; nur aus Gründen der Übersicht gibt es getrennte Kapitel.

Die Befehle INSERT, UPDATE, DELETE dienen der Speicherung von Daten und werden in DML (2) - DATEN SPEICHERN⁵ behandelt.

Bei diesen Befehlen ist immer genau eine Tabelle – nur bei SELECT auch mehrere – anzugeben, dazu Art und Umfang der Arbeiten sowie in aller Regel mit WHERE eine Liste von Bedingungen, welche Datensätze bearbeitet werden sollen.

4 Kapitel 7 auf Seite 63

5 Kapitel 8 auf Seite 75

Die folgenden Erläuterungen sind einheitlich für alle DML-Befehle zu beachten.

6.2.1. Datenmengen, nicht einzelne Datensätze

Bitte beachten Sie, dass SQL grundsätzlich **mengenorientiert** arbeitet. DML-Befehle wirken sich meistens nicht nur auf einen Datensatz aus, sondern auf eine ganze Menge, die aus 0, einem oder mehreren Datensätzen bestehen kann. Auch die WHERE-Bedingungen sorgen „nur“ für den Umfang der Datenmenge; aber das Ergebnis ist immer eine Datenmenge.

Im Einzelfall wissen Sie als Anwender oder Programmierer aufgrund der Bedingungen natürlich häufig, ob die Datenmenge 0, 1 oder n Datensätze enthalten kann oder soll. Aber Sie müssen selbst darauf achten, denn SQL oder das DBMS können das nicht wissen.

Die Struktur als Daten*menge* führt auch dazu, dass es bei einem SELECT-Befehl keine „natürliche“ Reihenfolge gibt, in der die Daten angezeigt werden. Manchmal kommen sie in der Reihenfolge, in der sie gespeichert wurden; aber bei umfangreicheren Tabellen mit vielen Änderungen sieht es eher nach einem großen Durcheinander aus – es sei denn, Sie verwenden die ORDER BY-Klausel.

6.2.2. SQL-Ausdrücke

In vielen Fällen wird der Begriff **Ausdruck** verwendet. Dies ist ein allgemeiner Begriff für verschiedene Situationen:

- An Stellen, an denen ein einzelner Wert angegeben werden muss, kann auch ein Werte-Ausdruck verwendet werden: ein konstanter Wert (Zahl, Text, boolescher Wert), das Ergebnis einer Funktion, die einen solchen Wert zurückgibt, oder eine Abfrage, die als Ergebnis einen einzigen Wert liefert.
- An Stellen, an denen eine oder mehrere Zeilen einer Datenmenge angegeben werden müssen, kann auch ein SQL-Ausdruck (im SQL-Standard als *query expression* bezeichnet) verwendet werden. Dabei handelt es sich um einen SQL-Befehl (in der Regel einen SELECT-Befehl), der als Ergebnis eine Menge von Datensätzen liefert.
- An Stellen, an denen eine Liste einzelner Werte angegeben werden muss, kann ebenfalls ein SQL-Ausdruck verwendet werden; dieser muss dann als Ergebnis eine Menge passender Werte liefern.

6.2.3. Datenintegrität

Bei allen Datenmanipulationen ist zu beachten, dass die Bedingungen für Querverweise erhalten bleiben, siehe das Kapitel FREMDSCHLÜSSEL-BEZIEHUNGEN⁶. Beispielsweise darf ein Datensatz in der Tabelle *Abteilung* erst dann gelöscht werden, wenn alle zugeordneten Mitarbeiter gelöscht oder versetzt wurden.

6.2.4. Hinweis für Programmierer: Parameter benutzen!

Immer dann, wenn ein SQL-Befehl in einem Programm eingebunden wird, sollte er nicht als „langer String“ mit *festen* Zeichenketten erstellt werden, sondern mit Parametern. Das erleichtert die variable Wiederverwendung, die Einbindung unterschiedlichster Werte, umgeht Probleme mit der Formatierung von Zahlen und Datumsangaben und verhindert SQL-Injection.

Im Kapitel DML (2) - DATEN SPEICHERN⁷ steht unter UPDATE folgendes Beispiel:

Dies korrigiert die Schreibweise des Namens beim Mitarbeiter mit der Personalnummer 20001

```
UPDATE Mitarbeiter
  SET Name = 'Mayer'
 WHERE Personalnummer = 20001;
```

Dieses Beispiel sieht besser so aus:

SQL Quelltext

```
UPDATE Mitarbeiter
  SET Name = @Name
 WHERE Personalnummer = @PersNr;
```

In einem Programm mit ADO.NET und Visual Basic für MS-SQL wird dieser Befehl wie folgt verwendet. (Das sieht zwar nach mehr Schreibarbeit aus, ist aber tatsächlich wesentlich einfacher und sicherer.)

6 Kapitel 28 auf Seite 343

7 Kapitel 8 auf Seite 75

Visual Basic Quelltext

```
Dim sel As String = "UPDATE ... @PersNr;"  
Dim cmd As SqlCommand = new SqlCommand(sel)  
cmd.Parameters.AddWithValue("@Name", "Mayer")  
cmd.Parameters.AddWithValue("@PersNr", 20001)
```

Zwar unterscheidet sich die Art, wie die Parameter bezeichnet werden, nach Programmiersprache und DBMS. Aber diese Unterschiede sind viel geringer als die Unterschiede nach den Datentypen und machen die Arbeit viel sicherer.

Parameter gibt es nur für DML-Befehle, nicht für DDL oder DCL.

6.3. DDL – Data Definition Language

DDL **definiert** die Struktur einer Datenbank.

Hierzu gibt es die Befehle CREATE, ALTER, DROP; diese werden für Datenbank-Objekte DATABASE, TABLE, VIEW usw. verwendet.

Einzelheiten werden in DDL - STRUKTUR DER DATENBANK⁸ behandelt.

6.4. TCL – Transaction Control Language

Damit die Daten dauerhaft zusammenpassen, also die Integrität der Daten gewahrt bleibt, sollen Änderungen, die zusammengehören, auch „am Stück“ übertragen und gespeichert werden. Falls eine einzelne dieser Änderungen nicht funktioniert, muss der gesamte Vorgang rückgängig gemacht werden.

Dies wird durch **Transaktionen** gesteuert, die mit COMMIT oder ROLLBACK abgeschlossen werden.

Einzelheiten werden in TCL - ABLAUFSTEUERUNG⁹ behandelt.

8 Kapitel 9 auf Seite 87

9 Kapitel 10 auf Seite 97

6.5. DCL – Data Control Language

Eine „vollwertige“ SQL-Datenbank regelt umfassend die **Rechte für den Zugriff** auf Objekte (Tabellen, einzelne Felder, interne Funktionen usw.). Hierzu kommen die Befehle GRANT und REVOKE zum Einsatz.

Mehr darüber steht in DCL - ZUGRIFFSRECHTE¹⁰.

6.6. Zusammenfassung

In diesem Kapitel lernten wir grundlegende Informationen zu SQL-Befehlen kennen:

- Die Befehle der DML (Data Manipulation Language) bearbeiten die Daten und gehören zum Aufgabenbereich eines jeden Benutzers.
- Mit SELECT werden Daten abgerufen, mit INSERT und UPDATE gespeichert und mit DELETE gelöscht.
- Die DML-Befehle arbeiten mengenorientiert; anstelle konkreter Werte werden häufig Ausdrücke verwendet.
- Die Befehle der DDL (Data Definition Language) steuern die interne Struktur der Datenbank und gehören zum Aufgabenbereich von Programmmentwicklern.
- Die Befehle der TCL (Transaction Control Language) sorgen für die Integrität der Daten beim Speichern und gehören ebenfalls zum Aufgabenbereich von Programmmentwicklern.
- Die Befehle der DCL (Data Control Language) sorgen für die Zugriffssicherheit und gehören zum Aufgabenbereich von Systemadministratoren.

Die Bestandteile der SQL-Befehle werden anhand von Schlüsselwörtern erkannt und ausgewertet.

6.7. Übungen

Übung 1 – Begriffsklärung

Was versteht man unter „SQL“?

10 Kapitel 11 auf Seite 105

1. Süß – Quadratisch – Lecker
2. Server's Quick Library
3. Structured Query Language
4. Standard Quarterly Lectures

Übung 2 – Begriffsklärung

Was versteht man unter „DBMS“?

1. Datenbankmanagementsprache
2. Datenbankmanagementsystem
3. Data-Base Manipulation Standard
4. Deutsche Bahn Mobilstation

Übung 3 – Begriffsklärung

Was versteht man unter „SQL-Dialekt“?

1. die Sprache des Computers
2. die Sprache des Benutzers
3. die Sprache des Programmierers eines DBMS
4. die Abweichungen der SQL-Befehle vom SQL-Standard
5. die jeweilige Version eines DBMS

Übung 4 – Teilbereiche von SQL

Zu welchem der SQL-Teilbereiche DML (Data Manipulation Language), DDL (Data Definition Language), TCL (Transaction Control Language), DCL (Data Control Language) gehören die folgenden Befehle?

1. wähle Daten aus
2. erzeuge eine Tabelle in der Datenbank
3. erzeuge eine Prozedur
4. ändere die Informationen, die zu einem Mitarbeiter gespeichert sind
5. ändere die Definition einer Spalte
6. bestätige eine Gruppe von zusammengehörenden Anweisungen
7. gewähre einem Benutzer Zugriffsrechte auf eine Tabelle
8. `SELECT ID FROM Mitarbeiter`
9. `ROLLBACK`
10. `UPDATE Versicherungsvertrag SET ...`
11. lösche einen Datensatz in einer Tabelle
12. lösche eine Tabelle insgesamt

Übung 5 – SQL-Kommentare

Welche der folgenden Zeilen enthalten korrekte Kommentare? *Mit ... werden weitere Teile angedeutet, die für die Frage unwichtig sind.*

1. SELECT * FROM Mitarbeiter;
2. UPDATE Mitarbeiter SET Name = 'neu'; -- ändert den Namen aller Zeilen
3. DEL/*löschen*/ETE FROM Mitarbeiter WHERE ...
4. -- DELETE FROM Mitarbeiter WHERE ...
5. UPDATE Mitarbeiter /* ändern */ SET Name = ...; /* usw. */

6.8. Lösungen**Lösung zu Übung 1 – Begriffsklärung**

Antwort 3 ist richtig.

Lösung zu Übung 2 – Begriffsklärung

Antwort 2 ist richtig.

Lösung zu Übung 3 – Begriffsklärung

Antwort 4 ist richtig.

Lösung zu Übung 4 – Teilbereiche von SQL

1. DML – 2. DDL – 3. DDL – 4. DML
5. DDL – 6. TCL – 7. DCL – 8. DML
9. TCL – 10. DML – 11. DML – 12. DDL

Lösung zu Übung 5 – SQL-Kommentare

1. Diese Zeile enthält keinen Kommentar.
2. Der letzte Teil nach dem Semikolon ist ein Kommentar.
3. Dieser Versuch eines Kommentars zerstört das Befehlswort DELETE und ist deshalb unzulässig.

4. Die gesamte Zeile zählt als Kommentar.
5. Diese Zeile enthält zwei Kommentare: zum einen zwischen dem Tabellennamen und dem Schlüsselwort SET sowie den Rest der Zeile hinter dem Semikolon.

6.9. Siehe auch

Bei Wikipedia gibt es Erläuterungen zu Fachbegriffen:

- [GRAFISCHE BENUTZEROBERFLÄCHE](#)¹¹ (GUI)
- [HTML](#)¹²
- [PARAMETER](#)¹³ in der Informatik
- [SQL-INJECTION](#)¹⁴
- [.NET](#)¹⁵ und [ADO.NET](#)¹⁶
- [DATENINTEGRITÄT](#)¹⁷

11 [HTTP://DE.WIKIPEDIA.ORG/WIKI/GRAFISCHE%20BENUTZEROBERFL%E4CHE](http://de.wikipedia.org/wiki/GRAFISCHE%20BENUTZEROBERFL%E4CHE)

12 [HTTP://DE.WIKIPEDIA.ORG/WIKI/HYPertext%20MARKUP%20LANGUAGE](http://de.wikipedia.org/wiki/HYPertext%20MARKUP%20LANGUAGE)

13 [HTTP://DE.WIKIPEDIA.ORG/WIKI/PARAMETER%20%28INFORMATIK%29](http://de.wikipedia.org/wiki/PARAMETER%20%28INFORMATIK%29)

14 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SQL-{}INJECTION](http://de.wikipedia.org/wiki/SQL-%7BINJECTION)

15 [HTTP://DE.WIKIPEDIA.ORG/WIKI/.NET](http://de.wikipedia.org/wiki/.NET)

16 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ADO.NET](http://de.wikipedia.org/wiki/ADO.NET)

17 [HTTP://DE.WIKIPEDIA.ORG/WIKI/DATENINTEGRIT%E4T](http://de.wikipedia.org/wiki/DATENINTEGRIT%E4T)

7. DML (1) – Daten abfragen

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.

Dieser Teilbereich der Data Manipulation Language (DML) behandelt den SQL-Befehl **SELECT**, mit dem Abfragen durchgeführt werden.

Dieses Kapitel behandelt einfache Abfragen. Ausführlicher wird der SELECT-Befehl unter „Abfragen für Fortgeschrittene“ behandelt, beginnend mit AUSFÜHRLICHE SELECT-STRUKTUR¹; unten im Abschnitt AUSBLICK AUF KOMPLEXE ABFRAGEN² gibt es Hinweise auf diese weiteren Möglichkeiten.

7.1. SELECT – Allgemeine Hinweise

SELECT ist in der Regel der erste und wichtigste Befehl, den der SQL-Neuling kennenlernt, und das aus gutem Grund: Man kann damit keinen Schaden anrichten. Ein Fehler im Befehl führt höchstens zu einer Fehlermeldung oder dem Ausbleiben des Abfrageergebnisses, aber nicht zu Schäden am Datenbestand. Trotzdem erlaubt der Befehl das Herantasten an die wichtigsten Konzepte von DML, und die anderen Befehle müssen nicht mehr so intensiv erläutert werden.

Dieser Befehl enthält die folgenden Bestandteile („Klauseln“ genannt).

```
SELECT [DISTINCT | ALL]
        <spaltenliste> | *
FROM    <tabelleliste>
[WHERE  <bedingungsliste>]
[GROUP BY <spaltenliste> ]
```

1 Kapitel 14 auf Seite 145

2 [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23AUSBLICK%20AUF%20KOMPLEXE%20ABFRAGEN](http://de.wikibooks.org/wiki/%23Ausblick%20auf%20komplexe%20abfragen)

```
[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.

Die wichtigsten Teile werden in den folgenden Abschnitten erläutert.

Die folgenden Punkte verlangen dagegen vertiefte Beschäftigung mit SQL:

- GROUP BY – Daten gruppieren
- HAVING – weitere Einschränkungen
- UNION – mehrere Abfragen verbinden

Diese Punkte sowie weitere Einzelheiten zu den wichtigsten Bestandteilen werden in AUSFÜHRLICHE SELECT-STRUKTUR³ und anderen „fortgeschrittenen“ Kapiteln behandelt.

Die Beispiele beziehen sich auf den Anfangsbestand der Beispieldatenbank; auf die Ausgabe der selektierten Datensätze wird in der Regel verzichtet. Bitte probieren Sie alle Beispiele aus und nehmen Sie verschiedene Änderungen vor, um die Auswirkungen zu erkennen.

7.2. Die einfachste Abfrage

► **Aufgabe:** Gesucht wird der Inhalt der Tabelle der Fahrzeughersteller mit all ihren Spalten und Datensätzen (Zeilen).

```
SELECT * FROM Fahrzeughersteller;
```

Schauen wir uns das Beispiel etwas genauer an:

- Die beiden Begriffe **SELECT** und **FROM** sind SQL-spezifische Bezeichner.
- *Fahrzeughersteller* ist der Name der Tabelle, aus der die Daten selektiert und ausgegeben werden sollen.

3 Kapitel 14 auf Seite 145

- Das Sternchen, Asterisk genannt, ist eine Kurzfassung für „alle Spalten“.

7.3. Eingrenzen der Spalten

Nun wollen wir nur bestimmte Spalten ausgeben, nämlich eine Liste aller Fahrzeughersteller; 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:** Die Namensliste aller Fahrzeughersteller wird durch folgenden SQL-Befehl erreicht:

```
SELECT Name FROM Fahrzeughersteller;
```

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

```
SELECT Name, Land FROM Fahrzeughersteller;
```

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 Fahrzeughersteller;
```

7.4. DISTINCT – Keine doppelten Zeilen

- **Aufgabe:** Gesucht wird die Liste der Herstellerländer:

```
SELECT Land
FROM Fahrzeughersteller;
```

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 Fahrzeughersteller;
```

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.

Vertiefte Erläuterungen sind unter **NÜTZLICHE ERWEITERUNGEN**⁴ zu finden.

7.5. 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.

4 Kapitel 22 auf Seite 237

► **Aufgabe:** Beispielsweise sollen alle Hersteller angezeigt werden, die ihren Sitz in Schweden oder Frankreich haben:

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

Auf die Klammern kann hier verzichtet werden.

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.

► **Aufgabe:** In einer anderen Abfrage sollen alle Fahrzeughersteller 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 Fahrzeughersteller
WHERE Land <> 'Deutschland';
```

Das Gleichheitszeichen aus der oberen Abfrage wurde durch das Ungleichheitszeichen ersetzt. Dadurch werden jetzt alle Hersteller ausgegeben, deren Sitz ungleich Deutschland ist.

Vertiefte Erläuterungen sind unter [WHERE-KLAUSEL IM DETAIL⁵](#) zu finden.

7.6. 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:** So lassen wir uns die Liste der Hersteller nach dem Namen sortiert ausgeben:

5 Kapitel 16 auf Seite 173

```
SELECT * FROM Fahrzeughersteller
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 Fahrzeughersteller
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*⁶ 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 Fahrzeughersteller
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.

► **Aufgabe:** 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 Kapitel 23 auf Seite 263

7.7. 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.

7.7.1. Traditionell mit 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, 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. Für die Verknüpfung der Tabellen werden folgende Bestandteile benötigt:

- 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. (Der Tabellename bzw. Alias kann sehr oft weggelassen werden; aber schon wegen der Übersichtlichkeit sollte er immer benutzt werden.)
- 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. Alle Tabellen müssen zumindest in-

direkt miteinander verknüpft sein. Falsche Verknüpfungen sind eine häufige Fehlerquelle.

Vertiefte Erläuterungen sind unter **EINFACHE TABELLENVERKNÜPFUNG**⁷ zu finden.

7.7.2. Modern mit 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**⁸ zu finden.

7 Kapitel 18 auf Seite 193

8 Kapitel 19 auf Seite 201

7.8. Ausblick auf komplexe Abfragen

Das folgende Beispiel ist erheblich umfangreicher und geht über „Anfängerbedürfnisse“ deutlich hinaus. Es zeigt aber sehr schön, was alles mit SQL möglich ist:

► **Aufgabe:** Gesucht werden die Daten der Versicherungsnehmer im Jahr 2008, und zwar die Adresse, die Höhe des Gesamtschadens und die Anzahl der Schadensfälle.

```
select vn.Name,
       vn.Vorname,
       vn.Strasse,
       vn.Hausnummer as HNR,
       vn.PLZ,
       vn.Ort,
       SUM(sf.Schadenshoehe) as Gesamtschaden,
       COUNT(sf.ID) as Anzahl
from Versicherungsnehmer vn
join Versicherungsvertrag vv
  on vv.Versicherungsnehmer_ID = vn.ID
join Fahrzeug fz
  on fz.ID = vv.Fahrzeug_ID
join Zuordnung_SF_FZ zu
  on zu.Fahrzeug_ID = fz.ID
join Schadensfall sf
  on sf.ID = zu.Schadensfall_ID
where EXTRACT(YEAR from sf.Datum) = 2008
group by vn.Name, vn.Vorname, vn.Strasse, vn.Hausnummer, vn.PLZ, vn.Ort
order by Gesamtschaden, Anzahl;
```

NAME	VORNAME	STRASSE	HNR	PLZ	ORT	GESAMTSCHADEN	ANZAHL
Heckel Obsthandel GmbH		Gahlener Str.	40	46282			
Dorsten	1.438,75					1	
Antonius	Bernhard	Coesfelder Str.	23	45892			
Gelsenkirchen	1.983,00					1	

Hierbei kommen die Funktionen **SUM** (Summe) und **COUNT** (Anzahl) zum Einsatz. Diese können nur eingesetzt werden, wenn die Datenmenge richtig gruppiert wurde. Deshalb wird mit **GROUP BY** das Datenmaterial nach allen verbliebenen, zur Ausgabe vorgesehenen, Datenfeldern gruppiert.

Vertiefte Erläuterungen sind zu finden unter **FUNKTIONEN**⁹ sowie **GRUPPIERUNGEN**¹⁰.

9 Kapitel 13 auf Seite 121

10 Kapitel 24 auf Seite 271

7.9. Zusammenfassung

In diesem Kapitel lernten wir die Grundlagen eines SELECT-Befehls kennen:

- 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.

7.10. Übungen

Bei den Übungen 2 ff. ist jeweils eine Abfrage zur Tabelle *Abteilung* zu erstellen.

Übung 1 – Pflichtangaben

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

Übung 2 – Alle Angaben

Geben Sie alle Informationen zu allen Abteilungen aus.

Übung 3 – Angaben mit Einschränkung

Geben Sie alle Abteilungen aus, deren Standort *Bochum* ist.

Übung 4 – Angaben mit Einschränkungen

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

Übung 5 – Abfrage mit Sortierung

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

7.11. Lösungen

Lösung zu Übung 1 – Pflichtangaben

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

Lösung zu Übung 2 – Alle Angaben

```
select * from Abteilung;
```

Lösung zu Übung 3 – Angaben mit Einschränkung

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

Lösung zu Übung 4 – Angaben mit Einschränkungen

```
select Bezeichnung from Abteilung  
where Ort = 'Bochum' or Ort = 'Essen';
```

Alternativ ist es auch so möglich:

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

Lösung zu Übung 5 – Abfrage mit Sortierung

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

8. DML (2) – Daten speichern

Dieser Teilbereich der Data Manipulation Language (DML) behandelt die Befehle, mit denen die Inhalte der Datenbank geändert werden: Neuaufnahme, Änderung, Löschung.

Bitte beachten Sie, dass mit den Befehlen **INSERT**, **UPDATE**, **DELETE** (fast) immer nur Daten genau einer Tabelle bearbeitet werden können – anders als beim **SELECT**-Befehl, der Daten mehrerer Tabellen zusammenfassen kann.

8.1. INSERT – Daten einfügen

Der **INSERT**-Befehl dient dem Erstellen von neuen Datensätzen. Es gibt ihn in zwei Versionen – zum einen durch die Angabe einzelner Werte, zum anderen mit Hilfe eines **SELECT**-Befehls.

In beiden Versionen müssen die **DATENTYPEN**¹ der Werte zu den Datentypen der Spalten passen. Man sollte nicht versuchen, einer Spalte, die eine Zahl erwartet, eine Zeichenkette zuzuweisen. Man wird nur selten das Ergebnis erhalten, welches man erwartet. Das Kapitel **FUNKTIONEN**² erläutert im Abschnitt „Konvertierungen“ Möglichkeiten, wie Werte implizit (also automatisch) oder explizit durch **CAST** oder **CONVERT** angepasst werden können.

8.1.1. Einzelne mit VALUES

Wenn ein einzelner Datensatz durch die Angabe seiner Werte gespeichert werden soll, gilt folgende Syntax:

1 Kapitel 12 auf Seite 107

2 Kapitel 13 auf Seite 121

```
INSERT INTO <tabellenname>
    [ ( <spaltenliste> ) ]
VALUES ( <werteliste> )
;
```

Zu diesem Befehl gehören folgende Angaben:

- INSERT als Name des Befehls, INTO als feststehender Begriff
- <Tabellenname> als Name der Tabelle, die diesen Datensatz erhalten soll
- in Klammern () gesetzt eine Liste von Spalten (Feldnamen), denen Werte zugewiesen werden
- der Begriff VALUES als Hinweis darauf, dass einzelne Werte angegeben werden
- in Klammern () gesetzt eine Liste von Werten, die in den entsprechenden Spalten gespeichert werden sollen

Wenn eine Liste von Spalten fehlt, bedeutet das, dass alle Spalten dieser Tabelle in der Reihenfolge der Struktur mit Werten versehen werden müssen.

► **Aufgabe:** So wird (wie im Skript der Beispieldatenbank) ein Eintrag in der Tabelle *Mitarbeiter* gespeichert:

```
INSERT INTO Mitarbeiter
    ( Personalnummer, Name, Vorname,
      Telefon, Email, Raum, Ist_Leiter, Abteilung_ID, Geburtsdatum )
VALUES ( '20002', 'Schmitz', 'Michael',
        '0231/5556187', 'michael.schmitz@unserefirma.de', '212', 'N', 2,
        '1959-08-25' );
```

Wenn Sie diesen Befehl mit der Tabellenstruktur vergleichen, werden Sie feststellen:

- Die Spalte *ID* fehlt. Dieser Wert wird von der Datenbank automatisch vergeben.
- Die Spalte *Mobil* fehlt. In dieser Spalte wird folglich ein NULL-Wert gespeichert.
- Die Reihenfolge der Spalten weicht von der Tabellendefinition ab; das ist also durchaus möglich.

In der Beschreibung der Beispieldatenbank werden sehr viele Spalten als „Pflicht“ festgelegt. Folgender Befehl wird deshalb zurückgewiesen:

SQL Quelltext

Falsch


```
INSERT INTO Mitarbeiter
  ( Personalnummer, Name, Vorname, Ist_Leiter, Abteilung_ID )
VALUES ( '17999', 'Liebich', 'Andrea', 'N', 17);
```

```
validation error for column GEBURTSDATUM, value "**** null ****".
```

Die Spalte *Geburtsdatum* darf laut Definition nicht NULL sein; eine Angabe fehlt in diesem Befehl; das wird als NULL interpretiert, also mit einer Fehlermeldung quittiert.

8.1.2. Mengen mit SELECT

Wenn eine Menge von Datensätzen mit Hilfe eines SELECT-Befehls gespeichert werden soll, gilt folgende Syntax:

```
INSERT INTO <tabellenname>
  [ ( <spaltenliste> ) ]
  SELECT <select-Ausdruck>
  ;
```

Zu diesem Befehl gehören die folgenden Angaben:

- INSERT INTO <Tabellenname> (wie oben)
- in Klammern () gesetzt eine Liste von Spalten (Feldnamen), sofern vorgesehen
- dazu ein vollständiger SELECT-Befehl, mit dem die passenden Inhalte geliefert werden

Da ein SELECT-Befehl auch ohne Bezug auf eine Tabelle nur mit konstanten Werten möglich ist, kann das obige Beispiel auch so formuliert werden:

```
INSERT INTO Mitarbeiter
  ( Personalnummer, Name, Vorname,
    Telefon, Email, Raum, Ist_Leiter, Abteilung_ID, Geburtsdatum )
  select '20002', 'Schmitz', 'Michael',
    '0231/5556187', 'michael.schmitz@unserefirma.de', '212', 'N', 2,
    '1959-08-25'
  /* from rdb%database (bei Firebird) */ ;
  /* from dual          (bei Oracle) */ ;
```

Hinweis: Firebird und Oracle kennen diese Art des SELECT-Befehls nicht; dort ist die als Kommentar jeweils eingefügte FROM-Klausel erforderlich.

Wichtig ist diese Art des INSERT-Befehls, wenn neue Datensätze aus vorhandenen anderen Daten abgeleitet werden wie im Skript der Beispieldatenbank:

► **Aufgabe:** Für jeden Abteilungsleiter aus der Tabelle *Mitarbeiter* wird ein Eintrag in der Tabelle *Dienstwagen* gespeichert:

```
INSERT INTO Dienstwagen
  ( Kennzeichen, Farbe, Fahrzeugtyp_ID, Mitarbeiter_ID )
SELECT 'DO-WB 42' || Abteilung_ID, 'elfenbein', 14, ID
FROM Mitarbeiter
WHERE Ist_Leiter = 'J';
```

Die Spalte *ID* wird automatisch zugewiesen. Alle anderen Spalten erhalten ausdrücklich Werte:

- Farbe und Fahrzeugtyp als Konstante
- dazu natürlich die ID des Mitarbeiters, dem der Dienstwagen zugeordnet wird
- und ein Kfz-Kennzeichen, das aus einem konstanten Teil mit der ID der Abteilung zusammengesetzt wird

Manche Datenbanken erlauben auch die Erstellung von Tabellen aus einem Select-Ausdruck:

Hier MS-SQL:

```
SELECT [ ( <spaltenliste> ) ]
INTO <tabellenname>
FROM <tabellenname>
```

Hier Teradata:

```
CREATE TABLE <tabellennameA> AS
SELECT [ ( <spaltenliste> ) ]
FROM <tabellenname>
```

8.2. UPDATE – Daten ändern

Der **UPDATE**-Befehl dient zum Ändern einer Menge von Datensätzen in einer Tabelle:

```
UPDATE <Tabellenname>  
  SET <Feldänderungen>  
  [WHERE <Bedingungsliste>];
```

Jede Änderung eines Feldes ist so einzutragen:

```
<Feldname> = <Wert> ,
```

Zu diesem Befehl gehören die folgenden Angaben:

- UPDATE als Name des Befehls
- <Tabellenname> als Name der Tabelle, in der diese Datenmenge geändert werden soll
- SET als Anfang der Liste von Änderungen
- <Feldname> als Name der Spalte, die einen neuen Inhalt erhalten soll, dazu das Gleichheitszeichen und der <Wert> als neuer Inhalt
- ein Komma als Hinweis, dass ein weiteres Feld zu ändern ist; vor der WHERE-Klausel oder dem abschließenden Semikolon muss das Komma entfallen
- die WHERE-Klausel mit Bedingungen, welche Datensätze zu ändern sind: einer oder eine bestimmte Menge

Die Struktur der WHERE-Klausel ist identisch mit derjenigen beim SELECT-Befehl. Wenn alle Datensätze geändert werden sollen, kann die WHERE-Bedingung entfallen; aber beachten Sie unbedingt:

Ohne WHERE-Bedingung wird wirklich und unwiderruflich alles geändert.

An den Beispielen ist zu sehen, dass die Änderung aller Datensätze nur selten sinnvoll ist und meistens mit WHERE-Bedingung gearbeitet wird.

Wie beim INSERT-Befehl muss der Datentyp eines Wertes zum Datentyp der Spalte passen.

Beispiele:

► **Aufgabe:** Korrigiere die Schreibweise des Namens beim Mitarbeiter mit der Personalnummer 20001.

```
UPDATE Mitarbeiter
  SET Name = 'Mayer'
  WHERE Personalnummer = 20001;
```

► **Aufgabe:** Ändere nach einer Eingemeindung PLZ und Ortsname für alle betroffenen Adressen.

SQL Quelltext

```
UPDATE Versicherungsnehmer
  SET Ort = 'Leipzig',
      PLZ = '04178'
  WHERE PLZ = '04430';
```

► **Aufgabe:** Erhöhe bei allen Schadensfällen die Schadenshöhe um 10 % (das ist natürlich keine sinnvolle Maßnahme):

```
UPDATE Schadensfall
  SET Schadenshoehe = Schadenshoehe * 1.1;
```

► **Aufgabe:** Berichtige das Geburtsdatum für einen bestimmten Versicherungsnehmer:

```
update Versicherungsnehmer
  set Geburtsdatum = '14.03.1963'
  where Name = 'Zenep' and Geburtsdatum = '13.02.1963';
```

```
0 row(s) affected.
```

Nanu, keine Zeilen wurden geändert? Bei diesem Befehl wurde zur Kontrolle, welcher Datensatz geändert werden sollte, nicht nur der Nachname, sondern auch das bisher notierte Geburtsdatum angegeben – und dieses war falsch.

Daran ist zu sehen, dass der UPDATE-Befehl tatsächlich eine Menge von Datensätzen ändert: je nach WHERE-Klausel null, einen, mehrere oder alle Zeilen der Tabelle.

8.3. DELETE – Daten löschen

Der **DELETE**-Befehl dient zum Löschen einer Menge von Datensätzen in einer Tabelle:

```
DELETE FROM <Tabellenname>
  [ WHERE <Bedingungsliste> ] ;
```

Zu diesem Befehl gehören folgende Angaben:

- DELETE als Name des Befehls, FROM als feststehender Begriff
- <Tabellenname> als Name der Tabelle, aus der diese Datenmenge entfernt werden soll
- die WHERE-Klausel mit Bedingungen, welche Datensätze zu löschen sind: einer oder eine bestimmte Menge

Die Struktur der WHERE-Klausel ist identisch mit derjenigen beim SELECT-Befehl. Wenn alle Datensätze gelöscht werden sollen, kann die WHERE-Bedingung entfallen; aber beachten Sie unbedingt:

Ohne WHERE-Bedingung wird wirklich und unwiderruflich immer alles gelöscht.

Beispiele:

► **Aufgabe:** Der Mitarbeiter mit der Personalnummer 20001 ist ausgeschieden.

```
DELETE FROM Mitarbeiter
  WHERE Personalnummer = 20001;
```

► **Aufgabe:** Die Abteilung 1 wurde ausgelagert, alle Mitarbeiter gehören nicht mehr zum Unternehmen.

```
DELETE FROM Mitarbeiter
  WHERE Abteilung_ID = 1;
```

► **Aufgabe:** Dies leert den gesamten Inhalt der Tabelle, aber die Tabelle selbst bleibt mit ihrer Struktur erhalten.

```
DELETE FROM Schadensfall;
```

Achtung: Dies löscht ohne weitere Rückfrage alle gespeicherten Schadensfälle. Ein solcher Befehl sollte unbedingt nur nach einer vorherigen Datensicherung ausgeführt werden. Auch der Versuch ist „strafbar“ und führt zum sofortigen Datenverlust.

8.4. TRUNCATE – Tabelle leeren

Wenn Sie entgegen den oben genannten Hinweisen wirklich alle Datensätze einer Tabelle löschen wollen, können Sie (soweit vorhanden) anstelle von DELETE den **TRUNCATE**-Befehl benutzen. Damit werden (ohne Verbindung mit WHERE) *immer* alle Datensätze gelöscht; dies geschieht schneller und einfacher, weil auf das interne Änderungsprotokoll der Datenbank verzichtet wird.

```
TRUNCATE TABLE Schadensfall;
```

8.5. Zusammenfassung

In diesem Kapitel lernten wir die SQL-Befehle kennen, mit denen der Datenbestand geändert wird:

- Mit INSERT + VALUES wird ein einzelner Datensatz eingefügt.
- Mit INSERT + SELECT wird eine Menge von Datensätzen mit Hilfe einer Abfrage eingefügt.
- Mit UPDATE wird eine Menge von Datensätzen geändert; die Menge wird durch WHERE festgelegt.
- Mit DELETE wird eine Menge von Datensätzen gelöscht; die Menge wird durch WHERE festgelegt.
- Mit TRUNCATE werden alle Datensätze einer Tabelle gelöscht.

Die WHERE-Bedingungen sind hier besonders wichtig, damit keine falschen Speicherungen erfolgen.

8.6. Übungen

Übung 1 – Daten einzeln einfügen

Welche Angaben werden benötigt, wenn ein einzelner Datensatz in der Datenbank gespeichert werden soll?

Übung 2 – Daten einzeln einfügen

Speichern Sie in der Tabelle *Mitarbeiter* einen neuen Datensatz und lassen Sie alle Spalten und Werte weg, die nicht benötigt werden.

Übung 3 – Daten einfügen

Begründen Sie, warum der Spalte *ID* beim Einfügen in der Regel kein Wert zugewiesen wird.

Übung 4 – Daten einfügen

Begründen Sie, warum die folgenden Befehle nicht ausgeführt werden können.

```
/* Mitarbeiter */
INSERT INTO Mitarbeiter
  ( ID, Personalnummer, Name, Vorname, Geburtsdatum, Ist_Leiter,
    Abteilung_ID )
values ( 4, 'PD-348', 'Çiçek', 'Yasemin', '23.08.1984', 'J', 9 );
/* Dienstwagen 1 */
INSERT INTO Dienstwagen
  ( Kennzeichen, Farbe, Fahrzeugtyp_ID, Mitarbeiter_ID )
values ( 'DO-UF 1234', null, null, null );
/* Dienstwagen 2 */
INSERT INTO Dienstwagen
  ( Kennzeichen, Farbe, Fahrzeugtyp_ID, Mitarbeiter_ID )
values ( 'HAM-AB 1234', 'rot', 7, null );
```

Übung 5 – Daten ändern und löschen

Warum gibt es selten UPDATE- oder DELETE-Befehle ohne eine WHERE-Klausel?

Übung 6 – Daten ändern

Schreiben Sie einen SQL-Befehl für folgende Änderung: Alle Mitarbeiter, die bisher noch keinen Mobil-Anschluss hatten, sollen unter einer einheitlichen Nummer erreichbar sein.

8.7. Lösungen

Lösung zu Übung 1 – Daten einzeln einfügen

- der INSERT-Befehl selbst
- INTO mit Angabe der Tabelle
- bei Bedarf in Klammern die Liste der Spalten, die mit Werten versehen werden
- VALUES zusammen mit (in Klammern) der Liste der zugeordneten Werte

Lösung zu Übung 2 – Daten einzeln einfügen

Beispielsweise so:

```
insert into Mitarbeiter
  ( Personalnummer, Name, Vorname, Geburtsdatum, Ist_Leiter,
    Abteilung_ID )
values ( 'PD-348', 'Çiçek', 'Yasemin', '23.08.1984', 'J', 9 );
```

Lösung zu Übung 3 – Daten einfügen

Dieser soll von der Datenbank automatisch zugewiesen werden; er muss deshalb weggelassen oder mit NULL vorgegeben werden.

Lösung zu Übung 4 – Daten einfügen

- Mitarbeiter: Diese ID ist schon vergeben; sie muss aber eindeutig sein.
- Dienstwagen 1: Der Fahrzeugtyp ist eine Pflicht-Angabe; die *Fahrzeugtyp_ID* darf nicht null sein.
- Dienstwagen 2: Das Kennzeichen ist zu lang; es darf maximal eine Länge von 10 Zeichen haben.

Lösung zu Übung 5 – Daten ändern und löschen

Dies würde die gleiche Änderung bzw. die Löschung für alle Datensätze ausführen; das ist selten sinnvoll bzw. gewünscht.

Lösung zu Übung 6 – Daten ändern

```
update Mitarbeiter
  set Mobil = '(0177) 44 55 66 77'
  where Mobil is null or Mobil = '';
```


9. DDL – Struktur der Datenbank

Mit den Befehlen der Data Definition Language (DDL) wird die Struktur der Datenbank gesteuert.

Dabei handelt es sich um Aufgaben, die über eine Einführung in SQL hinausgehen. Hier werden deshalb nur einige grundlegende Informationen und Beispiele behandelt, für welche Objekte einer Datenbank diese Befehle verwendet werden.

Am Ende des Buches werden einige Erweiterungen behandelt:

- DDL – EINZELHEITEN¹
- ÄNDERUNG DER DATENBANKSTRUKTUR²
- SQL-PROGRAMMIERUNG³

Bitte beachten Sie, dass ein Benutzer vor allem bei DDL-Befehlen über die entsprechenden Rechte verfügen muss, siehe DCL – ZUGRIFFSRECHTE⁴.

9.1. Allgemeine Syntax

Die **DDL-Befehle** sind grundsätzlich so aufgebaut:

```
BEFEHL OBJEKTYP <Objektname> [<weitere Angaben>]
```

9.1.1. CREATE

CREATE erzeugt ein Datenobjekt, zum Beispiel eine Datentabelle oder gar eine Datenbank.

1 Kapitel 27 auf Seite 321

2 Kapitel 34 auf Seite 443

3 Kapitel 29 auf Seite 363

4 Kapitel 11 auf Seite 105

9.1.2. ALTER

Mit ALTER kann das Objekt, z. B. die Tabelle, auch wieder geändert werden:

In neueren Versionen gibt es auch den gemeinsamen Aufruf (unterschiedlich je nach DBMS):

- CREATE OR ALTER
- CREATE OR REPLACE
- RECREATE

Das DBMS entscheidet dann selbst: Wenn das Objekt schon existiert, wird es geändert, andernfalls erzeugt.

9.1.3. DROP

Mit DROP kann das Objekt, z. B. eine Tabelle wieder gelöscht werden:

9.2. Hauptteile der Datenbank

9.2.1. DATABASE – die Datenbank selbst

Der Befehl zum Erstellen einer Datenbank lautet:

```
CREATE DATABASE <Dateiname> [ <Optionen> ] ;
```

Der <Dateiname> ist meistens ein vollständiger Name einschließlich Pfad; in einer solchen Datei werden alle Teile der Datenbank zusammengefasst. Zu den <Optionen> gehören z. B. der Benutzername des Eigentümers der Datenbank mit seinem Passwort, der Zeichensatz mit Angaben zur Standardsortierung, die Aufteilung in eine oder mehrere Dateien usw.

Jedes DBMS bietet sehr verschiedene Optionen; wir können hier keine Gemeinsamkeiten vorstellen und müssen deshalb ganz auf Beispiele verzichten.

Wegen der vielen Möglichkeiten ist zu empfehlen, dass eine Datenbank nicht per SQL-Befehl, sondern innerhalb einer Benutzeroberfläche erstellt wird.

Mit **ALTER DATABASE** werden die Optionen geändert, mit **DROP DATABASE** wird die Datenbank gelöscht. *Diese Befehle kennt nicht jedes DBMS.*

9.2.2. TABLE – eine einzelne Tabelle

CREATE TABLE

Um eine **Tabelle** zu erzeugen, sind wesentlich konkretere umfangreiche Angaben nötig.

```
CREATE TABLE <Tabellenname>
(
  <Spaltenliste>
  [ , <Zusatzangaben> ]
);
```

Zum Erzeugen einer Tabelle werden folgende Angaben benutzt:

- der **Name** der Tabelle, mit dem die Daten über die DML-Befehle gespeichert und abgerufen werden
- die Liste der **Spalten** (Felder), und zwar vor allem mit dem jeweiligen Datentyp
- Angaben wie der **Primärschlüssel** (PRIMARY KEY, PK) oder weitere Indizes

Jede Spalte und Zusatzangabe wird mit einem Komma abgeschlossen; dieses entfällt vor der schließenden Klammer. Die Zusatzangaben werden häufig nicht sofort festgelegt, sondern durch anschließende ALTER TABLE-Befehle; sie werden deshalb weiter unten besprochen.

► **Aufgabe:** In der Beispieldatenbank wird eine Tabelle so erzeugt:

MySQL Quelltext

```
create table Dienstwagen
(
  ID            integer      not null auto_increment primary key,
  Kennzeichen  varchar(30)  not null,
  Farbe        varchar(30),
  Fahrzeugtyp_ID integer    not null,
  Mitarbeiter_ID integer
);
```

Die einzelnen Spalten berücksichtigen mit ihren Festlegungen unterschiedliche Anforderungen:

- ID ist eine ganze Zahl, darf nicht NULL sein, wird automatisch hochgezählt und dient dadurch gleichzeitig als Primärschlüssel.
- Das Kennzeichen ist eine Zeichenkette von variabler Länge (maximal 30 Zeichen), die unbedingt erforderlich ist.

- Die Farbe ist ebenfalls eine Zeichenkette, deren Angabe entfallen kann.
- Für den Fahrzeugtyp wird dessen ID benötigt, wie er in der Tabelle *Fahrzeugtyp* gespeichert ist; diese Angabe muss sein – ein „unbekannter“ Fahrzeugtyp macht bei einem Dienstwagen keinen Sinn.
- Für den Mitarbeiter, dem ein Dienstwagen zugeordnet ist, wird dessen ID aus der Tabelle *Mitarbeiter* benötigt. Dieser Wert kann entfallen, wenn es sich nicht um einen „persönlichen“ Dienstwagen handelt.

ALTER TABLE

Die Struktur einer Tabelle wird wie folgt geändert:

```
ALTER TABLE <Aufgabe> <Zusatzangaben>
```

Mit der Aufgabe **ADD CONSTRAINT** wird eine interne Einschränkung – Constraint genannt – hinzugefügt: ► **Aufgabe:** Ein Primärschlüssel kann auch nachträglich festgelegt werden, z. B. so:

```
Firebird Quelltext
```

```
ALTER TABLE Dienstwagen
  ADD CONSTRAINT Dienstwagen_PK PRIMARY KEY (ID);
```

Die Einschränkung bekommt den Namen *Dienstwagen_PK* und legt fest, dass es sich dabei um den PRIMARY KEY unter Verwendung der Spalte *ID* handelt.

► **Aufgabe:** In der Tabelle *Mitarbeiter* muss auch die *Personalnummer* eindeutig sein (zusätzlich zur *ID*, die als PK sowieso eindeutig ist):

```
ALTER TABLE Mitarbeiter
  ADD CONSTRAINT Mitarbeiter_PersNr UNIQUE (Personalnummer);
```

► **Aufgabe:** In der Tabelle *Zuordnung_SF_FZ* – Verknüpfung zwischen den Schadensfällen und den Fahrzeugen – wird ein Feld für sehr lange Texte eingefügt:

```
alter Table Zuordnung_SF_FZ
  add Beschreibung blob;
```

Mit **ALTER ... DROP Beschreibung** kann dieses Feld auch wieder gelöscht werden.

DROP TABLE

► **Aufgabe:** Damit wird eine Tabelle mit allen Daten gelöscht (diese Tabelle gab es in einer früheren Version der Beispieldatenbank und wird jetzt nicht mehr benutzt):

```
DROP TABLE ZUORD_VNE_SCF;
```

Warnung: Dies löscht die Tabelle einschließlich aller Daten unwiderruflich!

9.2.3. USER – Benutzer

► **Aufgabe:** Auf diese Weise wird ein neuer Benutzer für die Arbeit mit der aktuellen Datenbank registriert.

```
MySQL Quelltext
```

```
CREATE USER Hans_Dampf IDENTIFIED BY 'cH4y37X1P';
```

Dieser Befehl richtet einen neuen Benutzer mit dem Namen *Hans_Dampf* ein, der sich mit dem Passwort 'cH4y37X1P' anmelden muss. – Jedes DBMS kennt eigene Regeln und weitere Optionen für die Zuordnung des Passworts und die Verwendung von Anführungszeichen.

9.3. Ergänzungen zu Tabellen

Weitere Objekte in der Datenbank erleichtern die Arbeit mit Tabellen.

9.3.1. VIEW – besondere Ansichten

Eine VIEW ist eine spezielle Sicht auf eine oder mehrere Tabellen. Für den Anwender sieht es wie eine eigene Tabelle aus; es handelt sich aber „nur“ um eine

fest gespeicherte Abfrage, die immer wieder in der gleichen Form benutzt und ausgeführt wird. Bitte beachten Sie: Nur die Abfrage wird fest gespeichert, nicht das Ergebnis; dieses muss bei jedem neuen Aufruf nach den aktuellen Daten neu erstellt werden.

Einzelheiten werden unter ERSTELLEN VON VIEWS⁵ behandelt.

9.3.2. INDEX – Datenzugriff beschleunigen

Ein Index beschleunigt die Suche nach bestimmten Datensätzen. Um beispielsweise in der Tabelle *Versicherungsnehmer* nach dem Namen „Schulze“ zu suchen, würde es zu lange dauern, wenn das DBMS alle Zeilen durchgehen müsste, bis es auf diesen Namen trafe. Stattdessen wird ein Index angelegt (ähnlich wie in einem Telefon- oder Wörterbuch), sodass schnell alle passenden Datensätze gefunden werden.

So wird ein Index mit der Bezeichnung *Versicherungsnehmer_Name* für die Kombination „Name, Vorname“ angelegt:

```
create index Versicherungsnehmer_Name
on Versicherungsnehmer (Name, Vorname);
```

Es ist dringend zu empfehlen, dass Indizes für alle Spalten bzw. Kombinationen von Spalten angelegt werden, die immer wieder zum Suchen benutzt werden.

Weitere Einzelheiten werden unter DDL – EINZELHEITEN⁶ behandelt.

9.3.3. IDENTITY – auch ein automatischer Zähler

Anstelle von AUTO_INCREMENT verwendet MS-SQL diese Erweiterung für die automatische Nummerierung neuer Datensätze:

MS-SQL Quelltext

```
CREATE TABLE Fahrzeug
  (ID          INTEGER          NOT NULL IDENTITY(1,1),
  Kennzeichen VARCHAR(10)     NOT NULL,
  Farbe       VARCHAR(30),
```

5 Kapitel 26 auf Seite 305

6 Kapitel 27 auf Seite 321


```
Fahrzeugtyp_ID INTEGER NOT NULL,
CONSTRAINT Fahrzeug_PK PRIMARY KEY (ID)
);
```

Der erste Parameter bezeichnet den Startwert, der zweite Parameter die Schrittwerte zum nächsten ID-Wert.

9.3.4. SEQUENCE – Ersatz für automatischen Zähler

Wenn das DBMS für Spalten keine automatische Zählung kennt (Firebird, Oracle), steht dies als Ersatz zur Verfügung.

Firebird Quelltext

```
/* zuerst die Folge definieren */
CREATE SEQUENCE Versicherungsnehmer_ID;
/* dann den Startwert festlegen */
ALTER SEQUENCE Versicherungsnehmer_ID RESTART WITH 1;
/* und im Trigger (s.u.) ähnlich wie eine Funktion benutzen */
NEXT VALUE FOR Versicherungsnehmer_ID
```

Während der automatische Zähler, der durch `AUTO_INCREMENT` eingerichtet wird, genau zu der betreffenden Tabelle gehört, bezieht sich eine „Sequenz“ nicht auf eine einzelne Tabelle, sondern auf die gesamte Datenbank. Es ist ohne weiteres möglich, eine einzige Sequenz `AllMyIDs` zu definieren und die neue ID einer jeden Tabelle daraus abzuleiten. Dies ist durchaus sinnvoll, weil die *ID* als Primärschlüssel sowieso keine inhaltliche Bedeutung haben darf, sondern nur ein fortlaufender Zähler ist. In der Beispieldatenbank benutzen wir getrennte Sequenzen, weil sie für die verschiedenen DBMS „ähnlich“ aussehen soll.

Oracle arbeitet (natürlich) mit anderer Syntax (mit mehr Möglichkeiten) und benutzt dann `NEXTVAL`.

9.4. Programmieren mit SQL

Die Funktionalität einer SQL-Datenbank kann erweitert werden, und zwar auch mit Bestandteilen einer „vollwertigen“ Programmiersprache, z. B. Schleifen und IF-Abfragen.

Dies wird unter PROGRAMMIERUNG⁷ behandelt; dabei gibt es relativ wenig Gemeinsamkeiten zwischen den DBMS.

FUNCTION – Benutzerdefinierte Funktionen

Eigene Funktionen ergänzen die (internen) Skalarfunktionen des DBMS.

PROCEDURE – Gespeicherte Prozeduren

Eine Prozedur – gespeicherte Prozedur, engl. StoredProcedure (SP) – ist vorgesehen für Arbeitsabläufe, die „immer wiederkehrende“ Arbeiten ausführen sollen. Es gibt sie mit und ohne Argumente und Rückgabewerte.

TRIGGER – Ereignisse beim Speichern

Ein Trigger ist ein Arbeitsablauf, der automatisch beim Speichern in einer Tabelle ausgeführt wird. Dies dient Eingabeprüfungen oder zusätzlichen Maßnahmen; beispielsweise holt sich Firebird durch einen Trigger den NEXT VALUE einer SEQUENCE (siehe oben).

TRIGGER werden unterschieden nach INSERT-, UPDATE- oder DELETE-Befehlen und können vor oder nach dem Speichern sowie in einer bestimmten Reihenfolge ausgeführt werden.

9.5. Zusammenfassung

In diesem Kapitel lernten wir die Grundbegriffe einer Datenbankstruktur kennen:

- Die DATABASE selbst sowie TABLE sind die wichtigsten Objekte, ein USER arbeitet damit.
- Eine VIEW ist eine fest gespeicherte Abfrage, die wie eine Tabelle abgerufen wird.
- Ein INDEX beschleunigt den Datenzugriff.

Darüber hinaus wurde auf weitere Möglichkeiten wie SEQUENCE, Funktionen, Prozeduren und Trigger hingewiesen.

7 Kapitel 29 auf Seite 363

9.6. Übungen

Übung 1 – Objekte bearbeiten

Welche der folgenden SQL-Befehle enthalten Fehler?

1. CREATE DATABASE C:\DBFILES\employee.gdb DEFAULT CHARACTER SET UTF8;
2. DROP DATABASE;
3. CREATE TABLE Person (ID PRIMARY KEY, Name VARCHAR(30), Vorname VARCHAR(30));
4. ALTER TABLE ADD UNIQUE KEY (Name);

Übung 2 – Tabelle erstellen

Auf welchen zwei Wegen kann der Primärschlüssel (Primary Key, PK) einer Tabelle festgelegt werden? *Ein weiterer Weg ist ebenfalls üblich, aber noch nicht erwähnt worden.*

Übung 3 – Tabelle ändern

Skizzieren Sie einen Befehl, mit dem die Tabelle *Mitarbeiter* um Felder für die Anschrift erweitert werden kann.

Übung 4 – Tabellen

Worin besteht der wesentliche Unterschied zwischen einer TABLE und einer VIEW?

9.7. Lösungen

Lösung zu Übung 1 – Objekte bearbeiten

Bei 2. fehlt der Name der Datenbank.

Bei 3. fehlt der Datentyp zur Spalte *ID*.

Bei 4. fehlt der Name der Tabelle, die geändert werden soll.

Lösung zu Übung 2 – Tabelle erstellen

1. Im CREATE TABLE-Befehl zusammen mit der Spalte, die als PK benutzt wird, z. B.:

```
ID INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

2. Durch einen ALTER TABLE-Befehl, der den PK hinzufügt:

```
ALTER TABLE Dienstwagen ADD PRIMARY KEY (ID);
```

Lösung zu Übung 3 – Tabelle ändern

```
alter table Mitarbeiter
add PLZ CHAR(5),
add Ort VARCHAR(24),
add Strasse VARCHAR(24),
add Hausnummer VARCHAR(10);
```

Lösung zu Übung 4 – Tabellen

Eine TABLE (Tabelle) ist real in der Datenbank gespeichert. Eine VIEW (Sichtta-
belle) ist eine Sicht auf eine oder mehrere tatsächlich vorhandene Tabellen; sie
enthält eine Abfrage auf diese Tabellen, die als Abfrage in der Datenbank gespe-
ichert ist, aber für den Anwender wie eine eigene Tabelle aussieht.

9.8. Siehe auch

Bei diesem Kapitel sind die folgenden Erläuterungen zu beachten:

- DATENTYPEN⁸

Manche Themen werden in den folgenden Kapiteln genauer behandelt:

- EIGENE FUNKTIONEN⁹ als Ergänzung zu denen, die in FUNKTIONEN¹⁰ und FUNKTIONEN (2)¹¹ behandelt werden.
- PROZEDUREN¹²
- TRIGGER¹³

8 Kapitel 12 auf Seite 107

9 Kapitel 30 auf Seite 389

10 Kapitel 13 auf Seite 121

11 Kapitel 15 auf Seite 157

12 Kapitel 31 auf Seite 399

13 Kapitel 32 auf Seite 421

10. TCL – Ablaufsteuerung

Dieses Kapitel gibt eine kurze Einführung in die Transaction Control Language (TCL). Deren Befehle sorgen für die Datensicherheit innerhalb einer Datenbank.

MySQL verfolgt eine etwas „offenere“ Philosophie und arbeitet anstelle von Transaktionen auch mit anderen Sicherungsmaßnahmen. Der Ersteller einer Datenbank muss sich für ein Verfahren entscheiden, kann aber auch danach noch variieren.

10.1. Beispiele

Eine SQL-Datenbank speichert Daten in der Regel in verschiedenen Tabellen. Dabei ist es nötig, dass die betreffenden Befehle immer „gleichzeitig“ ausgeführt werden. Da das nicht möglich ist – zwei Befehle können nur nacheinander erledigt werden –, muss sichergestellt sein, dass nicht der eine Befehl ausgeführt wird, während der andere Befehl scheitert.

- Zu einer Überweisung bei der Bank gehören immer zwei Buchungen: die Gutschrift auf dem einen und die Lastschrift auf dem anderen Konto; häufig gehören die Konten zu verschiedenen Banken. Es wäre völlig unerträglich, wenn die Gutschrift ausgeführt würde und die (externe) Lastschrift nicht, weil in diesem Moment die Datenleitung unterbrochen wird.
- Wenn in dem Versicherungsunternehmen der Beispieldatenbank ein neuer Vertrag abgeschlossen wird, gehören dazu mehrere INSERT-Befehle, und zwar in die Tabellen *Fahrzeug*, *Versicherungsnehmer*, *Versicherungsvertrag*. Zuerst müssen Fahrzeug und Versicherungsnehmer gespeichert werden; aber wenn das Speichern des Vertrags „schiefgeht“, hängen die beiden anderen Datensätze nutzlos in der Datenbank herum.
- Wenn dort eine Abteilung ausgelagert wird, werden alle ihre Mitarbeiter gestrichen, weil sie nicht mehr zum Unternehmen gehören. Wie soll verfahren werden, wenn nur ein Teil der DELETE-Befehle erfolgreich war?

- Eine Menge einzelner Befehle (z. B. 1000 INSERTs innerhalb einer Schleife) dauert „ewig lange“.

Solche Probleme können nicht nur durch die Hardware entstehen, sondern auch dadurch, dass parallel andere Nutzer denselben Datenbestand ändern wollen.

10.2. Transaktionen

Alle solche Ungereimtheiten werden vermieden, indem SQL-Befehle in **Transaktionen** zusammengefasst und ausgeführt werden:

- Entweder alle Befehle können ausgeführt werden. Dann wird die Transaktion bestätigt und erfolgreich abgeschlossen.
- Oder (mindestens) ein Befehl kann nicht ausgeführt werden. Dann wird die Transaktion für ungültig erklärt; alle Befehle werden rückgängig gemacht.
- Auch das Problem mit der langen Arbeitszeit von 1000 INSERTs wird vermieden, wenn das DBMS nicht jeden Befehl einzeln prüft und bestätigt, sondern erst alle 1000 am Schluss „am Stück“.

Es gibt verschiedene Arten von Transaktionen. Diese hängen vom DBMS und dessen Version, der Hardware (Einzel- oder Mehrplatzsystem) und dem Datenzugriff (direkt oder über Anwenderprogramme) ab.

- Wenn die Datenbank auf AUTOCOMMIT eingestellt ist, wird jeder SQL-Befehl als einzelne Transaktion behandelt und sofort gültig. (Das wäre die Situation mit der langen Arbeitszeit von 1000 INSERTs.)
- Wenn ein Stapel von Befehlen mit COMMIT bestätigt oder mit ROLLBACK verworfen wird, dann wird mit dem nächsten Befehl implizit eine neue Transaktion begonnen.
- Mit einem ausdrücklichen TRANSACTION-Befehl wird explizit eine neue Transaktion begonnen:

```
BEGIN TRANSACTION <Name>; /* bei MS-SQL */
SET TRANSACTION <Name>; /* bei Firebird */
START TRANSACTION; /* bei My-SQL */
```

Eine Transaktion kann mit einem Namen versehen werden. Dies ist vor allem dann nützlich, wenn Transaktionen geschachtelt werden. Außerdem gibt es je nach DBMS noch viele weitere Optionen, mit denen eine Transaktion detailliert gesteuert werden kann.

- Eine Transaktion, die implizit oder explizit begonnen wird, ist ausdrücklich abzuschließen durch COMMIT oder ROLLBACK. Wenn dies vergessen wird, wird die Transaktion erst dadurch beendet, dass die Verbindung mit der Datenbank geschlossen wird.

10.3. Transaktion erfolgreich beenden

Eine Transaktion wird mit einem der folgenden Befehle erfolgreich abgeschlossen:

```
COMMIT [ TRANSACTION | WORK ] <Name>;
```

Die genaue Schreibweise und Varianten müssen in der DBMS-Dokumentation nachgelesen werden.

Dieser Befehl bestätigt alle vorangegangenen Befehle einer Transaktion und sorgt dafür, dass sie „am Stück“ gespeichert werden.

► **Aufgabe:** Auszug aus dem Skript zum Erstellen der Beispieldatenbank:

Firebird Quelltext

```
COMMIT;
/* damit wird die vorherige Transaktion abgeschlossen und implizit eine
neue Transaktion gestartet */

INSERT INTO Dienstwagen (Kennzeichen, Farbe, Fahrzeugtyp_ID,
Mitarbeiter_ID)
VALUES ('DO-WB 111', 'elfenbein', 16, NULL);
INSERT INTO Dienstwagen (Kennzeichen, Farbe, Fahrzeugtyp_ID,
Mitarbeiter_ID)
SELECT 'DO-WB 3' || Abteilung_ID || SUBSTRING(Personalnummer FROM 5
FOR 1),
'gelb', SUBSTRING(Personalnummer FROM 5 FOR 1), ID
FROM Mitarbeiter
WHERE Abteilung_ID IN (5, 8)
AND Ist_Leiter = 'N';
/* damit wird diese Transaktion abgeschlossen */
COMMIT;
```

10.4. Sicherungspunkte

Mit dem folgenden Befehl kann eine Transaktion in „sichere“ Abschnitte geteilt werden:

```
SAVEPOINT <Name>;
```

Bis zu diesem Sicherungspunkt werden die Befehle auch dann als gültig abgeschlossen, wenn die Transaktion am Ende für ungültig erklärt wird.

10.5. Transaktion rückgängig machen

Eine Transaktion wird wie folgt für ungültig erklärt:

```
ROLLBACK [ TRANSACTION | WORK ] <Name> [ TO <Sicherungspunkt> ] ;
```

Damit werden alle Befehle der Transaktion für ungültig erklärt und rückgängig gemacht. Sofern ein Sicherungspunkt angegeben ist, werden die Befehle bis zu diesem Sicherungspunkt für gültig erklärt und erst alle folgenden für ungültig.

Die genaue Schreibweise und Varianten müssen in der DBMS-Dokumentation nachgelesen werden.

► **Aufgabe:** Mit dem folgenden Beispiel werden zu Testzwecken einige Daten geändert und abgerufen; abschließend werden die Änderungen rückgängig gemacht.

```
update Dienstwagen
  set Farbe = 'goldgelb/violett gestreift'
  where ID >= 14;
select * from Dienstwagen;
ROLLBACK;
```

10.6. Zusammenfassung

In diesem Kapitel lernten Sie die Grundbegriffe von Transaktionen kennen:

- Eine Transaktion wird implizit oder explizit begonnen.

- Eine Transaktion wird mit einem ausdrücklichen Befehl (oder durch Ende der Verbindung) abgeschlossen.
- Mit COMMIT wird eine Transaktion erfolgreich abgeschlossen; die Daten werden abschließend gespeichert.
- Mit ROLLBACK werden die Änderungen verworfen, ggf. ab einem bestimmten SAVEPOINT.

10.7. Übungen

Übung 1 – Zusammengehörende Befehle

Skizzieren Sie die Befehle, die gemeinsam ausgeführt werden müssen, wenn ein neues Fahrzeug mit einem noch nicht registrierten Fahrzeugtyp und Fahrzeughersteller gespeichert werden soll.

Übung 2 – Zusammengehörende Befehle

Skizzieren Sie die Befehle, die gemeinsam ausgeführt werden müssen, wenn ein neuer Schadensfall ohne weitere beteiligte Fahrzeuge registriert werden soll.

Übung 3 – Zusammengehörende Befehle

Skizzieren Sie die Befehle, die gemeinsam ausgeführt werden müssen, wenn ein neuer Schaden durch einen „Eigenen Kunden“ gemeldet wird; dabei sollen ein zweiter „Eigener Kunde“ sowie ein „Fremdkunde“ einer bisher nicht gespeicherten Versicherungsgesellschaft beteiligt sein. Erwähnen Sie dabei auch den Inhalt des betreffenden Eintrags.

Übung 4 – Transaktionen

Welche der folgenden Maßnahmen starten immer eine neue Transaktion, welche unter Umständen, welche sind unzulässig?

1. das Herstellen der Verbindung zur Datenbank
2. der Befehl `START TRANSACTION;`
3. der Befehl `SET TRANSACTION ACTIVE;`
4. der Befehl `SAVEPOINT <name>` wird ausgeführt
5. der Befehl `ROLLBACK` wird ausgeführt

Übung 5 – Transaktionen

Welche der folgenden Maßnahmen beenden immer eine Transaktion, welche unter Umständen, welche sind unzulässig?

1. das Schließen der Verbindung zur Datenbank
2. der Befehl `END TRANSACTION;`
3. der Befehl `SET TRANSACTION INACTIVE;`
4. der Befehl `SAVEPOINT <name>` wird ausgeführt
5. der Befehl `ROLLBACK` wird ausgeführt

10.8. Lösungen

Lösung zu Übung 1 – Zusammengehörende Befehle

- `INSERT INTO Fahrzeughersteller`
- `INSERT INTO Fahrzeugtyp`
- `INSERT INTO Fahrzeug`

Lösung zu Übung 2 – Zusammengehörende Befehle

- `INSERT INTO Schadensfall`
- `INSERT INTO Zuordnung_SF_FZ`

Lösung zu Übung 3 – Zusammengehörende Befehle

- `INSERT INTO Schadensfall`
- `INSERT INTO Versicherungsgesellschaft /* für den zusätzlichen Fremdkunden */`
- `INSERT INTO Versicherungsnehmer /* für den zusätzlichen Fremdkunden */`
- `INSERT INTO Fahrzeug /* für den zusätzlichen Fremdkunden */`
- `INSERT INTO Zuordnung_SF_FZ /* für den zusätzlichen Fremdkunden */`
- `INSERT INTO Zuordnung_SF_FZ /* für den beteiligten Eigenen Kunden */`
- `INSERT INTO Zuordnung_SF_FZ /* für den Eigenen Kunden laut Schadensmeldung */`

Lösung zu Übung 4 – Transaktionen

1. ja, sofern `AUTOCOMMIT` festgelegt ist
2. ja, aber nur, wenn das DBMS diese Variante vorsieht (wie MySQL)
3. ja, aber nur, wenn das DBMS diese Variante vorsieht (wie Firebird); denn das Wort „ACTIVE“ wird nicht als Schlüsselwort, sondern als Name der Transaction interpretiert
4. nein, das ist nur ein Sicherungspunkt *innerhalb* einer Transaktion
5. ja, nämlich implizit für die folgenden Befehle

Lösung zu Übung 5 – Transaktionen

1. ja, und zwar immer
2. nein, diesen Befehl gibt es nicht
3. nein, dieser Befehl ist unzulässig; wenn das DBMS diese Variante kennt (wie Firebird), dann ist es der Start einer Transaktion namens „INACTIVE“
4. teilweise, das ist ein Sicherungspunkt *innerhalb* einer Transaktion und bestätigt die bisherigen Befehle, setzen aber dieselbe Transaktion fort
5. ja, nämlich explizit durch Widerruf aller Befehle

11. DCL – Zugriffsrechte

Eine „vollwertige“ SQL-Datenbank enthält umfassende Regelungen über die Vergabe von Rechten für den Zugriff auf Objekte (Tabellen, einzelne Felder, interne Funktionen usw.). Am Anfang stehen diese Rechte nur dem Ersteller der Datenbank und dem System-Administrator zu. Andere Benutzer müssen ausdrücklich zu einzelnen Handlungen ermächtigt werden.

Da es sich dabei nicht um Maßnahmen für Einsteiger handelt, beschränken wir uns auf ein paar Beispiele.

11.1. GRANT – Zugriff gewähren

► **Aufgabe:** Der Benutzer Herr_Mueller darf Abfragen auf die Tabelle *Abteilungen* ausführen.

```
GRANT SELECT ON Abteilung TO Herr_Mueller
```

► **Aufgabe:** Die Benutzerin Frau_Schulze darf Daten in der Tabelle *Abteilungen* ändern.

```
GRANT UPDATE ON Abteilung TO Frau_Schulze
```

11.2. REVOKE – Zugriff verweigern

► **Aufgabe:** Herr_Mueller darf künftig keine solche Abfragen mehr ausführen.

```
REVOKE SELECT ON Abteilung FROM Herr_Mueller
```


12. Datentypen

SQL kennt verschiedene Arten von Datentypen: vordefinierte, konstruierte und benutzerdefinierte. Diese Arten und ihre Verwendungen werden in diesem Abschnitt erklärt.

12.1. Vordefinierte Datentypen

Laut SQL-Standard sind die folgenden Datentypen vordefiniert. Die fettgedruckten Begriffe sind die entsprechenden reservierten **SCHLÜSSELWÖRTER**¹.

Achtung

Bei allen Datentypen weichen die SQL-Dialekte mehr oder weniger vom Standard ab.

Dies betrifft vor allem die folgenden Punkte:

- bei den möglichen Werten, z. B. dem maximalen Wert einer ganzen Zahl oder der Länge einer Zeichenkette
- bei der Bearbeitung einzelner Typen; z. B. kennt Firebird noch nicht sehr lange den **BOOLEAN**-Typ und musste mit dem Ersatz „ganze Zahl gleich 1“ leben
- bei der Art der Werte, z. B. ob Datum und Zeit getrennt oder gemeinsam gespeichert sind

12.1.1. Zeichen und Zeichenketten

Es gibt Zeichenketten mit fester, variabler und sehr großer Länge.

- **CHARACTER(n)**, **CHAR(n)**

¹ Kapitel 39 auf Seite 495

Hierbei handelt es sich um Zeichenketten mit fester Länge von genau n Zeichen. Für ein einzelnes Zeichen muss die Länge (1) nicht angegeben werden.

- **CHARACTER VARYING(n), VARCHAR(n)**

Hierbei handelt es sich um Zeichenketten mit variabler Länge mit maximal n Zeichen.

- **CHARACTER LARGE OBJECT**

Hierbei handelt es sich um beliebig große Zeichenketten. Diese Variante ist relativ umständlich zu nutzen; sie wird vorwiegend für die Speicherung ganzer Textdateien verwendet.

Zu allen diesen Varianten gibt es auch die Festlegung eines nationalen Zeichensatzes durch **NATIONAL CHARACTER** bzw. **NCHAR** oder **NATIONAL CHARACTER VARYING** bzw. **NVARCHAR**. Erläuterungen dazu siehe unten im Abschnitt über Zeichensätze.

Die maximale Länge von festen und variablen Zeichenketten hängt vom DBMS ab. In früheren Versionen betrug sie oft nur 255, heute sind 32767 ein verbreiteter Wert.

Die maximale Feldlänge bei Zeichenketten ist für Eingabe, Ausgabe und interne Speicherung wichtig. Die DBMS verhalten sich unterschiedlich, ob am Anfang oder Ende stehende Leerzeichen gespeichert oder entfernt werden. Wichtig ist, dass bei fester Feldlänge ein gelesener Wert immer mit genau dieser Anzahl von Zeichen zurückgegeben wird und bei Bedarf rechts mit Leerzeichen aufgefüllt wird.

In der Praxis sind folgende Gesichtspunkte von Bedeutung:

- Für **indizierte Felder** (also Spalten, denen ein Index zugeordnet wird) sind feste Längen vorzuziehen; beachten Sie aber die nächsten Hinweise.
- Als **CHAR(n)**, also mit fester Länge, sind Felder vorzusehen, deren Länge bei der überwiegenden Zahl der Datensätze konstant ist. Beispiel: die deutschen Postleitzahlen mit CHAR(5).
- Als **VARCHAR(n)**, also mit variabler Länge, sind Felder vorzusehen, deren Länge stark variiert. Beispiel: Namen und Vornamen mit VARCHAR(30).
- In **Zweifelsfällen** ist pragmatisch vorzugehen. Beispiel: Die internationalen Postleitzahlen (Post-Code, Zip-Code) benötigen bis zu 10 Zeichen. Wenn eine Datenbank überwiegend nur deutsche Adressen enthält, passt VARCHAR(10) besser, bei hohem Anteil von britischen, US-amerikanischen, kanadischen und ähnlichen Adressen ist CHAR(10) zu empfehlen.

12.1.2. Zahlen mit exakter Größe

Werte dieser Datentypen werden mit der genauen Größe gespeichert.

- **INTEGER** bzw. **INT**

Ganze Zahl mit Vorzeichen. Der Größenbereich hängt von der Implementierung ab; auf einem 32-bit-System entspricht es meistens $\pm 2^{31}-1$, genauer von $-2\,147\,483\,648$ bis $+2\,147\,483\,647$.

- **SMALLINT**

Ebenfalls ein Datentyp für ganze Zahlen, aber mit kleinerem Wertebereich als **INTEGER**, oft von $-32\,768$ bis $+32\,767$.

- **BIGINT**

Ebenfalls ein Datentyp für ganze Zahlen, aber mit größerem Wertebereich als **INTEGER**. Auch der SQL-Standard akzeptiert, dass ein DBMS diesen Typ nicht kennt.

- **NUMERIC(p,s)** sowie **DECIMAL(p,s)**

Datentypen für Dezimalzahlen mit exakter Speicherung, also „Festkommazahlen“, wobei p die Genauigkeit und s die Anzahl der Nachkommastellen angibt. Dabei muss $0 \leq s \leq p$ sein, und s hat einen Defaultwert von 0. Der Parameter $s=0$ kann entfallen; der Vorgabewert, falls p nicht angegeben wird, hängt vom DBMS ab.

Diese Dezimalzahlen sind wegen der genauen Speicherung z. B. für Daten der Buchhaltung geeignet. Bei vielen DBMS gibt es keinen Unterschied zwischen **NUMERIC** und **DECIMAL**.

12.1.3. Zahlen mit „näherungsweise“ Größe

Werte dieser Datentypen werden nicht unbedingt mit der genauen Größe gespeichert, sondern in vielen Fällen nur näherungsweise.

- **FLOAT, REAL, DOUBLE PRECISION**

Diese Datentypen haben grundsätzlich die gleiche Bedeutung. Je nach DBMS kann durch **FLOAT(p,s)** die Genauigkeit oder die Anzahl der Dezimalstellen angegeben werden; auch der Wertebereich und die Genauigkeit hängen vom DBMS ab.

Diese „Gleitkommazahlen“ sind für technisch-wissenschaftliche Werte geeignet und umfassen auch die Exponentialdarstellung. Wegen der Speicherung im Binärformat sind sie aber für Geldbeträge nicht geeignet, weil sich beispielsweise der Wert 0,10 € (entspricht 10 Cent) nicht exakt abbilden lässt. Es kommt immer wieder zu Rundungsfehlern.

12.1.4. Zeitpunkte und Zeitintervalle

Für Datum und Uhrzeit gibt es die folgenden Datentypen:

- **DATE**

Das Datum enthält die Bestandteile **YEAR, MONTH, DAY**, wobei die Monate innerhalb eines Jahres und die Tage innerhalb eines Monats gemeint sind.

- **TIME**

Die Uhrzeit enthält die Bestandteile **HOURL, MINUTE, SECOND**, wobei die Minute innerhalb einer Stunde und die Sekunden innerhalb einer Minute gemeint sind. Sehr oft werden auch Millisekunden als Bruchteile von Sekunden registriert.

- **TIMESTAMP**

Der „Zeitstempel“ enthält Datum und Uhrzeit zusammen.

Zeitangaben können **WITH TIME ZONE** oder **WITHOUT TIME ZONE** deklariert werden. Ohne die Zeitzone ist in der Regel die lokale Zeit gemeint, mit der Zeitzone wird die Koordinierte Weltzeit (UTC) gespeichert.

Bei Datum und Uhrzeit enden die Gemeinsamkeiten der SQL-Dialekte endgültig; sie werden unterschiedlich mit „eigenen“ Datentypen realisiert. Man kann allenfalls annehmen, dass ein Tag intern mit einer ganzen Zahl und ein Zeitwert mit einem Bruchteil einer ganzen Zahl gespeichert wird.

Beispiele:

Datenbanksystem	Datentyp	Geltungsbereich	Genauigkeit
MS-SQL Server 2005	datetime	01.01.1753 bis 31.12.9999	3,33 Millisekunden
	smalldatetime	01.01.1900 bis 06.06.2079	1 Minute
MS-SQL Server 2008	date	01.01.0001 bis 31.12.9999	1 Tag

Firebird	time	00:00:00.0000000 bis 23:59:59.9999999	100 Nanosekunden
	datetime	01.01.0001 bis 31.12.9999	3,33 Mil- lisekunden
	smalldatetime	01.01.1900 bis 06.06.2079	1 Minute
	DATE	01.01.0100 bis 29.02.32768	1 Tag
MySQL 5.x	TIME	00:00 bis 23:59.9999	6,67 Mil- lisekunden
	DATETIME	01.01.1000 00:00:00 bis 31.12.9999 23:59:59	1 Sekunde
	DATE	01.01.1000 bis 31.12.9999	1 Tag
	TIME	-838:59:59 bis 838:59:59	1 Sekunde
	YEAR	1901 bis 2055	1 Jahr

Bitte wundern Sie sich nicht: bei jedem DBMS gibt es noch weitere Datentypen und Bezeichnungen.

Die Deklaration von TIME bei MySQL zeigt schon: Es muss sich dabei nicht um eine Uhrzeit innerhalb eines Datums handeln, sondern kann auch einen Zeitraum, d. h. ein Intervall darstellen.

- **INTERVAL**

Ein Intervall setzt sich – je nach betrachteter Zeitdauer – zusammen aus:

- **YEAR, MONTH** für längere Zeiträume (der SQL-Standard kennt auch nur die Bezeichnung "year-month-interval")

- **DAY, HOUR, MINUTE, SECOND** für Zeiträume innerhalb eines Tages oder über mehrere Tage hinweg

12.1.5. Große Objekte

BLOB (Binary Large Object, binäre große Objekte) ist die allgemeine Bezeichnung für unbestimmt große Objekte.

- **BLOB**

Allgemein werden binäre Objekte z. B. für Bilder oder Bilddateien verwendet, nämlich dann, wenn der Inhalt nicht näher strukturiert ist und auch Bytes enthalten kann, die keine Zeichen sind.

- **CLOB**

Speziell werden solche Objekte, die nur „echte“ Zeichen enthalten, zum Speichern von großen Texten oder Textdateien verwendet.

Je nach DBMS werden BLOB-Varianten durch *Sub_Type* oder spezielle Datentypen für unterschiedliche Maximalgrößen oder Verwendung gekennzeichnet.

12.1.6. Boolean

Der Datentyp **BOOLEAN** ist für logische Werte vorgesehen. Solche Felder können die Werte TRUE (wahr) und FALSE (falsch) annehmen; auch NULL ist möglich und wird als UNKNOWN (unbekannt) interpretiert.

Wenn ein DBMS diesen Datentyp (noch) nicht kennt – wie MySQL –, dann ist mit einem der numerischen Typen eine einfache Ersatzlösung möglich (wie früher bei Interbase und Firebird); siehe unten im Abschnitt über Domains.

12.2. Konstruierte und benutzerdefinierte Datentypen

Diese Datentypen, die aus den vordefinierten Datentypen zusammengesetzt werden, werden hier nur der Vollständigkeit halber erwähnt; sie sind in der Praxis eines Anwenders ziemlich unwichtig.

- **ROW**

Eine Zeile ist eine Sammlung von Feldern; jedes Feld besteht aus dem Namen und dem Datentyp. Nun ja, eine Zeile in einer Tabelle ist (natürlich) von diesem Typ.

- **REF**

Referenztypen sind zwar im SQL-Standard vorgesehen, treten aber in der Praxis nicht auf.

- **ARRAY, MULTISSET**

Felder und Mengen sind Typen von Ansammlungen ("collection type"), in denen jedes Element vom gleichen Datentyp ist. In einem Array ist jedes Element

durch die Position gekennzeichnet, ein Multiset ist eine ungeordnete Menge. Wegen der Notwendigkeit, Tabellen zu normalisieren, spielen diese Typen in der Praxis keine Rolle.

- **Benutzerdefinierte Typen**

Dazu gehören nicht nur ein Typ, sondern auch Methoden zu ihrer Verwendung. Auch für solche Typen sind keine sinnvollen Anwendungen zu finden.

12.3. Spezialisierte Datentypen

Laut SQL-Standard können Datentypen mit Einschränkungen, also **CONSTRAINTS**, versehen werden; auch der Begriff **Domain** wird verwendet. (Einen vernünftigen deutschen Begriff gibt es dafür nicht.) Viele Vorgaben gibt es im SQL-Standard nicht.

Der Befehl zum Erstellen einer Domain sieht allgemein so aus:

```
CREATE DOMAIN <Domain-Name> <zugehöriger Datentyp> [<Vorgabewert>]
[<Einschränkungen>]
```

Unter Interbase/Firebird wurde auf diese Weise ein Ersatz für den BOOLEAN-Datentyp erzeugt:

```
Firebird Quelltext

CREATE DOMAIN BOOLEAN
  -- definiere diesen Datentyp
  AS INT
  -- als Integer
  DEFAULT 0      NOT NULL
  -- Vorgabewert 0, hier ohne NULL-Werte
  CHECK (VALUE BETWEEN 0 AND 1);
  -- Werte können nur 0 (= false) und 1 (= true) sein
```

Bei MySQL können Spalten mit **ENUM** oder **SET** auf bestimmte Werte eingeschränkt werden, allerdings nur auf Zeichen.

12.4. Nationale und internationale Zeichensätze

Aus der Frühzeit der EDV ist das Problem der nationalen Zeichen geblieben: Mit 1 Byte (= 8 Bit) können höchstens 256 Zeichen (abzüglich 32 Steuerzeichen sowie Ziffern und einer Reihe von Satz- und Sonderzeichen) dargestellt werden; und das reicht nicht einmal für die Akzentbuchstaben (Umlaute) aller westeuropäischen Sprachen. Erst mit Unicode gibt es einen Standard, der weltweit alle Zeichen (z. B. auch die chinesischen Zeichen) darstellen und speichern soll.

Da ältere EDV-Systeme (Computer, Programme, Programmiersprachen, Datenbanken) weiterhin benutzt werden, muss die Verwendung nationaler Zeichensätze nach wie vor berücksichtigt werden. Dafür gibt es verschiedene Maßnahmen – jedes DBMS folgt eigenen Regeln für **CHARACTER SET** (Zeichensatz) und **COLLATE** (alphabetische Sortierung) und benutzt eigene Bezeichner für die Zeichensätze und die Regeln der Reihenfolge.

Vor allem wegen der DBMS-spezifischen Bezeichnungen kommen Sie nicht um intensive Blicke in die jeweilige Dokumentation herum.

12.4.1. Zeichensatz festlegen mit CHARACTER SET / CHARSET

Wenn eine Datenbank erstellt wird, muss ein Zeichensatz festgelegt werden. Wenn diese Festlegung fehlt, regelt jedes DBMS selbst je nach Version, welcher Zeichensatz als Standard gelten soll. Wenn ein Programm Zugriff auf eine vorhandene Datenbank nimmt, muss ebenso der verwendete Zeichensatz angegeben werden; dieser muss mit dem ursprünglich festgelegten übereinstimmen. Wenn Umlaute falsch angezeigt werden, dann stimmen in der Regel diese Angaben nicht.

Eine neue Datenbank sollte, wenn das DBMS und die verwendete Programmiersprache dies unterstützen, möglichst mit Unicode (in der Regel als UTF8) angelegt werden.

In neueren Versionen steht die Bezeichnung NCHAR (= NATIONAL CHAR) oft nicht für einen speziellen nationalen Zeichensatz, sondern für den allgemeinen Unicode-Zeichensatz. Bei CHAR bzw. VARCHAR wird ein spezieller Zeichensatz verwendet, abhängig von der Installation oder der Datenbank.

In diesem Abschnitt kann deshalb nur beispielhaft gezeigt werden, wie Zeichensätze behandelt werden.

- Firebird, Interbase; MySQL

CHARACTER SET beschreibt den Zeichensatz einer Datenbank. Dieser gilt als Vorgabewert für alle Zeichenketten: CHAR(n), VARCHAR(n), CLOB. Für eine einzelne Spalte kann ein abweichender Zeichensatz festgelegt werden. Beispiel:

Firebird Quelltext

```
CREATE DATABASE 'europe.fb' DEFAULT CHARACTER SET ISO8859_1;
ALTER TABLE xyz ADD COLUMN lname VARCHAR(30) NOT NULL CHARACTER SET CYRL;
```

Es kommt auch vor, dass ein Programm mit einem anderen Zeichensatz arbeitet als die Datenbank. Dann können die Zeichensätze angepasst werden:

Firebird Quelltext

```
SET NAMES DOS437;
CONNECT 'europe.fb' USER 'JAMES' PASSWORD 'U4EEAH';
-- die Datenbank selbst arbeitet mit ISO8859_1, das Programm mit der
   DOS-Codepage 437
```

- MS-SQL

In der Dokumentation wird nur allgemein auf „Nicht-Unicode-Zeichendaten“ eingegangen. Es gibt keinerlei Erläuterung, wie der dafür verwendete Zeichensatz festgelegt wird.

12.4.2. Sortierungen mit COLLATE

COLLATE legt fest, nach welchen Regeln die Reihenfolge von Zeichenketten (englisch: Collation Order) bestimmt wird. Der Vorgabewert für die Datenbank bzw. Tabelle hängt direkt vom Zeichensatz ab.

Abweichende Regeln können getrennt gesteuert werden für Spalten, Vergleiche sowie Festlegungen bei ORDER BY und GROUP BY.

Firebird Quelltext

```
CREATE DATABASE 'europe.fb' DEFAULT CHARACTER SET ISO8859_1;
-- dies legt automatisch die Sortierung nach ISO8859_1 fest
ALTER TABLE xyz ADD COLUMN lname VARCHAR(30) NOT NULL COLLATE FR_CA;
-- dies legt die Sortierung auf kanadisches Französisch fest
SELECT ... WHERE lname COLLATE FR_FR <= :lname_search;
-- dabei soll der Vergleich nach Französisch (Frankreich) durchgeführt
   werden
SELECT ...
```

```
ORDER BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA
GROUP BY LNAME COLLATE FR_CA, FNAME COLLATE FR_CA;
-- vergleichbare Festlegungen für Reihenfolge und Gruppierung bei SELECT
```

Diese Beispiele arbeiten mit den Kürzeln für Interbase/Firebird. Andere DBMS nutzen eigene Bezeichnungen; aber die Befehle selbst sind weitgehend identisch.

12.5. Zusammenfassung

In diesem Kapitel lernten Sie die Datentypen unter SQL kennen:

- Bei Zeichenketten ist zwischen fester und variabler Länge zu unterscheiden.
- Der Zeichensatz – UNICODE oder national – spielt bei Speicherung und Sortierung eine Rolle.
- Für Zahlen ist zwischen exakter und näherungsweise Speicherung zu unterscheiden und die Genauigkeit zu beachten.
- Für Datums- und Zeitwerte ist vor allem auf den jeweiligen Geltungsbereich und die Genauigkeit zu achten.
- Für spezielle Zwecke gibt es weitere Datentypen wie BLOB oder BOOLEAN.

12.6. Übungen

In den Übungen werden für Zahlen und Datumsangaben immer die im deutschsprachigen Raum üblichen Schreibweisen verwendet. Zeichen werden mit Hochkommata begrenzt.

Übung 1 – Texte und Zahlen

Geben Sie bei den folgenden Werten jeweils an, welche Datentypen passen, welche fraglich sind (also u. U. möglich, aber nicht sinnvoll oder unklar) und welche falsch sind.

1. 'A' als Char, Char(20), Varchar, Varchar(20)
2. der Ortsname 'Bietigheim-Bissingen' als Char, Char(20), Varchar, Varchar(20)
3. das Wort 'Übungen' als Varchar(20), NVarchar(20)
4. 123.456 als Integer, Smallint, Float, Numeric, Varchar(20)
5. 123,456 als Integer, Smallint, Float, Numeric, Varchar(20)

6. 789,12 [€] als Integer, Smallint, Float, Numeric, Varchar(20)

Übung 2 – Datum und Zeit

Geben Sie bei den folgenden Werten jeweils an, welche Datentypen passen, welche fraglich sind (also u. U. möglich, aber nicht sinnvoll oder unklar) und welche falsch sind.

1. '27.11.2009' als Date, Time, Timestamp, Char(10), Varchar(20)
2. '11:42:53' als Date, Time, Timestamp, Char(10), Varchar(20)
3. '27.11.2009 11:42:53' als Date, Time, Timestamp, Char(10), Varchar(20)
4. 'November 2009' als Date, Time, Timestamp, Char(10), Varchar(20)

Übung 3 – Personen

Bereiten Sie eine Tabelle *Person* vor: Notieren Sie mit möglichst konsequenter Aufteilung der Bestandteile die möglichen Spaltennamen und deren Datentypen; berücksichtigen Sie dabei auch internationale Adressen, aber keine Kontaktdaten (wie Telefon).

Übung 4 – Buchhaltung

Bereiten Sie eine Tabelle *Kassenbuch* vor: Notieren Sie die möglichen Spaltennamen und deren Datentypen; berücksichtigen Sie dabei auch, dass Angaben der Buchhaltung geprüft werden müssen.

12.7. Lösungen

Lösung zu Übung 1 – Texte und Zahlen

Die „richtige“ Festlegung hängt vom Zusammenhang innerhalb einer Tabelle ab.

1. Char und Varchar(20) passen, Varchar ist nicht sinnvoll, Char(20) ist falsch.
2. Char(20) und Varchar(20) passen, Char und Varchar sind falsch.
3. Je nach verwendetem Zeichensatz können beide Varianten richtig, ungeeignet oder falsch sein.

4. Integer und Numeric sind richtig, Float ist möglich, Smallint ist falsch, Varchar(20) ist nicht ganz ausgeschlossen.
5. Float und Numeric sind richtig, Integer und Smallint sind falsch, Varchar(20) ist nicht ganz ausgeschlossen.
6. Numeric ist richtig, Float ist möglich, Integer und Smallint sind falsch, Varchar(20) ist nicht ganz ausgeschlossen.

Lösung zu Übung 2 – Datum und Zeit

1. Date und Timestamp sind richtig, Char(10) und Varchar(20) sind möglich, Time ist falsch.
2. Time und Timestamp sind richtig, Varchar(20) ist möglich, Char(10) und Date sind falsch.
3. Timestamp ist richtig, Varchar(20) ist möglich, Date, Time und Char(10) sind falsch.
4. Varchar(20) ist richtig, alles andere falsch.

Lösung zu Übung 3 – Personen

Bitte wundern Sie sich nicht über unerwartete Unterteilungen: Bei der folgenden Lösung werden auch Erkenntnisse der Datenbank-Theorie und der praktischen Arbeit mit Datenbanken berücksichtigt.

- ID Integer
- Titel Varchar(15)
- Vorname Varchar(30)
- Adelszusatz Varchar(15) – Trennung ist wegen der alphabetischen Sortierung sinnvoll
- Name Varchar(30)
- Adresszusatz Varchar(30)
- Strasse Varchar(24)
- Hausnr Integer (oder Smallint)
- HausnrZusatz Varchar(10) – Trennung ist wegen der numerischen Sortierung sinnvoll
- Länderkennung Char(2) – nach ISO 3166, auch Char(3) möglich, Integer oder Smallint denkbar; der Ländername ist auf jeden Fall unpraktisch
- PLZ Char(10) oder Varchar(10) – international sind bis zu 10 Zeichen möglich
- Geburtsdatum Date

Lösung zu Übung 4 – Buchhaltung

- ID Integer
- Buchungsjahr Integer oder Smallint
- Buchungsnummer Integer
- Buchungstermin Timestamp – je nach Arbeitsweise genügt auch Date
- Betrag Numeric oder Decimal
- Vorgang Varchar(50) – als Beschreibung der Buchung
- Bearbeiter Varchar(30) – derjenige, der den Kassenbestand ändert
- Nutzer Varchar(30) – derjenige, der die Buchung registriert
- Buchhaltung Timestamp – Termin, zu dem die Buchung registriert wird

Wenn das Kassenbuch explizit ein Teil der Buchhaltung ist, werden auch Spalten wie Buchungskonto (Haupt- und Gegenkonten) benötigt.

12.8. Siehe auch

In Wikipedia gibt es zusätzliche Hinweise:

- GLEITKOMMAZAHLEN² mit ausführlicher Erläuterung ihrer Ungenauigkeiten
- KOORDINIERTE WELTZEIT (UTC)³
- UNICODE⁴ als umfassender Zeichensatz
- ISO 3166⁵ – POSTLEITZAHL⁶ – LISTE DER POSTLEITSYSTEME⁷

2 [HTTP://DE.WIKIPEDIA.ORG/WIKI/GLEITKOMMAZAHLEN](http://de.wikipedia.org/wiki/Gleitkommazahlen)

3 [HTTP://DE.WIKIPEDIA.ORG/WIKI/KOORDINIERTE%20WELTZEIT](http://de.wikipedia.org/wiki/Koordinierte_Weltzeit)

4 [HTTP://DE.WIKIPEDIA.ORG/WIKI/UNICODE](http://de.wikipedia.org/wiki/Unicode)

5 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ISO%203166](http://de.wikipedia.org/wiki/ISO_3166)

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/POSTLEITZAHL](http://de.wikipedia.org/wiki/Postleitzahl)

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/LISTE%20DER%20POSTLEITSYSTEME](http://de.wikipedia.org/wiki/Liste_der_Postleitsysteme)

13. Funktionen

Im SQL-Standard werden verschiedene Funktionen festgelegt, die in jedem SQL-Dialekt vorkommen. In aller Regel ergänzt jedes DBMS diese Funktionen durch weitere eigene.

- **Skalarfunktionen** verarbeiten Werte oder Ausdrücke aus einzelnen Zahlen, Zeichenketten oder Datums- und Zeitwerten. Sofern die Werte aus einer Spalte geholt werden, handelt es sich immer um einen Wert aus einer einzelnen Zeile.
- **Spaltenfunktionen** verarbeiten alle Werte aus einer Spalte.
- Ergänzend kann man für beide Varianten auch **benutzerdefinierte Funktionen** erstellen; dies wird unter PROGRAMMIERUNG¹ angesprochen.

Dieses Kapitel enthält als Grundlage nur die wichtigsten Skalarfunktionen. Unter FUNKTIONEN (2)² gibt es viele Ergänzungen.

13.1. Allgemeine Hinweise

Die Funktionen können überall dort verwendet werden, wo ein SQL-Ausdruck möglich ist. Wichtig ist, dass das Ergebnis der Funktion zu dem Datentyp passt, der an der betreffenden Stelle erwartet wird.

Auch wenn der SQL-Standard nur einige wenige Funktionen vorschreibt, können eine ganze Reihe von Funktionen als Standard angesehen werden, weil sie immer (oder fast immer) vorhanden sind. An vielen Stellen ist aber auf Unterschiede im Namen oder in der Art des Aufrufs hinzuweisen.

Bei Firebird wurden mit Version 2.1 viele Funktionen direkt eingebaut, die vorher als benutzerdefinierte Funktionen (user defined functions, UDF) manuell eingebunden werden mussten.

1 Kapitel 29 auf Seite 363

2 Kapitel 15 auf Seite 157

Schreibweise für Funktionen: Bei Funktionen müssen die Parameter (auch „Argumente“ genannt) immer in Klammern gesetzt werden. Diese Klammern sind auch bei einer Funktion mit konstantem Wert wie PI erforderlich. Eine Ausnahme sind lediglich die Systemfunktionen CURRENT_DATE u. ä.

Schreibweise der Beispiele: Aus Platzgründen werden Beispiele meistens nur einfach in einen Rahmen gesetzt. Für Funktionen ohne Bezug auf Tabellen und Spalten genügt in der Regel ein einfacher SELECT-Befehl; in manchen Fällen muss eine Tabelle oder eine fiktive Quelle angegeben werden:

```
SELECT 2 * 3;                               /* Normalfall */
SELECT 2 * 3 FROM rdb$database;             /* bei Firebird und Interbase
*/
SELECT 2 * 3 FROM dual;                     /* bei Oracle */
```

In diesem Kapitel und auch im zweiten Kapitel zu Funktionen werden diese Verfahren durch eine verkürzte Schreibweise zusammengefasst:

```
SELECT 2 * 3 [from fiktiv];
```

Das ist so zu lesen: Die FROM-Klausel ist für Firebird, Interbase und Oracle notwendig und muss die jeweils benötigte Tabelle angeben; bei allen anderen DBMS muss sie entfallen.

13.2. Funktionen für Zahlen

Bei allen numerischen Funktionen müssen Sie auf den genauen Typ achten. Es ist beispielsweise ein Unterschied, ob das Ergebnis einer Division zweier ganzer Zahlen als ganze Zahl oder als Dezimalzahl behandelt werden soll.

13.2.1. Operatoren

Für Zahlen stehen die üblichen Operatoren zur Verfügung:

```
+ Addition
- Subtraktion, Negation
* Multiplikation
/ Division
```

Dafür gelten die üblichen mathematischen Regeln (Punkt vor Strich, Klammern zuerst). Bitte beachten Sie auch folgende Besonderheit:

- Bei der Division ganzer Zahlen ist auch das Ergebnis eine ganze Zahl; man nennt das „Integer-Division“:

```
SELECT 3 / 5      [from fiktiv];      /* Ergebnis: 0      */
```

- Wenn Sie das Ergebnis als Dezimalzahl haben wollen, müssen Sie (mindestens) eine der beiden Zahlen als Dezimalzahl vorgeben:

```
SELECT 3.0 / 5   [from fiktiv];      /* Ergebnis: 0.6    */
```

- Ausnahme: MySQL liefert auch bei "3/5" eine Dezimalzahl. Für die „Integer-Division“ gibt es die DIV-Funktion:

```
SELECT 3 DIV 5;                                /* Ergebnis: 0      */
```

- Division durch 0 liefert zunächst eine Fehlermeldung "division by zero has occurred" und danach das Ergebnis NULL.

13.2.2. MOD – der Rest einer Division

Die Modulo-Funktion **MOD** bestimmt den Rest bei einer Division ganzer Zahlen:

```
MOD( <dividend>, <divisor> )      /* allgemein */
<dividend> % <divisor>            /* bei MS-SQL und MySQL */
```

Beispiele:

```
SELECT MOD(7, 3)  [from fiktiv];          /* Ergebnis:
1 */
SELECT ID FROM Mitarbeiter WHERE MOD(ID, 10) = 0; /* listet die
IDs mit '0' am Ende */
```

13.2.3. CEILING, FLOOR, ROUND, TRUNCATE – die nächste ganze Zahl

Es gibt mehrere Möglichkeiten, zu einer Dezimalzahl die nächste ganze Zahl zu bestimmen.

CEILING oder **CEIL** liefert die nächstgrößere ganze Zahl, genauer: die kleinste Zahl, die größer oder gleich der gegebenen Zahl ist.

```
SELECT CEILING(7.3), CEILING(-7.3)    [from fiktiv];    /* Ergebnis:  
8, -7 */
```

FLOOR ist das Gegenstück dazu und liefert die nächstkleinere ganze Zahl, genauer: die größte Zahl, die kleiner oder gleich der gegebenen Zahl ist.

```
SELECT FLOOR(7.3), FLOOR(-7.3)      [from fiktiv];    /* Ergebnis:  
7, -8 */
```

TRUNCATE oder **TRUNC** schneidet den Dezimalanteil ab.

```
SELECT TRUNCATE(7.3), TRUNCATE(-7.3) [from fiktiv];    /* Ergebnis:  
7, -7 */  
SELECT TRUNCATE(Schadenshoehe) FROM Schadensfall;    /* liefert  
die Euro-Werte ohne Cent */
```

ROUND liefert eine mathematische Rundung: ab 5 wird aufgerundet, darunter wird abgerundet.

```
ROUND( <Ausdruck> [ , <Genauigkeit> ] )
```

<Ausdruck> ist eine beliebige Zahl oder ein Ausdruck, der eine beliebige Zahl liefert.

- Wenn <Genauigkeit> nicht angegeben ist, wird 0 angenommen.
- Bei einer positiven Zahl für <Genauigkeit> wird auf entsprechend viele Dezimalstellen gerundet.
- Bei einer negativen Zahl für <Genauigkeit> wird links vom Dezimaltrenner auf entsprechend viele Nullen gerundet.

Beispiele:


```

SELECT ROUND(12.248,-2) [from fiktiv];          /* Ergebnis:
0,000 */
SELECT ROUND(12.248,-1) [from fiktiv];        /* Ergebnis:
10,000 */
SELECT ROUND(12.248, 0) [from fiktiv];        /* Ergebnis:
12,000 */
SELECT ROUND(12.248, 1) [from fiktiv];        /* Ergebnis:
12,200 */
SELECT ROUND(12.248, 2) [from fiktiv];        /* Ergebnis:
12,250 */
SELECT ROUND(12.248, 3) [from fiktiv];        /* Ergebnis:
12,248 */
SELECT ROUND(12.25, 1) [from fiktiv];         /* Ergebnis:
12,300 */
SELECT ROUND(Schadenshoehe) FROM Schadensfall; /* liefert
die Euro-Werte gerundet */

```

13.3. Funktionen für Zeichenketten

Zur Bearbeitung und Prüfung von Zeichenketten (Strings) werden viele Funktionen angeboten.



13.3.1. Verknüpfen von Strings

Als **Operatoren**, um mehrere Zeichenketten zu verbinden, stehen zur Verfügung:

```

||      als SQL-Standard
+       für MS-SQL oder MySQL
CONCAT  für MySQL oder Oracle

```

Der senkrechte Strich wird als „Verkettungszeichen“ bezeichnet und oft auch „Pipe“-Zeichen genannt. Es wird auf der deutschen PC-Tastatur unter Windows durch die Tastenkombination  +  erzeugt.

Ein Beispiel in diesen Varianten:

```

SELECT Name || ', ' || Vorname   from Mitarbeiter;
SELECT Name + ', ' + Vorname     from Mitarbeiter;
SELECT CONCAT(Name, ', ', Vorname) from Mitarbeiter;

```

Alle diese Varianten liefern das gleiche Ergebnis: Für jeden Datensatz der Tabelle *Mitarbeiter* werden Name und Vorname verbunden und dazwischen ein weiterer String gesetzt, bestehend aus Komma und einem Leerzeichen.

13.3.2. Länge von Strings

Um die Länge einer Zeichenkette zu erfahren, gibt es die folgenden Funktionen:

CHARACTER_LENGTH(<string>)	SQL-Standard
CHAR_LENGTH(<string>)	SQL-Standard Kurzfassung
LEN(<string>)	nur für MS-SQL

Beispiele:

```
SELECT CHAR_LENGTH('Hello World')      [from fiktiv]; /* Ergebnis:
11 */
SELECT CHAR_LENGTH('')                  [from fiktiv]; /* Ergebnis:
0 */
SELECT CHAR_LENGTH( NULL )              [from fiktiv]; /* Ergebnis:
<null> */
SELECT Name FROM Mitarbeiter ORDER BY CHAR_LENGTH(Name) DESC;
/* liefert die Namen der Mitarbeiter, absteigend sortiert nach
Länge */
```

13.3.3. UPPER, LOWER – Groß- und Kleinbuchstaben

UPPER konvertiert den gegebenen String zu Großbuchstaben; **LOWER** gibt einen String zurück, der nur aus Kleinbuchstaben besteht.

```
SELECT UPPER('Abc Äöü Xyzß ÀÉÇ àéç') [from fiktiv]; /* Ergebnis:
'ABC ÄÖÜ XYZß ÀÉÇ ÀÉÇ' */
SELECT LOWER('Abc Äöü Xyzß ÀÉÇ àéç') [from fiktiv]; /* Ergebnis:
'abc äöü xyzß àéç àéç' */
SELECT UPPER(Kuerzel), Bezeichnung FROM Abteilung; /*
Kurzbezeichnungen in Großbuchstaben */
```

Ob die Konvertierung bei Umlauten richtig funktioniert, hängt vom verwendeten Zeichensatz ab.

13.3.4. SUBSTRING – Teile von Zeichenketten

SUBSTRING ist der SQL-Standard, um aus einem String einen Teil herauszuholen:

```
SUBSTRING( <text> FROM <start> FOR <anzahl> )      /* SQL-Standard
*/
SUBSTRING( <text> ,      <start> ,      <anzahl> )  /* MS-SQL, MySQL,
Oracle */
```

Diese Funktion heißt unter Oracle **SUBSTR** und kann auch bei MySQL so bezeichnet werden.

Der Ausgangstext wird von Position 1 an gezählt. Der Teilstring beginnt an der hinter FROM genannten Position und übernimmt so viele Zeichen wie hinter FOR angegeben ist:

```
SELECT SUBSTRING('Abc Def Ghi' FROM 6 FOR 4) [from fiktiv]; /*
Ergebnis: 'ef G' */
SELECT CONCAT(Name, ', ', SUBSTRING(Vorname FROM 1 FOR 1), '.') from
Mitarbeiter;
/* liefert den Namen und vom Vornamen den Anfangsbuchstaben */
```

Wenn der <anzahl>-Parameter fehlt, wird alles bis zum Ende von <text> übernommen:

```
SELECT SUBSTRING('Abc Def Ghi' FROM 6) [from fiktiv]; /*
Ergebnis: 'ef Ghi' */
```

Wenn der <anzahl>-Parameter 0 lautet, werden 0 Zeichen übernommen, man erhält also einen leeren String.

MySQL bietet noch eine Reihe weiterer Varianten.

Hinweis: Nach SQL-Standard liefert das Ergebnis von SUBSTRING seltsamerweise einen Text von gleicher Länge wie der ursprüngliche Text; die jetzt zwangsläufig folgenden Leerzeichen müssen ggf. mit der TRIM-Funktion (im zweiten Kapitel über FUNKTIONEN³) entfernt werden.

13.4. Funktionen für Datums- und Zeitwerte

Bei den Datums- und Zeitfunktionen gilt das gleiche wie für Datum und Zeit als Datentyp: Jeder SQL-Dialekt hat sich seinen eigenen „Standard“ ausgedacht. Wir müssen uns deshalb auf die wichtigsten Funktionen beschränken, die es so oder so ähnlich „immer“ gibt, und können nur auf die Dokumentation zum jeweiligen DBMS verweisen.

Vor allem MySQL bietet viele zusätzliche Funktionen an. Teilweise sind es nur verkürzte und spezialisierte Schreibweisen der Standardfunktionen, teilweise liefern sie zusätzliche Möglichkeiten.

13.4.1. Systemdatum und -uhrzeit

Nach SQL-Standard werden die aktuelle Uhrzeit und das aktuelle Datum abgefragt:

```
CURRENT_DATE  
CURRENT_TIME  
CURRENT_TIMESTAMP
```

In Klammern kann als <precision> die Anzahl der Dezimalstellen bei den Sekunden angegeben werden.

Beispiele:

```
SELECT CURRENT_TIMESTAMP [from fiktiv]; /* Ergebnis: '19.09.2009  
13:47:49' */  
UPDATE Versicherungsvertrag SET Aenderungsdatum = CURRENT_DATE WHERE  
irgendetwas;
```

Bei MS-SQL gibt es nur CURRENT_TIMESTAMP, aber dafür andere Funktionen mit höherer Genauigkeit.

13.4.2. Teile von Datum oder Uhrzeit bestimmen

Für diesen Zweck gibt es vor allem **EXTRACT** als Standardfunktion:

```
EXTRACT ( <part> FROM <value> )
```

<value> ist der Wert des betreffenden Datums und/oder der Uhrzeit, die aufgeteilt werden soll. Als <part> wird angegeben, welcher Teil des Datums gewünscht wird:

```
YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | MILLISECOND
```

Bei einem DATE-Feld ohne Uhrzeit sind HOUR usw. natürlich unzulässig, bei einem TIME-Feld nur mit Uhrzeit die Bestandteile YEAR usw. Beispiele:

```
SELECT ID, Datum, EXTRACT(YEAR FROM Datum) AS Jahr, EXTRACT(MONTH
FROM Datum) AS Monat
  FROM Schadensfall   ORDER BY Jahr, Monat
SELECT 'Stunde = ' || EXTRACT(HOUR FROM CURRENT_TIME) [from fiktiv];
/* Ergebnis: 'Stunde = 14' */
```

Bei MS-SQL heißt diese Standardfunktion DATEPART; als <part> können viele weitere Varianten genutzt werden.

Sehr oft gibt es weitere Funktionen, die direkt einen Bestandteil abfragen, zum Beispiel:

```
YEAR( <value> )      liefert das Jahr
MONTH( <value> )     liefert den Monat usw.
MINUTE( <value> )    liefert die Minute usw.
DAYOFWEEK( <value> ) gibt den Wochentag (als Zahl, 1 für Sonntag
usw.) an
```

Wie gesagt: Lesen Sie in der DBMS-Dokumentation nach, was es sonst noch gibt.

13.5. Funktionen für logische und NULL-Werte

Wenn man es genau nimmt, gehören dazu auch **Prüfungen** wie „Ist ein Wert NULL?“. Der SQL-Standard hat dazu und zu anderen Prüfungen verschiedene spezielle Ausdrücke vorgesehen. Diese gehören vor allem zur WHERE-

KLAUSEL⁴ des SELECT-Befehls:

= < > usw.	Größenvergleich zweier Werte
BETWEEN AND	Werte zwischen zwei Grenzen
LIKE	Ähnlichkeiten (1)
CONTAINS u.a.	Ähnlichkeiten (2)
IS NULL	null-Werte prüfen
IN	genauer Vergleich mit einer Liste
EXISTS	schneller Vergleich mit einer Liste

Alle diese Ausdrücke liefern einen der logischen Werte TRUE, FALSE – also WAHR oder FALSCH – und ggf. NULL – also <unbekannt> – zurück und können als boolescher Wert weiterverarbeitet oder ausgewertet werden.

13.5.1. Operatoren

Zur **Verknüpfung logischer Werte** gibt es die „booleschen Operatoren“ NOT, AND, OR (nur bei MySQL auch XOR):

NOT	als Negation
AND	als Konjunktion
OR	als Adjunktion
XOR	als Kontravalenz

Auch diese Operatoren werden bei der WHERE-Klausel behandelt.

Zur **Verknüpfung von NULL-Werten** gibt es vielfältige Regeln, je nach dem Zusammenhang, in dem das von Bedeutung ist. Man kann sich aber folgende Regel merken:

- Wenn ein Ausgangswert NULL ist, also <unbekannt>, und dieser „Wert“ mit etwas verknüpft wird (z. B. mit einer Zahl addiert wird), kann das Ergebnis nur <unbekannt> sein, also NULL lauten.

13.6. Konvertierungen

In der EDV – also auch bei SQL-Datenbanken – ist der verwendete Datentyp immer von Bedeutung. Mit Zahlen kann gerechnet werden, mit Zeichenketten nicht. Größenvergleiche von Zahlen gelten immer und überall; bei Ze-

4 Kapitel 16 auf Seite 173

ichenketten hängt die Reihenfolge auch von der Sprache ab. Ein Datum wird in Deutschland durch '11.09.2001' beschrieben, in England durch '11/09/2001' und in den USA durch '09/11/2001'. Wenn *wir* etwas aufschreiben (z. B. einen SQL-Befehl), dann benutzen wir zwangsläufig immer Zeichen bzw. Zeichenketten, auch wenn wir Zahlen oder ein Datum meinen.

In vielen Fällen „versteht“ das DBMS, was wir mit einer solchen Schreibweise meinen; dann werden durch „implizite Konvertierung“ die Datentypen automatisch ineinander übertragen. In anderen Fällen muss der Anwender dem DBMS durch eine Konvertierungsfunktion erläutern, was wie zu verstehen ist.

13.6.1. Implizite Konvertierung

Datentypen, die problemlos vergleichbar sind, werden „automatisch“ ineinander übergeführt.

- Die Zahl 17 kann je nach Situation ein INTEGER, ein SMALLINT, ein BIGINT, aber auch NUMERIC oder FLOAT sein.
- Die Zahl 17 kann in einem SELECT-Befehl auch als String '17' erkannt und verarbeitet werden.

```
SELECT ID, Name, Vorname FROM Mitarbeiter WHERE ID = '17';
```

- Je nach DBMS kann ein String wie '11.09.2001' als Datum erkannt und verarbeitet werden. Vielleicht verlangt es aber auch eine andere Schreibweise wie '11/09/2001'. Die Schreibweise '2009-09-11' nach ISO 8601 sollte dagegen immer richtig verstanden werden (aber auch da gibt es Abweichungen).

Es gibt bereits durch den SQL-Standard ausführliche Regeln, welche Typen immer, unter bestimmten Umständen oder niemals ineinander übergeführt werden können. Jedes DBMS ergänzt diese allgemeinen Regeln durch eigene.

Wenn ein solcher Befehl ausgeführt wird, dürfte es niemals Missverständnisse geben, sondern er wird „mit an Sicherheit grenzender Wahrscheinlichkeit“ korrekt erledigt. Wenn ein Befehl nicht eindeutig ist, wird das DBMS eher zu früh als zu spät mit einer Fehlermeldung wie "conversion error from string ..." reagieren. Dann ist es Zeit für eine explizite Konvertierung, meistens durch CAST.

13.6.2. CAST

Die **CAST**-Funktion ist der SQL-Standard für die Überführung eines Wertes von einem Datentyp in einen anderen.

```
CAST ( <expression> AS <type> )
```

Als <expression> ist etwas angegeben, was den „falschen“ Typ hat, nämlich ein Wert oder Ausdruck. Mit <type> wird der Datentyp angegeben, der an der betreffenden Stelle gewünscht oder benötigt wird. Das Ergebnis des CASTings (der Konvertierung) ist dann genau von diesem Typ.

Beispiele für Datum:

```
SELECT EXTRACT(DAY FROM '07.14.2009')           [from fiktiv];

/* expression evaluation not supported */
SELECT EXTRACT(DAY FROM CAST('07.14.2009' as date)) [from fiktiv];
/* conversion error from string "07.14.2009" */
SELECT EXTRACT(DAY FROM CAST('14.07.2009' as date)) [from fiktiv];
/* Ergebnis: '14' */
SELECT Name, Vorname, CAST(Geburtsdatum AS CHAR(10)) FROM
Versicherungsnehmer;
/* Ergebnis wird in der Form '1953-01-13' angezeigt */
```

Kürzere Zeichenketten können schnell verlängert werden, wenn es nötig ist:

```
SELECT ID, CAST(Kuerzel AS CHAR(20)) FROM Abteilung;
-- 20 Zeichen Länge statt eigentlich 10 Zeichen
```

Das Verkürzen funktioniert nicht immer so einfach. Ob bei Überschreitung einer Maximallänge einfach abgeschnitten wird oder ob es zu einer Fehlermeldung "string truncation" kommt, hängt vom DBMS ab; dann müssen Sie ggf. eine SUBSTRING-Variante benutzen.

```
-- vielleicht funktioniert es so:
SELECT CAST(Name AS CHAR(15)) || Vorname from Versicherungsnehmer;
-- aber sicherer klappt es so:
SELECT SUBSTRING(Name FROM 1 FOR 15) || Vorname from
Versicherungsnehmer;
-- unter Berücksichtigung des Hinweises bei SUBSTRING:
SELECT TRIM( SUBSTRING(Name FROM 1 FOR 15) ) || Vorname from
Versicherungsnehmer;
```


Bitte lesen Sie in Ihrer SQL-Dokumentation nach, zwischen welchen Datentypen implizite Konvertierung möglich ist und wie die explizite Konvertierung mit CAST ausgeführt wird.

13.6.3. CONVERT

Nach dem SQL-Standard ist **CONVERT** vorgesehen zum Konvertieren von Zeichenketten in verschiedenen Zeichensätzen:

CONVERT (<text> USING <transcoding name>)	SQL-Standard
CONVERT (<text>, <transcoding name>)	alternative
Schreibweise	

Firebird kennt diese Funktion überhaupt nicht. MS-SQL benutzt eine andere Syntax und bietet vor allem für Datums- und Zeitformate viele weitere Möglichkeiten:

CONVERT (<type>, <text> [, <style>])
--

Wegen dieser starken Abweichungen verzichten wir auf weitere Erläuterungen und verweisen auf die jeweilige Dokumentation.

13.6.4. Datum und Zeit

Vor allem für die Konvertierung mit Datums- und Zeitangaben bieten die verschiedenen DBMS Erweiterungen. Beispiele:

- MS-SQL hat die Syntax von **CONVERT** für diesen Zweck erweitert.
- MySQL ermöglicht die Konvertierung mit **STR_TO_DATE** und **DATE_FORMAT**.
- Oracle kennt eine Reihe von Funktionen wie **TO_DATE** usw..

Noch ein Grund für das Studium der Dokumentation...

13.7. Spaltenfunktionen

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. Näheres siehe unter GRUPPIERUNGEN⁵.

13.7.1. 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).

5 Kapitel 24 auf Seite 271

13.7.2. 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;
```

Eine Besonderheit ist das Berechnen von Vergleichen. Ein Vergleich wird als WAHR oder FALSCH ausgewertet. Sofern das DBMS (wie bei MySQL oder Access) das Ergebnis als Zahl benutzt, ist das Ergebnis eines Vergleichs daher 1 oder 0 (bei Access -1 oder 0). Um alle Fälle zu zählen, deren Schadenshöhe größer als 1000 ist, müsste der Befehl so aussehen:

```
SELECT SUM(Schadenshoehe > 1000) AS Anzahl_Schadenshoehe_gt_1000 FROM Schadensfall;
```

Dabei werden nicht etwa die Schäden aufsummiert, sondern nur das Ergebnis des Vergleichs, also 0 oder 1, im Grunde also gezählt. Die Funktion COUNT kann hier nicht genommen werden, da sie sowohl die 1 als auch die 0 zählen würde.

Einige DBMS (z. B. DB2, Oracle) haben eine strengere Typenkontrolle; Firebird nimmt eine Zwischenstellung ein. Dabei haben Vergleichsausdrücke grundsätzlich ein boolesches Ergebnis, das nicht summiert werden kann. Dann kann man sich mit der CASE-Funktion behelfen, die dem Wahrheitswert TRUE eine 1 zuordnet und dem Wert FALSE eine 0:

```
SELECT SUM(CASE WHEN Schadenshoehe > 1000 THEN 1
              ELSE 0
              END) AS Anzahl_Schadenshoehe_gt_1000
FROM Schadensfall;
```

13.7.3. 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**.

13.7.4. 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;
```

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

13.7.5. STDDEV – Standardabweichung

Die Standardabweichung **STDDEV** oder **STDEV** kann auch nur für numerische Werte berechnet werden. NULL-Werte fließen nicht mit in die Berechnung ein, Nullen schon. Wie bei SUM können auch Berechnungen als Werte genommen werden. Die Standardabweichung der Schadensfälle wird so berechnet:

```
SELECT STDDEV(Schadenshoehe) AS StdAbw_Schadenshoehe FROM Schadensfall;
```

13.8. Zusammenfassung

In diesem Kapitel lernten wir die wichtigsten „eingebauten“ Funktionen kennen:

- Für Zahlen gibt es vor allem die Operatoren, dazu die modulo-Funktionen und Möglichkeiten für Rundungen.
- Für Zeichenketten gibt es vor allem das Verknüpfen und Aufteilen, dazu die Längenbestimmung und die Umwandlung in Groß- und Kleinbuchstaben.
- Für Datums- und Zeitwerte gibt es neben Systemfunktionen die Verwendung einzelner Teile.
- Für logische und NULL-Werte gibt es vor allem Vergleiche und Kombinationen durch Operatoren.
- Konvertierungen – implizite, CAST, CONVERT – dienen dazu, dass ein Wert des einen Datentyps anstelle eines anderen Typs verwendet werden kann.

Mit den Spaltenfunktionen werden alle Werte einer Spalte gemeinsam ausgewertet.

13.9. Übungen

Übung 1 – Definitionen

Welche der folgenden Feststellungen sind richtig, welche sind falsch?

1. Eine Skalarfunktion bestimmt aus allen Werten eines Feldes einen gemeinsamen Wert.

2. Eine Spaltenfunktion bestimmt aus allen Werten eines Feldes einen gemeinsamen Wert.
3. Eine Skalarfunktion kann keine Werte aus Spalten einer Tabelle verarbeiten.
4. Eine benutzerdefinierte Funktion kann als Skalarfunktion, aber nicht als Spaltenfunktion dienen.
5. Wenn einer Funktion keine Argumente übergeben werden, kann auf die Klammern hinter dem Funktionsnamen verzichtet werden.
6. Wenn eine Funktion für einen SQL-Ausdruck benutzt wird, muss das Ergebnis der Funktion vom Datentyp her mit dem übereinstimmen, der an der betreffenden Stelle erwartet wird.

Übung 2 – Definitionen

Welche der folgenden Funktionen sind Spaltenfunktionen? Welche sind Skalarfunktionen, welche davon sind Konvertierungen?

1. Bestimme den Rest bei einer Division ganzer Zahlen.
2. Bestimme die Länge des Namens des Mitarbeiters mit der ID 13.
3. Bestimme die maximale Länge aller Mitarbeiter-Namen.
4. Bestimme die Gesamtlänge aller Mitarbeiter-Namen.
5. Verwandle eine Zeichenkette, die nur Ziffern enthält, in eine ganze Zahl.
6. Verwandle eine Zeichenkette, die keine Ziffern enthält, in einen String, der nur Großbuchstaben enthält.
7. Bestimme das aktuelle Datum.

Übung 3 – Funktionen mit Zahlen

Benutzen Sie für die folgenden Berechnungen die Spalte *Schadenshoehe* der Tabelle *Schadensfall*. Welche Datensätze benutzt werden, also der Inhalt der WHERE-Klausel, soll uns dabei nicht interessieren. Auch geht es nur um die Formeln, nicht um einen SELECT-Befehl.

1. Berechnen Sie (ohne die AVG-Funktion) die durchschnittliche Schadenshöhe.
2. Bestimmen Sie den prozentualen Anteil eines bestimmten Schadensfalls an der gesamten Schadenshöhe.

Übung 4 – Funktionen mit Zeichenketten

Schreiben Sie Name, Vorname und Abteilung der Mitarbeiter in tabellarischer Form (nehmen wir an, dass das *Kuerzel* der Abteilung in der Tabelle *Mitarbeiter* stünde); benutzen Sie dazu nacheinander die folgenden Teilaufgaben:

1. Bringen Sie die Namen auf eine einheitliche Länge von 20 Zeichen.
2. Bringen Sie die Namen auf eine einheitliche Länge von 10 Zeichen.
3. Bringen Sie die Vornamen ebenso auf eine Länge von 10 Zeichen.
4. Setzen Sie diese Teilergebnisse zusammen und fügen Sie dazwischen je 2 Leerzeichen ein.

Übung 5 – Funktionen mit Datum und Zeit

Gegeben ist ein Timestamp-Wert mit dem Spaltennamen *Zeitstempel* und dem Inhalt "16. Dezember 2009 um 19:53 Uhr". Zeigen Sie diesen Wert als Zeichenkette im Format "12/2009; 16. Tag; 7 Minuten vor 20 Uhr" an. (Für die String-Verknüpfung benutzen wir jetzt das Plus-Zeichen. Die Leerzeichen zwischen den Bestandteilen können Sie ignorieren. Auch muss es keine allgemeingültige Lösung sein, die alle Eventualitäten beachtet.)

Übung 6 – Funktionen mit Datum und Zeit

Gegeben ist eine Zeichenkette *datum* mit dem Inhalt "16122009". Sorgen Sie dafür, dass das Datum für jedes DBMS gültig ist. Zusatzfrage: Muss dafür CAST verwendet werden und warum bzw. warum nicht?

13.10. Lösungen

Lösung zu Übung 1 – Definitionen

Richtig sind 2 und 6, falsch sind 1, 3, 4, 5.

Lösung zu Übung 2 – Definitionen

Spaltenfunktionen sind 3 und 4; beide benutzen das Ergebnis von Skalarfunktionen. Alle anderen sind Skalarfunktionen, wobei 5 eine Konvertierung ist und 7 eine Systemfunktion, die ohne Klammern geschrieben wird.

Lösung zu Übung 3 – Funktionen mit Zahlen

1. SUM(Schadenshoehe) / COUNT(Schadenshoehe)
2. ROUND(Schadenshoehe * 100 / SUM(Schadenshoehe))

Lösung zu Übung 4 – Funktionen mit Zeichenketten

```
1. CAST(Name AS CHAR(20))
2. SUBSTRING( CAST(Name AS CHAR(20)) FROM 1 FOR 10 )
3. SUBSTRING( CAST(Vorname AS CHAR(20)) FROM 1 FOR 10 )
4. SUBSTRING( CAST(Name AS CHAR(20)) FROM 1 FOR 10 )   {{{!}} ' '
{{{!}}}
   SUBSTRING( CAST(Vorname AS CHAR(20)) FROM 1 FOR 10 ) {{{!}} ' '
{{{!}}} Kuerzel
```

Lösung zu Übung 5 – Funktionen mit Datum und Zeit

```
EXTRACT(MONTH FROM CURRENT_TIMESTAMP) + '/' + EXTRACT(YEAR FROM
CURRENT_TIMESTAMP) + ';'
+ EXTRACT(DAY FROM CURRENT_TIMESTAMP) + '.Tag; '
+ CAST((60 - EXTRACT(MINUTE FROM CURRENT_TIMESTAMP)) AS Varchar(2)) +
' Minuten vor '
+ CAST( (EXTRACT(HOUR FROM CURRENT_TIMESTAMP) + 1) AS Varchar(2)) + '
Uhr'
```

Lösung zu Übung 6 – Funktionen mit Datum und Zeit

```
SUBSTRING(datum FROM 5 FOR 4) + '-'
+ SUBSTRING(datum FROM 3 FOR 2) + '-'
+ SUBSTRING(datum FROM 1 FOR 2)
```

Auf CAST kann (fast immer) verzichtet werden, weil mit dieser Substring-Verwendung die Standardschreibweise '2009-12-16' nach ISO 8601 erreicht wird.

13.11. Siehe auch

Einige Hinweise sind in den folgenden Kapiteln zu finden:

- SQL-BEFEHLE⁶ beschreibt auch, was unter einem „SQL-Ausdruck“ zu verstehen ist.

6 Kapitel 6 auf Seite 53

- DATENTYPEN⁷

Weitere Erläuterungen stehen bei Wikipedia:

- SENKRECHTER STRICH⁸ oder „Pipe“-Zeichen
- BOOLESCHER OPERATOREN⁹
- ISO 8601¹⁰ zur Schreibweise von Datumsangaben

7 Kapitel 12 auf Seite 107

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SENKRECHTER%20STRICH](http://de.wikipedia.org/wiki/Senkrechter_Strich)

9 [HTTP://DE.WIKIPEDIA.ORG/WIKI/BOOLESCHER%20OPERATOR](http://de.wikipedia.org/wiki/Boolescher_Operator)

10 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ISO%208601](http://de.wikipedia.org/wiki/ISO_8601)

Teil III.

Mehr zu Abfragen

14. Ausführliche SELECT-Struktur

Dieses Kapitel erläutert die Syntax des SELECT-Befehls. Anstelle von Beispielen gibt es Verweise auf diejenigen Kapitel, die die betreffenden Klauseln genauer behandeln.

In diesem Kapitel werden vorzugsweise die englischen Begriffe aus der SQL-Dokumentation benutzt.

Bitte beachten Sie: Der SELECT-Befehl bietet so umfangreiche Möglichkeiten, dass auch bei dieser Übersicht nicht alle Einzelheiten vorgestellt werden können.

14.1. Allgemeine Syntax

Der **SELECT**-Befehl wird als <query specification> oder <select expression>, also **Abfrage-Ausdruck** bezeichnet und setzt sich grundsätzlich aus diesen Bestandteilen zusammen:

```
SELECT
  [ DISTINCT | ALL ]
  <select list>
  FROM <table reference list>
  [ <where clause> ]
  [ <group by clause> ]
  [ <having clause> ]
  [ UNION [ALL] <query specification> ]
  [ <order by clause> ]
```

Diese Bestandteile setzen sich je nach Situation aus weiteren einfachen oder komplexen Bestandteilen zusammen und werden in den folgenden Abschnitten erläutert.

Unbedingt erforderlich sind:

- das Schlüsselwort **SELECT**
- eine Liste von Spalten

- das Schlüsselwort **FROM** mit einem oder mehreren Verweisen auf Tabellen

Alle anderen Bestandteile sind optional, können also auch weggelassen werden. Wenn sie benutzt werden, müssen sie in genau dieser Reihenfolge verwendet werden.

14.2. Set Quantifier – Mengenquantifizierer

Als Mengenquantifizierer stehen **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**.

Erläuterungen dazu finden sich im Kapitel **NÜTZLICHE ERWEITERUNGEN**¹.

14.3. Select List – Auswahlliste

Die Liste der Spalten, die ausgewählt werden sollen, wird angegeben durch den Asterisk oder mit einer `<column list>`.

14.3.1. Asterisk *

Der Asterisk, d. h. das Sternchen *, ist eine Kurzfassung und steht für die Liste aller Spalten (Felder) einer einzelnen Tabelle.

```
*
```

Wenn mehrere Tabellen verknüpft werden, ist der Asterisk zusammen mit dem Namen oder dem Alias einer Tabelle anzugeben.

```
<Tabellennamen>.* /* oder */  
<Tabellen-Alias>.*
```

Erläuterungen dazu finden sich in allen Kapiteln, die sich mit Abfragen befassen.

1 Kapitel 22 auf Seite 237

14.3.2. Column List

Die <Spaltenliste> ist eine Liste von einem oder mehreren Elementen, die in der SQL-Dokumentation als <select sublist> bezeichnet werden:

```
<select sublist> [ , <select sublist> , <select sublist> , ... ]
```

Jedes einzelne Element ist eine Spalte einer Tabelle oder eine abgeleitete Spalte <derived column> – siehe den nächsten Abschnitt. Jedem Element kann ein Spalten-Alias zugewiesen, das Wort AS kann dabei auch weggelassen werden:

```
<element> [ [AS] <column name> ]
```

Bei Spalten aus Tabellen wird einfach deren Name angegeben. Wenn mehrere Tabellen verknüpft werden und ein Spaltenname nicht eindeutig ist, ist der Spaltenname zusammen mit dem Namen oder dem Alias einer Tabelle anzugeben.

```
<spaltenname> /* oder */
<Tabellenname>.<spaltenname> /* oder */
<Tabellen-Alias>.<spaltenname>
```

Hinweis: In manchen DBMS darf der Tabellenname nicht mehr benutzt werden, wenn ein Tabellen-Alias angegeben ist.

Erläuterungen dazu finden sich in allen Kapiteln, die sich mit Abfragen befassen.

14.3.3. Derived Column

Eine abgeleitete Spalte bezeichnet den Wert, der von einem <value expression> zurückgeliefert wird. Das kann ein beliebiger Ausdruck sein, der genau einen Wert als Ergebnis hat: vor allem eine Funktion oder eine passende Unterabfrage.

Erläuterungen dazu finden sich vor allem in den Kapiteln BERECHNETE SPALTEN² und UNTERABFRAGEN³.

2 Kapitel 23 auf Seite 263

3 Kapitel 25 auf Seite 283

14.4. Table Reference List – Tabellen-Verweise

Als Bestandteil der **FROM**-Klausel werden die beteiligten Tabellen und Verweise darauf aufgeführt:

```
FROM <reference list>      /* nämlich */
FROM <reference1> [ , <reference2> , <reference3> ... ]
```

In der <Tabellenliste> stehen eine oder (mit Komma getrennt) mehrere Verweise (Referenzen); diese können direkt aus der Datenbank übernommen oder auf verschiedene Arten abgeleitet werden.

Jede dieser Varianten kann erweitert werden:

```
<reference> [ [ AS ] <Alias-Name>
              [ ( <derived column list> ) ] ]
```

Mit **AS** kann ein Alias-Name dem Verweis hinzugefügt werden; das Schlüsselwort **AS** kann auch entfallen. Diesem Tabellen-Alias kann (je nach DBMS) in Klammern eine Liste von Spaltennamen hinzugefügt werden, die anstelle der eigentlichen Spaltennamen angezeigt werden sollen.

Die folgenden Varianten sind als Verweise möglich.

14.4.1. Tabellen, Views

In erster Linie stehen die Tabellen der Datenbank sowie die Views (fest definierte Sichten) zur Verfügung.

Erläuterungen dazu finden sich in allen Kapiteln, die sich mit Abfragen befassen, sowie in ERSTELLEN VON VIEWS⁴.

14.4.2. Derived Table – Abgeleitete Tabellen

Vor allem können Unterabfragen wie eine Tabelle benutzt werden.

Hinweis: Es gibt noch eine Reihe weiterer Varianten, mit denen aus vorhandenen Tabellen andere Tabellen „vorübergehend“ abgeleitet werden können. Da diese je

4 Kapitel 26 auf Seite 305

nach DBMS unterschiedlich benutzt werden, würde es zu weit führen (und die Übersicht zu kompliziert machen); wir verzichten deshalb auf eine vollständige Übersicht.

Erläuterungen dazu finden sich im Kapitel UNTERABFRAGEN⁵.

14.4.3. Joined Table – Verknüpfte Tabelle

Wie eine Tabelle kann auch eine Verknüpfung verwendet werden. Dabei handelt es sich um eine Teilmenge des kartesischen Produkts zweier Tabellen, die nach bestimmten Bedingungen verbunden werden.

Für die Verknüpfung zweier (oder mehrerer) Tabellen gibt es zwei Verfahren: Beim (traditionellen) direkten Weg werden die beiden Tabellen einfach nacheinander aufgeführt. Beim (modernen) Weg wird zu einer Tabelle eine weitere mit JOIN genannt, die durch eine Verknüpfungsbedingung über ON verbunden wird.

Erläuterungen dazu finden sich in MEHRERE TABELLEN⁶ sowie den dann folgenden Kapiteln.

14.5. Where Clause – WHERE-Klausel

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.

```
WHERE <search condition>
```

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.

5 Kapitel 25 auf Seite 283

6 Kapitel 17 auf Seite 187

Erläuterungen zu den folgenden Einzelheiten finden sich vor allem in den Kapiteln WHERE-KLAUSEL IM DETAIL⁷ und UNTERABFRAGEN⁸.

14.5.1. Eine 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:

```
= < > <= >= <>
```

14.5.2. Mehrere Suchbedingungen

Mehrere Suchbedingungen können miteinander verbunden werden:

```
NOT <search condition>
<search condition1> AND <search condition2>
<search condition1> OR <search condition2>
```

7 Kapitel 16 auf Seite 173

8 Kapitel 25 auf Seite 283

Bitte beachten Sie: NOT hat die stärkste Verbindung und wird zuerst ausgewertet. Danach hat AND eine stärkere Verbindung und wird als nächstes ausgewertet. Erst zum Schluss kommen die OR-Verbindungen. Um Unklarheiten zu vermeiden, wird dringend empfohlen, zusammengesetzte Suchbedingungen in Klammern zu setzen, um Prioritäten deutlich zu machen.

14.6. Group By Clause – GROUP BY-Klausel

Mit der **GROUP BY**-Klausel 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 allgemeine Syntax lautet:

```
GROUP BY <Spaltenliste>
```

Die Spaltenliste enthält, durch Komma getrennt, die Namen von einer oder mehreren Spalten. Bei jeder Spalte kann eine eigene Sortierung angegeben werden (wie bei den DATENTYPEN⁹ erläutert):

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

Erläuterungen dazu finden sich vor allem im Kapitel GRUPPIERUNGEN¹⁰.

14.7. Having Clause – HAVING-Klausel

Die **HAVING**-Klausel dient dazu, nicht alle ausgewählten Zeilen in die Ausgabe zu übernehmen, sondern nur diejenigen, die den zusätzlichen Bedingungen entsprechen. Sie wird in der Praxis überwiegend als Ergänzung zur GROUP BY-Klausel verwendet und folgt ggf. direkt danach.

9 Kapitel 12 auf Seite 107

10 Kapitel 24 auf Seite 271

```
HAVING <bedingungsliste>
```

Erläuterungen dazu finden sich ebenfalls im Kapitel GRUPPIERUNGEN¹¹.

14.8. Union Clause – UNION-Klausel

Mit der **UNION**-Klausel werden mehrere eigentlich getrennte Abfragen zusammengefasst, um ein einheitliches Ergebnis zu liefern. Dabei sind die einzelnen Bedingungen zu komplex, um sie zusammenzufassen; oder sie können nicht sinnvoll verbunden werden. Es setzt eine weitgehend identische Spaltenliste voraus.

```
SELECT <spaltenliste1> FROM <tabellenliste1> WHERE  
<bedingungsliste1>  
UNION  
SELECT <spaltenliste2> FROM <tabellenliste2> WHERE  
<bedingungsliste2>
```

Erläuterungen dazu finden sich im Kapitel NÜTZLICHE ERWEITERUNGEN¹².

14.9. Order By Clause – ORDER BY-Klausel

Mit der **ORDER BY**-Klausel werden die Datensätze der Ergebnismenge in eine bestimmte Reihenfolge gebracht.

Erläuterungen dazu finden sich in allen Kapiteln, die sich mit Abfragen befassen.

14.10. Zusammenfassung

In diesem Kapitel erhielten Sie einen umfangreichen Überblick über die Syntax des SELECT-Befehls:

11 Kapitel 24 auf Seite 271

12 Kapitel 22 auf Seite 237

- Die Listen der gewünschten Spalten und der beteiligten Tabellen sind Pflichtangaben, alle anderen Klauseln sind optional.
- Für die Verknüpfung mehrerer Tabellen gibt es einen (traditionellen) direkten Weg und den (modernen) Weg über JOIN.
- Die WHERE-Klausel ermöglicht komplexe Bedingungen darüber, welche Datensätze abgefragt werden sollen.
- Mit Gruppierung, Sortierung und Zusammenfassung gibt es weitere Möglichkeiten für Abfragen.

14.11. Übungen

Übung 1 – Allgemeine Syntax

Bringen Sie die folgenden Bestandteile des SELECT-Befehls in die richtige Reihenfolge (es gibt Begriffe, die an zwei bzw. drei Stellen gehören):

```
<bedingung> - DISTINCT - FROM - GROUP BY - HAVING -  
ORDER BY - SELECT - <spalten> - <tabelle> - WHERE
```

Übung 2 – Allgemeine Syntax

Welche der genannten Bestandteile eines SELECT-Befehls sind unbedingt erforderlich?

Übung 3 – Spaltenliste

Welche der folgenden Spaltenlisten aus der Beispieldatenbank sind richtig, welche nicht?

1. SELECT * FROM Mitarbeiter;
2. SELECT ID, Name FROM Mitarbeiter;
3. SELECT ID, Name FROM Mitarbeiter, Abteilung;
4. SELECT ID, Name, Kuerzel FROM Mitarbeiter, Abteilung;
5. SELECT ab.ID, Name FROM Mitarbeiter, Abteilung ab;
6. SELECT ab.ID, Name, Krz Kuerzel FROM Mitarbeiter, Abteilung ab;
7. SELECT ab.ID, Name, Kuerzel Krz FROM Mitarbeiter, Abteilung ab;

8. `SELECT ab.ID, mi.Name, ab.Kuerzel FROM Mitarbeiter mi,
Abteilung ab;`

Übung 4 – Spaltenliste

Schreiben Sie für folgende Abfragen die Spalten und Tabellen auf.

1. Zeige alle Informationen zu den Mitarbeitern.
2. Zeige zu jedem Mitarbeiter Name, Vorname und Nummer der Abteilung.
3. Zeige zu jedem Mitarbeiter Name, Vorname und Kuerzel der Abteilung.
4. Zeige zu jedem Mitarbeiter ID, Name, Vorname sowie das Kennzeichen des Dienstwagens.

Übung 5 – Suchbedingungen

Welche der folgenden Suchbedingungen sind richtig, welche nicht? Welche korrekten Bedingungen liefern immer FALSE als Ergebnis?

1. `WHERE Name NOT = 'Meyer';`
2. `WHERE 1 = 2;`
3. `WHERE NOT Name LIKE 'M%';`
4. `WHERE Geburtsdatum LIKE '1980';`
5. `WHERE ID BETWEEN 20 AND 10;`
6. `WHERE Mobil IS NULL;`
7. `WHERE Name IS NULL;`
8. `WHERE Name STARTING WITH 'L' AND CONTAINING 'a';`
9. `WHERE ID IN (1, 3, 'A');`
10. `WHERE ID IN (1, 3, '5');`

Übung 6 – Suchbedingungen

Formulieren Sie die folgenden Aussagen als Bedingungen der WHERE-Klausel zur Tabelle *Mitarbeiter*.

1. Der Vorname lautet 'Petra'.
2. Der Name enthält die Zeichen 'mann'.
3. Der Name beginnt mit 'A', es handelt sich um Abteilung 8.
4. Es ist keine Mobil-Nummer gespeichert.

14.12. Lösungen

Lösung zu Übung 1 – Allgemeine Syntax

Richtig ist dies (mit Vervollständigung zu einem Befehl):

```
SELECT DISTINCT <spalten>
  FROM <tabelle>
 WHERE <bedingung>
 GROUP BY <spalten>
   HAVING <bedingung>
 ORDER BY <spalten>;
```

Lösung zu Übung 2 – Allgemeine Syntax

```
SELECT <spalten> FROM <tabelle>
```

Lösung zu Übung 3 – Spaltenliste

Richtig sind 1, 2, 5, 7, 8.

Falsch sind 3, 4 (ID ist mehrdeutig), 6 (Spaltenname und -Alias in falscher Reihenfolge).

Lösung zu Übung 4 – Spaltenliste

1. SELECT * FROM Mitarbeiter
2. SELECT Name, Vorname, Abteilung_ID FROM Mitarbeiter
3. SELECT Name, Vorname, Kuerzel FROM Mitarbeiter, Abteilung

oder mit Tabellen-Alias:

```
SELECT mi.Name, mi.Vorname, ab.Kuerzel FROM Mitarbeiter mi,
Abteilung ab
```

1. SELECT Name, Vorname, Dienstwagen.ID, Kennzeichen FROM
 Mitarbeiter, Dienstwagen

(auch einheitlich mit Tabellen-Namen oder Tabellen-Alias möglich)

Lösung zu Übung 5 – Suchbedingungen

1. Richtig.
2. Richtig, immer FALSE: 1 ist immer ungleich 2.
3. Falsch, das NOT gehört hinter <wert1>, also hinter *Name*.
4. Richtig, weil das Jahr laut ISO 8601 am Anfang steht.
5. Richtig, immer FALSE: es gibt keine Zahl „größer/gleich 20“ und gleichzeitig „kleiner/gleich 10“.
6. Richtig.
7. Richtig, immer FALSE, weil der *Name* als Pflichtangabe niemals NULL sein kann.
8. Falsch, weil *Name* in der zweiten Bedingung hinter AND fehlt.
9. Falsch, weil 'A' keine Zahl ist, aber zu *ID* bei IN eine Liste von Zahlen gehört.
10. Richtig, weil '5' automatisch als Zahl konvertiert wird.

Lösung zu Übung 6 – Suchbedingungen

1. WHERE Vorname = 'Petra';
2. WHERE Name CONTAINING 'mann';
3. WHERE Name STARTING WITH 'A' AND Abteilung_ID = 8;
4. WHERE Mobil IS NULL;

15. Funktionen (2)

Dieses Kapitel behandelt weitere **Skalarfunktionen** in Ergänzung zu den grundlegenden FUNKTIONEN¹.

15.1. Allgemeine Hinweise

Auch hier gelten die dort aufgeführten Hinweise:

- Jedes DBMS bietet eigene Funktionen sowie Varianten.
- Die Klammern werden in den Beschreibungen der Funktionen oft nicht angegeben.
- Die Beispiele werden durch eine verkürzte Schreibweise dargestellt, wobei der Zusatz "from fiktiv" als optional gekennzeichnet ist und für Firebird/Interbase durch "from rdb\$database" bzw. für Oracle durch "from dual" zu ersetzen ist.

```
SELECT 2 * 3 [from fiktiv];
```

15.2. Funktionen für Zahlen

Auch bei diesen weiteren Funktionen müssen Sie auf den Typ achten.

15.2.1. POWER, SQRT – Potenzen und Wurzeln

Mit **POWER** wird eine beliebige Potenz oder Wurzel berechnet:

```
POWER( <basis>, <exponent> )
```

1 Kapitel 13 auf Seite 121

Sowohl für <basis> als auch für <exponent> sind nicht nur ganze positive Zahlen, sondern alle Zahlen zulässig. Mit Dezimalzahlen als <exponent> werden (genau nach mathematischen Regeln) beliebige Wurzeln berechnet; in diesem Fall sind als <basis> negative Zahlen unzulässig. Beispiele:

```
SELECT POWER( 5, 3 ) [from fiktiv]; /* Ergebnis: 125,000 */
SELECT POWER( 5, 2.5) [from fiktiv]; /* Ergebnis: 55,902 */
SELECT POWER( 5, 0.5) [from fiktiv]; /* Ergebnis: 2,236
das ist Wurzel aus 5 */
SELECT POWER( 0.5, -3 ) [from fiktiv]; /* Ergebnis: 8,000
das ist 3.Potenz aus 2 */
SELECT POWER( 12.35, 1.5) [from fiktiv]; /* Ergebnis: 43,401 */
SELECT POWER(-12.35, 1.5) [from fiktiv]; /* expression evaluation
not supported. */
SELECT POWER( 12.35,-1.5) [from fiktiv]; /* Ergebnis: 0,023 */
SELECT POWER(-12.35,-1.5) [from fiktiv]; /* expression evaluation
not supported */
```

Mit **SQRT** (= Square Root) gibt es für die Quadratwurzel eine kürzere Schreibweise anstelle von **POWER(x,0.5)**:

```
SELECT SQRT(12.25) [from fiktiv]; /* Ergebnis: 3,500 */
```

15.2.2. EXP, LOG – Exponentialfunktion und Logarithmen

Mit **EXP** wird die Exponentialfunktion im engeren Sinne bezeichnet, also mit der Eulerschen Zahl e als Basis.

```
SELECT EXP(1) [from fiktiv]; /* Ergebnis:
2,71828182845905 */
```

Mit **LOG**(<wert>, <basis>) wird umgekehrt ein Logarithmus bestimmt, mit **LN** der natürliche und mit **LOG10** der dekadische Logarithmus.

```
SELECT LOG(10, EXP(1)) [from fiktiv]; /* Ergebnis:
2,30258509299405 */
SELECT LOG(10, 10) [from fiktiv]; /* Ergebnis: 1,000
*/
SELECT LN(10) [from fiktiv]; /* Ergebnis:
2,30258509299405 */
```

15.2.3. Winkelfunktionen

Die trigonometrischen Funktionen arbeiten mit dem Bogenmaß (nicht mit einer Grad-Angabe).

SIN	Sinus	
COS	Cosinus	
TAN	Tangens	
COT	Cotangens	
ASIN	Arcussinus	als Umkehrfunktion des Sinus
ACOS	Arcuscosinus	als Umkehrfunktion des Cosinus
ATAN	Arcustangens	als Umkehrfunktion des Tangens

Mit **DEGREES** wird ein Bogenmaß in Grad umgerechnet, mit **RADIANS** ein Gradmaß in das Bogenmaß.

PI liefert die entsprechende Zahl und kann auch für die trigonometrischen Funktionen verwendet werden:

```
SELECT SIN( PI()/6 ) [from fiktiv]; /*  $\pi/6$  sind  $30^\circ$ , also Ergebnis
0,5 */
```

15.2.4. ABS, RAND, SIGN – verschiedene Funktionen

Mit **ABS** wird der absolute Betrag der gegebenen Zahl zurückgegeben.

SIGN liefert als Hinweis auf das Vorzeichen der gegebenen Zahl einen der Werte 1, 0, -1 – je nachdem, ob die gegebene Zahl positiv, 0 oder negativ ist.

```
SELECT SIGN(12.34), SIGN(0), SIGN(-5.67) [from fiktiv];
/* Ergebnis: 1      0      -1      */
```

RAND liefert eine Zufallszahl im Bereich zwischen 0 und 1 (jeweils einschließlich). Bitte beachten Sie, dass dies keine echten Zufallszahlen sind, sondern Pseudozufallszahlen.

Mit **RAND(<vorgabewert>)** wird innerhalb einer Sitzung immer dieselbe Zufallszahl erzeugt.

Mit einer Kombination von **RAND** und **FLOOR** erhält man eine „zufällige“ Folge ganzer Zahlen:

```
FLOOR( <startwert> + ( RAND() * <zielwert> - <startwert> + 1 ) )
```

Beispielsweise liefert die mehrfache Wiederholung der folgenden Abfrage diese Zahlen zwischen 7 und 12:

```
SELECT FLOOR(7 + (RAND() * 6)) [from fiktiv];  
/* Ergebnisse: 10 9 9 7 8 7 9 9 10 12 usw. */
```

Diese Funktion ist geeignet, um Datensätze mit SELECT in beliebiger Reihenfolge abzurufen:

```
SELECT * FROM <tabelle> ORDER BY RAND();
```

15.3. Funktionen für Zeichenketten

Auch zur Bearbeitung und Prüfung von Zeichenketten (Strings) gibt es weitere Funktionen.

15.3.1. Verknüpfen von Strings

Zu den Standardverfahren `||` + `CONCAT` gibt es Ergänzungen.

MySQL bietet mit **CONCAT_WS** eine nützliche Erweiterung, bei der zwischen den Teiltexten ein Trennzeichen gesetzt wird.

SPACE(n) – für MS-SQL und MySQL – erzeugt einen String, der aus n Leerzeichen besteht.

REPEAT(<text>, <n>) – für My-SQL – und **REPLICATE**(<text>, <n>) – für MS-SQL – erzeugen eine neue Zeichenkette, in der der <text> n-mal wiederholt wird.

Mit **LPAD** wird <text1>, sofern erforderlich, auf die gewünschte <länge> gebracht und dabei von links mit <text2> bzw. Leerzeichen aufgefüllt. Mit **RPAD** wird von rechts aufgefüllt. *MS-SQL kennt diese Funktionen nur für Access.*

```
LPAD ( <text1>, <länge> [ , <text2> ] )
RPAD ( <text1>, <länge> [ , <text2> ] )
```

Wenn der dadurch erzeugte Text zu lang wird, wird zuerst <text2> und notfalls auch <text1> abgeschnitten. Beispiele:

```
SELECT LPAD( CAST(12345 AS CHAR(8)), 10, '0') [from fiktiv]; /*
Ergebnis: '0012345  ' */
```

Nanu, das sind doch nur 7 statt 10 Ziffern? Achso, zuerst wird mit CAST ein 8 Zeichen langer String erzeugt; dann ist nur noch Platz für 2 Nullen. Also muss es mit einer dieser Varianten gehen:

```
SELECT LPAD( CAST(12345 AS CHAR(5)), 10, '0') [from fiktiv]; /*
Ergebnis: '0000012345' */
SELECT LPAD( 12345, 10, '0' ) [from fiktiv]; /*
Ergebnis: '0000012345' */
SELECT LPAD( 'Hilfe', 10, '-_/' ) [from fiktiv]; /*
Ergebnis: '-_/_Hilfe' */
SELECT LPAD( 'Ich brauche Hilfe', 10, '-_/' ) [from fiktiv]; /*
Ergebnis: 'Ich brauch' */
SELECT RPAD( 'Hilfe', 10, '-_/' ) [from fiktiv]; /*
Ergebnis: 'Hilfe-_/_' */
SELECT RPAD( 'Ich brauche Hilfe', 10, '-_/' ) [from fiktiv]; /*
Ergebnis: 'Ich brauch' */
```

15.3.2. LEFT, RIGHT – Teile von Zeichenketten

Als Ergänzung zu SUBSTRING wird mit **LEFT**(<text>, <anzahl>) der linke Teil, also der Anfang eines Textes mit der gewünschten Länge <anzahl> ausgegeben. Ebenso erhält man mit **RIGHT**(<text>, <anzahl>) den rechten Teil, also das Ende eines Textes.

```
SELECT LEFT ('Abc Def Ghi', 5) [from fiktiv]; /* Ergebnis: 'Abc D'
*/
SELECT RIGHT('Abc Def Ghi', 5) [from fiktiv]; /* Ergebnis: 'f Ghi'
*/
```

15.3.3. TRIM, LTRIM, RTRIM – Leerzeichen u. a. entfernen

Mit der **TRIM**-Funktion werden bestimmte Zeichen – meistens Leerzeichen – am Anfang und/oder am Ende eines Textes entfernt:

```
TRIM( [ [ LEADING | TRAILING | BOTH ] [ <zeichen> ] FROM ] <text> )
```

Die Parameter werden wie folgt benutzt:

- Es soll <zeichen> entfernt werden. Es kann sich um ein einzelnes Zeichen, aber auch um einen Text handeln.
- Wenn dieser Parameter fehlt, wird nach führenden bzw. abschließenden Leerzeichen gesucht.
- Wenn LEADING angegeben ist, werden nur führende Zeichen geprüft, bei TRAILING nachfolgende und bei BOTH sowohl als auch. Wenn nichts davon angegeben wird, wird BOTH angenommen.

Beispiele:

```
SELECT TRIM( ' Dies ist ein Text. ' ) [from fiktiv]; /*
Ergebnis: 'Dies ist ein Text.' */
SELECT TRIM( LEADING 'a' FROM 'abcde' ) [from fiktiv]; /*
Ergebnis: 'bcde' */
SELECT TRIM( TRAILING 'e' FROM 'abcde' ) [from fiktiv]; /*
Ergebnis: 'abcd' */
SELECT TRIM( 'Test' FROM 'Test als Test') [from fiktiv]; /*
Ergebnis: ' als ' */
```

LTRIM (= Left-Trim) und **RTRIM** (= Right-Trim) sind Kurzfassungen, bei denen Leerzeichen am Anfang bzw. am Ende entfernt werden; MS-SQL kennt nur diese beiden Kurzfassungen.

15.3.4. Suchen und Ersetzen

Mit **POSITION** wird der Anfang eines Textes innerhalb eines anderen gesucht.

```
POSITION( <text1> IN <text2> )           SQL-Standard
POSITION( <text1>, <text2> [, <start>] ) nicht bei MySQL
LOCATE ( <text1>, <text2> [, <start>] )  bei MySQL
```

Bei MS-SQL gibt es diese Funktionen nicht; stattdessen kann (mit abweichender Bedeutung) **PATINDEX** verwendet werden. Oracle bietet zusätzlich **INSTR** (= "in string") an.

Die Bedeutung der Parameter dürfte offensichtlich sein:

- <text2> ist der Text, in dem gesucht werden soll.
- <text1> ist ein Teiltex, der in <text2> gesucht wird.
- Sofern <start> angegeben ist, wird erst ab dieser Position innerhalb <text2> gesucht. Wenn <start> fehlt, wird ab Position 1 gesucht.
- Die Funktion gibt die Startposition von <text1> innerhalb von <text2> an. Wenn <text1> nicht gefunden wird, lautet das Ergebnis 0.

Beispiele:

```
SELECT POSITION( 'ch', 'Ich suche Text' )    [from fiktiv]; /*
Ergebnis: 2 */
SELECT POSITION( 'ch', 'Ich suche Text', 3 ) [from fiktiv]; /*
Ergebnis: 7 */
SELECT POSITION('sch', 'Ich suche Text' )    [from fiktiv]; /*
Ergebnis: 0 */
```

REPLACE dient zum Ersetzen eines Teiltexes durch einen anderen innerhalb eines Gesamttextes:

```
REPLACE( <quelltext>, <suche>, <ersetze> )
```

Die verschiedenen SQL-Dialekte verhalten sich unterschiedlich, ob NULL-Werte oder leere Zeichenketten zulässig sind.

```
SELECT REPLACE('Ich suche Text', 'ch', 'sch') [from fiktiv]; /*
Ergebnis: 'Isch susche Text' */
SELECT REPLACE('Die liebe Seele', 'e', '')    [from fiktiv]; /*
Ergebnis: 'Di lib Sl' */
```

REVERSE passt zwar nicht zu dem, was man in diesem Zusammenhang erwartet; aber auch diese Funktion ändert einen vorhandenen String, und zwar dadurch, dass die Reihenfolge aller Zeichen umgekehrt wird:

```
SELECT REVERSE( 'Hilfe' )                    [from fiktiv]; /*
Ergebnis: 'efliH' */
```

15.4. Funktionen für Datums- und Zeitwerte

Bitte beachten Sie wiederum die Besonderheiten der Datentypen je nach DBMS.

15.4.1. Differenzen bei Datum oder Uhrzeit

Dafür gibt es vorzugsweise die **DATEDIFF**-Funktion in unterschiedlicher Version:

```
DATEDIFF ( <part>, <start>, <end> ) /* bei MS-SQL oder Firebird */
DATEDIFF ( <start>, <end> ) /* bei MySQL nur die Anzahl der
Tage */
```

Das Ergebnis ist vom gleichen Typ wie die gesuchte Differenz (also meistens ein ganzzahliger Wert). Als <part> gibt es die gleichen Varianten wie bei den wichtigsten Funktionen:

```
YEAR | MONTH | DAY | HOUR | MINUTE | SECOND | MILLISECOND
```

Beim Vergleich von Start- und Enddatum gilt:

- Das Ergebnis ist positiv, wenn der zweite Wert größer ist als der erste.
- Das Ergebnis ist 0, wenn beide Werte gleich sind.
- Das Ergebnis ist negativ, wenn der zweite Wert kleiner ist als der erste.

Bitte beachten Sie: Die DBMS verhalten sich unterschiedlich, ob die Datumangaben verglichen werden oder der jeweilige Bestandteil. Beispielsweise kann das Ergebnis für beide der folgenden Prüfungen 1 lauten, obwohl die „echte“ Differenz im einen Fall ein Tag, im anderen fast 2 Jahre sind:

```
DATEDIFF( YEAR, '31.12.2008', '01.01.2009' )
DATEDIFF( YEAR, '01.01.2008', '31.12.2009' )
```

➤ **Aufgabe:** Bestimme die Anzahl der Tage seit dem letzten gemeldeten Schadensfall.

```
SELECT DATEDIFF(DAY, MAX(Datum), CURRENT_DATE)
FROM Schadensfall; /* Ergebnis: 49 */
```


► **Aufgabe:** Bestimme die Anzahl der Minuten seit Tagesbeginn.

```
SELECT DATEDIFF(MINUTE, CAST('00:00' AS TIME), CURRENT_TIME) [from
fiktiv]; /* Ergebnis: 967 */
```

Datumsangaben können grundsätzlich auch per Subtraktion verglichen werden, weil „intern“ häufig ein Tag gleich 1 ist. Darauf kann man sich aber nicht immer verlassen; und es ist schwierig, die Bruchteile eines Tages zu berücksichtigen. Beispiel:

```
SELECT (CURRENT_DATE - MAX(Datum)) FROM Schadensfall; /*
Ergebnis: 49 */
```

15.4.2. Werte für Datum oder Uhrzeit ändern

Sehr häufig muss aus einem vorhandenen Datum oder Uhrzeit ein neuer Wert berechnet werden. Der SQL-Standard sieht dazu die direkte **Addition und Subtraktion** vor:

```
<datetime> + <value>
<datetime> - <value>
<value> + <datetime>
```

<datetime> steht für den gegebenen Wert, <value> für den Zeitraum, der addiert oder subtrahiert werden soll.

► **Aufgabe:** Aus dem aktuellen Zeitwert '19.09.2009 16:10' wird ein neuer Wert bestimmt, der einen halben Tag in der Zukunft liegt:

```
SELECT CURRENT_TIMESTAMP + 0.5 [from fiktiv]; /* Ergebnis:
'20.09.2009 04:10:39' */
```

MySQL akzeptiert nur ganze Zahlen; deshalb ist explizit die Art des Intervalls anzugeben (siehe Dokumentation).

Da das Umrechnen von Zahlen in Datums- und Zeitwerte und umgekehrt für den Anwender umständlich ist, werden viele zusätzliche Funktionen bereitgestellt. Sehr verbreitet ist **DATEADD**:

```
DATEADD( <part>, <value>, <datetime> )           /*
Firebird, MS-SQL */
DATE_ADD( <datetime> , INTERVAL <value> <part> )   /* MySQL
*/
```

► **Aufgabe:** Welche Versicherungsverträge laufen schon mehr als 10 Jahre?

```
SELECT ID, Vertragsnummer, Abschlussdatum FROM Versicherungsvertrag
WHERE DATEADD(YEAR, 10, Abschlussdatum) <= CURRENT_DATE; /*
Ergebnis: 18 Datensätze */
```

Als Ergänzung oder Alternative gibt es weitere Funktionen, beispielsweise DATE_SUB als Subtraktion, ADDDATE oder ADD_MONTHS.

15.5. Funktionen für logische und NULL-Werte

Neben den Standardprüfungen vor allem bei der WHERE-Klausel (siehe nächstes Kapitel) und den Operatoren AND, OR, NOT gibt es weitere Prüfungen.

15.5.1. COALESCE – Suche Wert ungleich NULL

Die **COALESCE**-Funktion sucht in einer Liste von Werten (bzw. Ausdrücken) den ersten, der nicht NULL ist. Wenn alle Werte NULL sind, ist der Rückgabewert (zwangsläufig) NULL.

► **Aufgabe:** Nenne zu jedem Mitarbeiter eine Kontaktmöglichkeit: vorzugsweise Mobilnummer, dann Telefonnummer, dann Email-Adresse.

```
SELECT Name, Vorname, COALESCE(Mobil, Telefon, Email) AS Kontakt FROM
Mitarbeiter;
```

Das Ergebnis überrascht zunächst, denn einige Mitarbeiter hätten danach keine Kontaktmöglichkeit. Bei der Abfrage nach IS NULL zur WHERE-Klausel wird aber erläutert, dass eine leere Zeichenkette ungleich NULL ist; bei diesen Mitarbeitern wird also ein leerer Eintrag, aber nicht "nichts" angezeigt.

- Bitte nehmen Sie diesen Hinweis als Empfehlung, lieber NULL zu speichern als den leeren String ”.

15.5.2. NULLIF

Die Funktion **NULLIF** vergleicht zwei Werte und liefert NULL zurück, wenn beide Werte gleich sind; andernfalls liefert der erste Wert das Ergebnis.

► **Aufgabe:** Suche alle Versicherungsnehmer, die im Alter von 18 Jahren ihren Führerschein gemacht haben.

```
select Name, Vorname, EXTRACT(YEAR from Geburtsdatum) + 18 as Jahr
  from Versicherungsnehmer
 where Geburtsdatum is not null
    and NULLIF( EXTRACT(YEAR from Geburtsdatum) + 18, EXTRACT(YEAR from
Fuehrerschein) ) is null;
```

NAME	VORNAME	JAH
Liebermann	Maria	1988

Hinweis: Sowohl COALESCE als auch NULLIF sind Kurzfassungen für spezielle Fallunterscheidungen mit CASE WHEN, in denen zusätzlich IS NULL eingebunden wird – siehe dazu NÜTZLICHE ERWEITERUNGEN².

15.6. Verschiedene Funktionen

Auch wenn die Funktionen in diesem Abschnitt beim SQL-Standard vorgesehen sind, sind sie nicht immer vorhanden. Wir verzichten deshalb wiederum auf nähere Erläuterungen und verweisen auf die jeweilige Dokumentation.

15.6.1. ROW_NUMBER – Zeilen nummerieren

Mit der **ROW_NUMBER**-Funktion werden die Zeilen im Ergebnis einer Abfrage nach der betreffenden Sortierung durchnummeriert. MS-SQL verwendet die folgende Syntax:

```
ROW_NUMBER() OVER ( [ <partition_by_clause> ] <order_by_clause> )
```

2 Kapitel 22 auf Seite 237

15.6.2. CURRENT_USER – der aktuelle Benutzer

Mit **CURRENT_USER** (in der Regel ohne Klammer) wird der aktuelle Benutzername abgefragt. Dieser kann auch per DEFAULT bei Neuaufnahmen automatisch in einer Spalte einer Tabelle eingetragen werden.

```
SELECT CURRENT_USER [from fiktiv]; /* Ergebnis: SYSDBA */
```

15.7. Zusammenfassung

In diesem Kapitel lernten Sie weitere eingebaute Funktionen kennen:

- Für Zahlen gibt es viele mathematische Funktionen wie Potenzen, Wurzeln, Exponential- oder Winkelfunktionen.
- Zeichenketten können auf vielfache Weise verknüpft oder zum Erstellen neuer Strings bearbeitet werden.
- Datums- und Zeitwerte können im Detail verglichen und verrechnet werden.

15.8. Übungen

Übung 1 – Funktionen für Zahlen

Geben Sie mit SQL-Funktionen die Formeln für die folgenden Aufgaben an. *Es geht nur um die Formeln, nicht um einen passenden SELECT-Befehl. Bei den Aufgaben 1 bis 3 sind jeweils zwei Lösungen möglich.*

1. Gegeben seien zwei Zahlen a, b. Berechnen Sie $a^2 + 2ab + b^2$.
2. Berechnen Sie die Quadratwurzel von 216,09.
3. Ein Auftragsverwaltungsprogramm speichert in der Spalte *Bestellung* die Bestellwerte und in der Spalte *Zahlung* die Einzahlungen. Bestimmen Sie mit einem einzigen Ausdruck die Prüfung, ob der Kunde Guthaben oder Schulden (Zahlungsverpflichtung) hat.
4. Bestimmen Sie den Betrag von Aufgabe 3 (unabhängig davon, ob es sich um Guthaben oder Schulden handelt).

Übung 2 – Zufallszahlen

Bestimmen Sie mit SQL-Funktionen die folgenden Zufallszahlen.

1. eine beliebige Zufallszahl zwischen 0 und 1
2. eine beliebige Zufallszahl zwischen 0 und 4
3. eine beliebige Zufallszahl zwischen 1 und 5
4. eine Zufallszahl als ganze Zahl zwischen 1 und 5
5. eine Zufallszahl als ganze Zahl zwischen 1 und 26
6. einen zufällig ausgewählten Buchstaben aus der Liste *letters* der Großbuchstaben 'ABC...XYZ'

Übung 3 – Zeichenketten bearbeiten

Aus der Tabelle *Mitarbeiter* sollen die Werte *ID* und *Abteilung_ID* zusammengesetzt werden. Dabei soll die *ID* immer 4-stellig und die *Abteilung_ID* immer 2-stellig geschrieben werden, bei Bedarf sollen die Teile mit '0' aufgefüllt werden.

Übung 4 – Zeichenketten bearbeiten

Geben Sie für die Tabelle *Mitarbeiter* eine der vorhandenen Telefonnummern an – vorrangig die Mobilnummer; berücksichtigen Sie dabei auch, ob überhaupt eine Nummer gespeichert ist.

Übung 5 – Zeichenketten bearbeiten

Zeigen Sie für die Spalte *Kennzeichen* der Tabelle *Fahrzeug* den zugehörigen Kreis an.

Übung 6 – Zeichenketten bearbeiten

In der *Beschreibung* für den *Schadensfall* mit der ID 6 ist das Kennzeichen 'RE-LM 903' durch 'RE-LM 902' zu berichtigen.

Übung 7 – Datum und Zeit bearbeiten

Die folgenden Teilaufgaben werden benutzt, um im Kapitel TESTDATEN ERZEUGEN³ Geburtsdatum, Führerschein-Erwerb oder Abschluss des Versicherungsvertrags zufällig zu bestimmen.

1. Bestimmen Sie aus dem *Geburtsdatum* das Datum des 18. Geburtstags.

3 Kapitel 35 auf Seite 455

- Bestimmen Sie (ausgehend vom 01.01.1950) Datumsangaben bis zum 31.12.1990, bei denen der Monat und das Jahr per Zufallsfunktion bestimmt werden.
- Bestimmen Sie ebenso Datumsangaben, bei denen auch der Tag zufällig festgelegt wird und immer ein gültiges Datum erzeugt wird.
- Prüfen Sie, ob ein neuer Kunde seinen Führerschein bei Vertragsabschluss bereits drei Jahre besitzt. Sie können annehmen, dass sowohl *Fuehrerschein* als auch *Abschlussdatum* in derselben Tabelle liegen; Schaltjahre können ignoriert werden.

15.9. Lösungen

Lösung zu Übung 1 – Funktionen für Zahlen

- `POWER(a, 2) + 2*a*b + POWER(b, 2)`
 - `POWER(a + b, 2)` *als einfachste Binomische Formel*
- `SQRT(216.09)`
 - `POWER(216.09, 0.5)`
- `IF (SUM(Bestellung) > SUM(Zahlung))`
 - `IF(SIGN(SUM(Bestellung) - SUM(Zahlung)) = 1)`
- `ABS(SUM(Bestellung) - SUM(Zahlung))`

Lösung zu Übung 2 – Zufallszahlen

- `RAND()`
- `RAND() * 4`
- `1 + RAND() * 4`
- `FLOOR(1 + RAND() * 4)`
- `FLOOR(1 + RAND() * 25)`
- `SUBSTRING(letters FROM FLOOR(1 + RAND() * 25) FOR 1)`

Lösung zu Übung 3 – Zeichenketten bearbeiten

```
SELECT LPAD( ID, 4, '0') + LPAD( Abteilung_ID, 2, '0') FROM
Mitarbeiter;
```

Lösung zu Übung 4 – Zeichenketten bearbeiten

```
SELECT COALESCE( NULLIF(Mobil, ''), Telefon) AS Tel FROM Mitarbeiter;
```

Erklärung: Wenn *Mobil* einen Wert enthält, kommt bei NULLIF dieser Wert heraus; andernfalls wird immer NULL geliefert – entweder als Feldinhalt oder als Ergebnis des Vergleichs mit der leeren Zeichenkette.

Lösung zu Übung 5 – Zeichenketten bearbeiten

```
SELECT SUBSTRING( Kennzeichen FROM 1 FOR Position('-', Kennzeichen) - 1) FROM Fahrzeug;
```

Lösung zu Übung 6 – Zeichenketten bearbeiten

```
UPDATE Schadensfall
   SET Beschreibung = REPLACE( Beschreibung, 'RE-LM 903', 'RE-LM 902'
 )
 WHERE ID = 6;
```

Lösung zu Übung 7 – Datum und Zeit bearbeiten

1. DATEADD(YEAR, 18, Geburtsdatum)
2. DATEADD(MONTH, FLOOR(RAND()*11), DATEADD(YEAR, FLOOR(RAND()*40), '1950-01-01'))

Es handelt sich um eine verschachtelte Funktion: Zuerst wird das Jahr neu bestimmt, es wird maximal ein Wert von 40 addiert. Zu diesem Ergebnis wird der Monat neu bestimmt, es wird maximal 11 addiert. In manchen Fällen sind einzelne Angaben per CAST genauer zu definieren.

3. DATEADD(DAY, FLOOR(RAND()*(41*365+10)), '1950-01-01')

Es handelt sich um 41 Jahre zu je 365 Tagen, dazu 10 Schalttage.

4. SELECT Fuehrerschein, Abschlussdatum FROM (Vertrag/Kunde)
 WHERE DATEDIFF(DAY, Fuehrerschein, Abschlussdatum) < 3*365

Hinweis: DATEDIFF(YEAR...) ist nicht unbedingt geeignet, weil ein DBMS nicht die Termine, sondern die Jahreszahlen vergleichen könnte.

15.10. Siehe auch

Bei Wikipedia gibt es fachliche Erläuterungen:

- POTENZEN⁴
- EXPONENTIALFUNKTION⁵
- TRIGONOMETRISCHE FUNKTIONEN⁶ und BOGENMASS⁷
- BETRAGSFUNKTION⁸ – der absolute Betrag
- ZUFALLSZAHL⁹ gibt auch Hinweise zu „Pseudozufallszahlen“.
- BINOMISCHE FORMELN¹⁰

4 [HTTP://DE.WIKIPEDIA.ORG/WIKI/POTENZ%20%28MATHEMATIK%29](http://de.wikipedia.org/wiki/Potenz%20%28Mathematik%29)

5 [HTTP://DE.WIKIPEDIA.ORG/WIKI/EXPONENTIALFUNKTION](http://de.wikipedia.org/wiki/Exponentialfunktion)

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/TRIGONOMETRISCHE%20FUNKTIONEN](http://de.wikipedia.org/wiki/Trigonometrische%20Funktionen)

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/BOGENMAß](http://de.wikipedia.org/wiki/Bogenmaß)

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/BETRAGSFUNKTION](http://de.wikipedia.org/wiki/Betragsfunktion)

9 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ZUFALLSZAHL](http://de.wikipedia.org/wiki/Zufallszahl)

10 [HTTP://DE.WIKIPEDIA.ORG/WIKI/BINOMISCHE%20FORMEL](http://de.wikipedia.org/wiki/Binomische%20Formel)

16. WHERE-Klausel im Detail

In diesem Kapitel werden die Einzelheiten der WHERE-Klausel genauer behandelt. Diese Angaben sind vor allem für den SELECT-Befehl, aber auch für UPDATE und DELETE von Bedeutung.

Die Beispiele beziehen sich auf den Anfangsbestand der Beispieldatenbank; auf die Ausgabe der selektierten Datensätze wird verzichtet. Bitte probieren Sie alle Beispiele aus und nehmen Sie verschiedene Änderungen vor, um die Auswirkungen zu erkennen.

16.1. Allgemeine Hinweise

Die **WHERE**-Klausel ist (neben der Verknüpfung mehrerer Tabellen) der wichtigste Bestandteil des SELECT-Befehls: Je sorgfältiger die Auswahlbedingungen formuliert werden, desto genauer ist das Ergebnis der Abfrage.

Neben den hier erläuterten Varianten bietet jedes DBMS noch andere, z. B. STARTING WITH oder SIMILAR.

Anstelle konstanter Werte können auch passende Ausdrücke angegeben werden, z. B. Funktionen oder Unterabfragen.

Verwenden Sie bei den Beispielen möglichst immer auch die Umkehrung der Auswahl mit bzw. ohne NOT. Wie im Kapitel „Ausführliche SELECT-Struktur“ zur WHERE-Klausel angegeben, steht das NOT ggf. unmittelbar vor dem Parameter-Namen.

16.2. Eine einzelne Bedingung

16.2.1. Größenvergleich zweier Werte

Der einfachste Weg ist der direkte Vergleich zweier Werte, nämlich der Inhalt einer Spalte mit einem konstanten Wert. Dies ist möglich mit den folgenden

Vergleichsoperatoren, und zwar für alle Datentypen, die verglichen werden können – Zahlen, Zeichenketten, Datumsangaben.

```
= < > <= >= <>
```

Beispiele: ► **Aufgabe:** Suche einen Datensatz, bei dem der Wert in der Spalte *ID* gleich ist zu einem vorgegebenen Wert.

```
select * from Versicherungsnehmer
where ID = 10;
```

► **Aufgabe:** Suche Datensätze, bei denen der Name kleiner als 'B' ist, also mit 'A' anfängt.

```
select * from Versicherungsnehmer
where Name < 'B';
```

► **Aufgabe:** Suche Führerschein-Neulinge.

```
select * from Versicherungsnehmer
where Fuehrerschein >= '01.01.2007';
```

► **Aufgabe:** Suche Fahrzeugtypen mit kurzer Bezeichnung.

```
select * from Fahrzeugtyp
where Char_Length(Bezeichnung) <= 3;
```

Bei diesen Vergleichen ist NOT zwar ebenfalls möglich; besser verständlich ist aber ein anderer passender Operator.

16.2.2. BETWEEN AND – Werte zwischen zwei Grenzen

Mit der Bedingung **BETWEEN** <wert1> **AND** <wert2> wird direkt mit einem Bereich verglichen; die Grenzwerte gehören zum Bereich. 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';
```

16.2.3. LIKE – Ähnlichkeiten (1)

Die **LIKE**-Bedingung vergleicht Zeichenketten „ungenau“: Der gesuchte Text soll als Wert in einer Spalte enthalten sein; dazu werden „Wildcard“ 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: ► **Aufgabe:** Der Ortsname beginnt nicht mit 'B'; der Inhalt dahinter ist beliebig.

```
select * from Versicherungsnehmer
where Ort NOT LIKE 'B%';
```

► **Aufgabe:** Der Ortsname enthält irgendwo 'alt' mit beliebigem Inhalt davor und dahinter.

```
select * from Versicherungsnehmer
where Ort LIKE '%alt%';
```

► **Aufgabe:** Der Anfangsbuchstabe des Namens ist unklar, aber danach folgen 'ei'.

```
select * from Versicherungsnehmer
where Name LIKE '_ei%';
```

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: ► **Aufgabe:** Innerhalb der Beschreibung kommt die Zeichenfolge '10%' vor.

```
select * from Schadensfall
where Beschreibung LIKE '%10\%%' ESCAPE '\';
```

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.

16.2.4. CONTAINS u. a. – Ähnlichkeiten (2)

Ein Problem des LIKE-Parameters ist die Verwendung der Wildcard-Zeichen '%' und '_', die man gerne vergisst oder (wie im letzten Beispiel) nicht genau genug beachtet. Deshalb gibt es verschiedene Vereinfachungen.

CONTAINS – in Firebird **CONTAINING** – prüft, ob eine Zeichenkette im Feldinhalt enthalten ist.

```
select * from Schadensfaelle
where Beschreibung CONTAINS '10%';
```

Teil der Beschreibung ist die Zeichenkette '10%'

Bitte prüfen Sie in der Beschreibung Ihres DBMS, welche Möglichkeiten für die Suche nach Ähnlichkeiten außerdem angeboten werden.

16.2.5. IS NULL – null-Werte prüfen

Wie schon bei den relationalen Datenbanken besprochen, haben NULL-Werte 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

Nanu, es gibt doch 28 Mitarbeiter; wo sind die übrigen geblieben? 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 überhaupt kein Ergebnis, 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 Mobil-Angabe 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 = " );
```

20 Mitarbeiter ohne ausdrückliche Angabe

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.)

16.2.6. 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. ► **Aufgabe:** Hole die Liste aller Fahrzeuge, deren Typen als „VW-Kleinwagen“ registriert sind.

```
select * from Fahrzeug
where Fahrzeugtyp_ID in (1, 2);
```

► **Aufgabe:** Suche nach einem Unfall Fahrzeuge mit einer von mehreren möglichen Farben.

```
select * from Fahrzeug
where Farbe in ('ocker', 'gelb');
```

Vor allem das erste Beispiel wird sehr oft mit einer Unterabfrage versehen; vergleichen Sie dazu auch den folgenden Abschnitt zu EXISTS. ► **Aufgabe:** Hole die Liste aller Fahrzeuge vom Typ „Volkswagen“.

```
select * from Fahrzeug
where Fahrzeugtyp_ID in
( select ID from Fahrzeugtyp
  where Hersteller_ID = 1 );
```

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.

16.2.7. 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.*

16.3. Mehrere Bedingungen verknüpfen

Bei der WHERE-Klausel geht es darum festzustellen, ob ein bestimmter 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**.

16.3.1. NOT als Negation

Dieser Operator kehrt das Ergebnis um: aus TRUE wird FALSE, aus FALSE wird TRUE.

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

Siehe oben: keine Führerschein-Neulinge

16.3.2. AND als Konjunktion

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

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

Die nach Alphabet erste Hälfte der Versicherungsnehmer eines PLZ-Bereichs

16.3.3. 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.

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

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

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**.

16.3.4. XOR als Kontravalenz

Eine Bedingung, die durch eine XOR-Verknüpfung zweier Bedingungen 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.*

MySQL Quelltext

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

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

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> ) )
```

16.3.5. Klammern benutzen oder weglassen?

Bereits im Kapitel „Ausführliche SELECT-Struktur“ wurde die Hierarchie genannt:

- NOT ist die engste Verbindung und wird vorrangig ausgewertet.
- AND ist die nächststärkere Verbindung und wird danach ausgewertet.
- OR ist die schwächste Verbindung und wird zuletzt ausgewertet.
- 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 feststellen. 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.

16.4. 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.

16.5. Übungen

Bei den folgenden Aufgaben kommt es nur auf die WHERE-Klausel an; Sie dürfen ein SELECT „mit allen Spalten“ benutzen.

Übung 1 – 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...).

Übung 2 – Auswahl nach Datumsbereich

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

Übung 3 – Auswahl nach Ähnlichkeit

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

Übung 4 – Auswahl für unbekannte Werte

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

Hinweis: Die Prüfung „Mitarbeiter ohne Dienstwagen“ ist komplizierter; das dafür erforderliche OUTER JOIN wird erst später behandelt.

Übung 5 – 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.

Übung 6 – Bedingungen verknüpfen

Gesucht werden die Versicherungsverträge für Haftpflicht (= 'HP') und Teilka-sko (= 'TK'), die mindestens seit dem Ende des Jahres 1980 bestehen und aktuell nicht mit dem minimalen Prämiensatz berechnet werden.

Tragen Sie ausnahmsweise nur die notwendigen Klammern ein, nicht alle sinnvollen.

16.6. Lösungen

Lösung zu Übung 1 – Auswahl nach Zeichenketten

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

```
and PLZ STARTING WITH '45' /* oder: */
    PLZ like '45%';
```

Lösung zu Übung 2 – Auswahl nach Datumsbereich

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

Lösung zu Übung 3 – Auswahl nach Ähnlichkeit

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

Lösung zu Übung 4 – Auswahl für unbekannte Werte

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

Lösung zu Übung 5 – Bedingungen verknüpfen

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

Lösung zu Übung 6 – Bedingungen verknüpfen

```
SELECT * from Versicherungsvertrag
where (Art = 'HP' or Art = 'TK')
    and Abschlussdatum <= '31.12.1980'
    and (not Praemiensatz = 30) /* oder */
    and Praemiensatz > 30;
```

16.7. Siehe auch

Dieses Kapitel verweist auf die folgenden Kapitel:

- FUNKTIONEN¹
- UNTERABFRAGEN²
- AUSFÜHRLICHE SELECT-STRUKTUR³
- RELATIONALE DATENBANKEN⁴
- OUTER JOIN⁵

Bei Wikipedia gibt es weitere fachliche Hinweise:

- WILDCARDS⁶
- ESCAPE-ZEICHEN⁷
- BOOLESCHER OPERATOREN⁸
 - NEGATION⁹
 - KONJUNKTION¹⁰
 - ADJUNKTION¹¹
 - KONTRAVALENZ¹²

1 Kapitel 13 auf Seite 121

2 Kapitel 25 auf Seite 283

3 Kapitel 14 auf Seite 145

4 Kapitel 3 auf Seite 17

5 Kapitel 20 auf Seite 211

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/WILDCARD%20%28INFORMATIK%29](http://de.wikipedia.org/wiki/Wildcard%20%28Informatik%29)

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ESCAPE-{}SEQUENZ](http://de.wikipedia.org/wiki/Escape-{}sequenz)

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/BOOLESCHER%20OPERATOR](http://de.wikipedia.org/wiki/Boolescher%20operator)

9 [HTTP://DE.WIKIPEDIA.ORG/WIKI/NEGATION%23LOGIK](http://de.wikipedia.org/wiki/Negation%23logik)

10 [HTTP://DE.WIKIPEDIA.ORG/WIKI/KONJUNKTION%20%28LOGIK%29](http://de.wikipedia.org/wiki/Konjunktion%20%28Logik%29)

11 [HTTP://DE.WIKIPEDIA.ORG/WIKI/DISJUNKTION](http://de.wikipedia.org/wiki/Disjunktion)

12 [HTTP://DE.WIKIPEDIA.ORG/WIKI/DISJUNKTION](http://de.wikipedia.org/wiki/Disjunktion)

17. Mehrere Tabellen

Ein besonderes Merkmal von relationalen Datenbanken und damit von SQL ist, dass die Informationen fast immer über mehrere Tabellen verteilt sind und bei Abfragen in der Ergebnismenge zusammengeführt werden müssen. Dieses Kapitel gibt einen Überblick über die Möglichkeiten dazu; Einzelheiten stehen in den folgenden Kapiteln.

17.1. Schreibweisen bei mehreren Tabellen

Bitte beachten Sie bei allen Befehlen, die mehrere Tabellen verwenden (das sind zwangsläufig nur SELECT-Befehle):

- 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 Tabellenname vorangesetzt 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 der Tabellenname in der Regel auch dann bei jeder Spalte angegeben, wenn es wegen der ersten Regel nicht erforderlich wäre.

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

- 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;
```

Alle diese Befehle, mit denen man „Alle Kombinationen“ von Mitarbeitern und Dienstwagen erhält, sind gleichwertig. Zu empfehlen ist die vollständige Schreibweise mit Alias wie im vorigen Beispiel.

Ein ähnlicher Befehl unter Verwendung der JOIN-Klausel sieht 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
  JOIN Dienstwagen dw on mi.ID = dw.Mitarbeiter_ID;
```

Der Alias kann auch aus einem einzelnen Buchstaben bestehen und ist nur für den betreffenden SQL-Befehl gültig. Ein und dieselbe Tabelle kann mal als 'a', dann als 'mi' oder auch als 'xyz' bezeichnet werden. Wegen des leichteren Verständnisses sind aussagefähige Kürzel sinnvoll; auch deshalb sind sie im Kapitel TABELLENSTRUKTUR DER BEISPIELDATENBANK¹ angegeben.

17.2. Verknüpfung über WHERE – der traditionelle Weg

Beim einfachsten Verfahren, mehrere Tabellen gleichzeitig abzufragen, werden in der FROM-Klausel alle Tabellen aufgeführt; die WHERE-Klausel enthält neben den Auswahlbedingungen auch Verknüpfungsbedingungen, wie die Tabellen zueinander gehören.

Einzelheiten werden in EINFACHE TABELLENVERKNÜPFUNG² behandelt.

1 Kapitel 36 auf Seite 469

2 Kapitel 18 auf Seite 193

17.3. JOINs - der moderne Weg

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

Die Einführung dazu wird in ARBEITEN MIT JOIN³ besprochen.

17.4. 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.

Einzelheiten dazu werden in OUTER JOIN⁴ behandelt.

17.5. 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“.

Diese Ergänzungen werden in MEHR ZU JOIN⁵ besprochen.

17.6. Zusammenfassung

In diesem Kapitel erhielten wir Hinweise darauf, wie mehrere Tabellen verknüpft werden können:

- einfach über die FROM-Klausel und passende WHERE-Bedingungen
- übersichtlich über die JOIN-Klausel mit verschiedenen Varianten

3 Kapitel 19 auf Seite 201

4 Kapitel 20 auf Seite 211

5 Kapitel 21 auf Seite 225

17.7. Übungen

Übung 1 – Was ist an diesem SELECT-Befehl falsch?

Zeigen Sie zu bestimmten Versicherungsverträgen die Daten der Fahrzeuge an.

```
select ID, Abschlussdatum, Art,  
       vv.Kennzeichen, Farbe  
from Versicherungsvertrag vv, Fahrzeug  
where vv.Fahrzeug_ID = Fahrzeug.ID  
      and Kennzeichen LIKE 'BO%';
```

Übung 2 – Was ist an diesem SELECT-Befehl falsch?

Zeigen Sie zu einem Versicherungsvertrag die Daten des Versicherungsnehmers und des Sachbearbeiters an.

```
select ID, Vorname + ' ' + Name AS Kunde, Ort  
       Name AS Sachbearbeiter, Telefon  
from Versicherungsvertrag, Versicherungsnehmer, Mitarbeiter  
where ID = 27  
      and Versicherungsvertrag.Versicherungsnehmer_ID =  
      Versicherungsnehmer.ID  
      and Versicherungsvertrag.Mitarbeiter_ID = Mitarbeiter.ID;
```

Übung 3 – Berichtigen Sie den folgenden SELECT-Befehl.

Zeigen Sie zu jedem Mitarbeiter die Daten seines Dienstwagens (Kennzeichen, Typ, Hersteller) an.

```
select ID, Name, Vorname,  
       Kennzeichen, Bezeichnung, Name  
from Mitarbeiter mi, Dienstwagen dw,  
     Fahrzeugtyp ft, Fahrzeughersteller fh  
where ID = dw.Mitarbeiter_ID  
      and ID = dw.Fahrzeugtyp_ID  
      and ID = ft.Hersteller_ID  
order by Name, Vorname;
```

17.8. Lösungen

Lösung zu Übung 1 – Was ist an diesem SELECT-Befehl falsch?

1. Die ID muss mit Tabellennamen oder Alias versehen sein, weil sie in beiden Tabellen enthalten ist.
2. Die Spalte *Kennzeichen* gehört zur Tabelle *Fahrzeug*, also ist der Alias *vv* falsch.

Lösung zu Übung 2 – Was ist an diesem SELECT-Befehl falsch?

1. Die ID muss sowohl in der Spaltenliste als auch in der WHERE-Klausel mit Tabellennamen oder Alias versehen sein, weil sie in allen Tabellen enthalten ist.
2. Gleiches gilt für Name und Vorname, weil diese Angaben in mehreren Tabellen enthalten sind.

Wenn (wie in den ANMERKUNGEN ZUR BEISPIELDATENBANK⁶ erwähnt) auch für die Kunden Kontaktdaten gespeichert wären, müsste das auch bei der Spalte *Telefon* beachtet werden. Für die Spalte *Ort* gilt das nicht, weil diese nicht zur Tabelle *Mitarbeiter* gehört, sondern zur Tabelle *Abteilung*, die hier nicht benutzt wird.

Lösung zu Übung 3 – Berichtigten Sie den folgenden SELECT-Befehl.

```
select mi.ID, mi.Name, mi.Vorname,
       dw.Kennzeichen, ft.Bezeichnung, fh.Name
from Mitarbeiter mi, Dienstwagen dw,
     Fahrzeugtyp ft, Fahrzeughersteller fh
where mi.ID = dw.Mitarbeiter_ID
      and ft.ID = dw.Fahrzeugtyp_ID
      and fh.ID = ft.Hersteller_ID
order by mi.Name, mi.Vorname;
```

⁶ [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCFHRUNG%20IN%20SQL%3A%20BEISPIELDATENBANK%23ANMERKUNGEN](http://de.wikibooks.org/wiki/Einf%C3%9Cfchrung%20in%20SQL%3A%20Beispieldatenbank%23Anmerkungen)

18. Einfache Tabellenverknüpfung

Dieses Kapitel behandelt den „traditionellen“ Weg, mehrere Tabellen gleichzeitig abzufragen. Dazu werden in der FROM-Klausel alle Tabellen aufgeführt; die WHERE-Klausel enthält neben den Auswahlbedingungen auch Verknüpfungsbedingungen, wie die Tabellen zueinander gehören.

18.1. Alle Kombinationen aller Datensätze

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). Nützlich ist sie nur dann, wenn auf einfachem Wege große Mengen von Testdaten erzeugt werden sollen, wie es im Kapitel TESTDATEN ERZEUGEN¹ benutzt wird.

18.2. Zwei Tabellen einfach verbinden

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

Gib mir (einige) Spalten der Tabelle *Mitarbeiter* zusammen mit (einigen) Spalten der Tabelle *Dienstwagen*, und zwar bei jedem Mitarbeiter denjenigen Dienstwagen, der zu diesem Mitarbeiter gehört.

Woran erkennt man, zu welchem Mitarbeiter ein Dienstwagen gehört? Nun, in der Tabelle *Dienstwagen* ist eine Spalte *Mitarbeiter_ID* enthalten; dieser Wert ist identisch mit der ID eines Eintrags in der Tabelle *Mitarbeiter*.

Wenn man diese Anfrage und diese Information in „Pseudocode“ übersetzt, dann kommt so etwas heraus:

```
Hole Spalten der Tabelle Mitarbeiter
sowie Spalten der Tabelle Dienstwagen
wobei die Mitarbeiter_ID eines Dienstwagens gleich ist der ID eines
Mitarbeiters
```

Das können wir nun in eine vollständige SQL-Abfrage übersetzen; 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;
```

1 Kapitel 35 auf Seite 455

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. */

Wir bekommen also tatsächlich genau diejenigen Mitarbeiter, die über einen (persönlichen) Dienstwagen verfügen.

Hinweis: Wundern Sie sich nicht über die seltsame Reihenfolge. Die Personalnummer wurde als VARCHAR definiert; also kommt das Ergebnis in alphabetischer und nicht in numerischer Reihenfolge.

18.3. Mehrere Tabellen verbinden

In der gleichen Weise können auch mehr als zwei Tabellen verknüpft werden. Im Kapitel GRUPPIERUNGEN² steht ein Beispiel ähnlich wie dieses: ► **Aufgabe:** Gesucht wird für jeden *Fahrzeughersteller* (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, Fahrzeughersteller 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;
```

Wichtig ist, dass es immer eine eindeutige Zuordnung zwischen jeweils einer Spalte einer Tabelle und einer Spalte einer anderen Tabelle gibt. Bitte beachten Sie:

- Statt einer einzigen Spalte kann auch eine Gruppe von Spalten verknüpft werden (z. B. Name + Vorname). Dies macht aber alles umständlicher, unübersichtlicher und unsicherer. Deshalb sollte vorzugsweise über eindeutige IDs o. ä. verknüpft werden.
- Wenn es zwischen einzelnen Tabellen keine „gemeinsamen“ Spalten gibt, dann kommt wieder das kartesische Produkt heraus; das Ergebnis ist dann eher sinnlos.

18.4. Verknüpfungs- und Abfragebedingungen

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. Wer soll da noch durchblicken? Besser ist das in den nächsten Kapiteln ausführlich behandelte Verfahren mit JOIN.

18.5. Zusammenfassung

Dieses Kapitel erläutert, wie mehrere Tabellen einfach durch die FROM-Klausel und passende WHERE-Bedingungen verknüpft werden können:

- In der Spaltenliste sollte immer der jeweilige Tabellename angegeben werden; es kann auch ein Kürzel als Tabellen-Alias verwendet werden.
- In der FROM-Klausel werden alle Tabellen aufgelistet und in der WHERE-Klausel durch geeignete Bedingungen aufeinander bezogen.
- Durch die Vermischung zwischen Verknüpfungs- und Auswahlbedingungen wird dieses Verfahren schnell unübersichtlich.

18.6. 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.

Übung 1 – Eine einfache Abfrage

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?

Übung 2 – Das kartesische Produkt

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?

Übung 3 – Das kartesische Produkt

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?

Übung 4 – Sinnvolle Verknüpfung von Tabellen

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?

Übung 5 – Sinnvolle Verknüpfung von Tabellen

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?

Übung 6 – Sinnvolle Verknüpfung von Tabellen

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?

18.7. Lösungen

Lösung zu Übung 1 – Eine einfache Abfrage

```
SELECT Vertragsnummer, Abschlussdatum, Art,  
       Versicherungsnehmer_ID, Fahrzeug_ID, Mitarbeiter_ID  
from Versicherungsvertrag
```

Es werden 28 Zeilen angezeigt.

Lösung zu Übung 2 – Das kartesische Produkt

```
SELECT vv.Vertragsnummer, vv.Abschlussdatum, vv.Art,  
       vn.Name, vn.Vorname,  
       Fahrzeug_ID,  
       Mitarbeiter_ID  
from Versicherungsvertrag vv, Versicherungsnehmer vn;
```

Es werden etwa 728 Zeilen angezeigt.

Lösung zu Übung 3 – Das kartesische Produkt

```
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;
```

Es werden etwa 570 752 Zeilen angezeigt.

Lösung zu Übung 4 – Sinnvolle Verknüpfung von Tabellen

```
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;
```

Es werden etwa 28 Zeilen angezeigt.

Lösung zu Übung 5 – Sinnvolle Verknüpfung von Tabellen

```
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;
```

Es werden etwa 28 Zeilen angezeigt.

Lösung zu Übung 6 – Sinnvolle Verknüpfung von Tabellen

```
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-' ) );
```

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

18.8. Siehe auch

Bei Wikipedia stehen weitere Hinweise:

- [KARTESISCHES PRODUKT](#)³
- [PSEUDOCODE](#)⁴

³ [HTTP://DE.WIKIPEDIA.ORG/WIKI/KARTESISCHES%20PRODUKT](http://de.wikipedia.org/wiki/Kartesisches%20Produkt)

⁴ [HTTP://DE.WIKIPEDIA.ORG/WIKI/PSEUDOCODE](http://de.wikipedia.org/wiki/Pseudocode)

19. Arbeiten mit JOIN

Dieses Kapitel enthält die Einführung in den „modernen“ Weg, mehrere Tabellen gleichzeitig abzufragen. Dazu wird jede verknüpfte Tabelle in einer JOIN-Klausel aufgeführt; der ON-Parameter enthält die Verknüpfungsbedingung. Die WHERE-Klausel enthält „nur“ die Auswahlbedingungen.

19.1. Die Syntax von 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 <verknüpfte tabelle> ON <bedingung>
```

Als <join-typ> stehen zur Verfügung:

- **[INNER] JOIN**, auch *Equi-Join* genannt, bezeichnet Verknüpfungen „innerhalb“ zweier Tabellen, d. h. einen Teil des kartesischen Produkts, bei dem ein Wert in beiden Tabellen vorhanden ist. INNER JOIN ist der Inhalt dieses Kapitels.
- **OUTER JOIN** bezeichnet Verknüpfungen, bei denen auch Datensätze geliefert 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.

OUTER JOIN wird im NÄCHSTEN KAPITEL¹ behandelt.

Einige Sonderfälle und Ergänzungen zu JOIN werden im Kapitel MEHR ZU JOIN² behandelt.

1 Kapitel 20 auf Seite 211

2 Kapitel 21 auf Seite 225

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. (Zwei Beispiele für andere Übereinstimmungen lernen Sie in „Mehr zu JOIN“ kennen.)

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.

Alle diese Möglichkeiten werden in den nächsten Abschnitten und Kapiteln genauer erläutert.

19.2. INNER JOIN von zwei Tabellen

► **Aufgabe:** Das Beispiel „alle Mitarbeiter mit den zugehörigen Dienstwagen“ aus dem vorigen Kapitel benötigt nur geringe Änderungen.

```
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;

```

Das Ergebnis dieser Abfrage ist identisch mit der Liste im vorigen Kapitel; wir verzichten deshalb auf die erneute Ausgabe.

Die zweite Tabelle wird in die JOIN-Klausel verschoben, die Verknüpfungsbedingung in den ON-Parameter – fertig.

19.3. WHERE-Klausel bei JOINS

Eine solche Abfrage kann wie üblich durch eine WHERE-Klausel auf bestimmte Datensätze eingeschränkt werden. Wenn sich eine solche Suchbedingung auf die verknüpfte Tabelle *Dienstwagen* bezieht, kann man sie sowohl über die WHERE-Klausel als auch über die JOIN-Klausel einbauen. In den beiden folgenden Beispielen geht es nur um die Dienstwagen von Mercedes. Die Information, welche Typen zu Mercedes gehören, kommt über eine Unterabfrage, die ebenfalls einen JOIN verwendet und die in Klammern gesetzt ist. ► **Aufgabe:** Suche die Dienstwagen vom Typ *Mercedes*.

SQL Quelltext

Falsch

```

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 Fahrzeughersteller fh
                               on ft.Hersteller_ID = fh.ID
                             and fh.Name = 'Mercedes-Benz' );

```

Zulässig, aber nicht so schön, weil Vergleichsbedingung und Auswahlbedingung vermischt werden

SQL Quelltext

```

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
where dw.Fahrzeugtyp_ID in ( SELECT ft.ID

```

```
from Fahrzeugtyp ft
  join Fahrzeughersteller fh
    on ft.Hersteller_ID = fh.ID
where fh.Name = 'Mercedes-Benz');
```

Besseres Vorgehen, weil die Auswahlbedingungen als solche direkt zu erkennen sind

Natürlich sind Einschränkungen auf beide Tabellen möglich: ► **Aufgabe:** Gesucht werden die Mitarbeiter mit 'M' und Mercedes als Dienstwagen.

```
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
where dw.Fahrzeugtyp_ID in ( SELECT ft.ID
  from Fahrzeugtyp ft
    join Fahrzeughersteller fh
      on ft.Hersteller_ID = fh.ID
    where fh.Name = 'Mercedes-Benz')
and mi.Name like 'M%';
```

MITNR	NAME	VORNAME	ID	KENNZEICHEN	TYP
10001	Müller	Kurt	1	DO-WB 421	14
20001	Meyer	Walter	2	DO-WB 422	14

Bei diesem Beispiel wird sofort deutlich, welche Bedingungen die Verknüpfung und welche Bedingungen die Auswahl bezeichnen. Auf diese Übersichtlichkeit sollten Sie immer achten.

Übrigens gibt es keine allgemeine Regel, was als **Haupttabelle** und was als verknüpfte Tabelle zu verwenden ist. In den bisherigen Beispielen können die beiden Tabellen ohne weiteres vertauscht werden:

```
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 Fahrzeughersteller fh
      on ft.Hersteller_ID = fh.ID
    where fh.Name = 'Mercedes-Benz')
and mi.Name like 'M%';
```

Mitarbeiter mit 'M' und Mercedes als Dienstwagen

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; das beschleunigt die Abfrage besonders stark.

19.4. INNER JOIN mehrerer Tabellen

Dazu nehmen wir wiederum das komplexe Beispiel aus dem vorigen Kapitel, das bei den Gruppierungen genauer besprochen wird. In diesem Fall spielt die Reihenfolge der JOIN-Klauseln eher keine Rolle, weil es sich sowieso um direkte Übereinstimmungen handelt und nur solche Datensätze benutzt werden, die es zu den betreffenden Werten tatsächlich gibt. ► **Aufgabe:** Gesucht wird für jeden *Fahrzeughersteller* (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 Fahrzeughersteller fh ON fh.ID = ft.Hersteller_ID
GROUP BY Hersteller_ID, Hersteller_Name, Jahr
ORDER BY Jahr, Hersteller_ID;
```

Übrigens ist es zulässig, den „traditionellen“ Weg mit mehreren Tabellen in der FROM-Klausel und den „modernen“ Weg über JOIN zu mischen. Wenn Sie in einem Ausnahmefall wirklich so vorgehen wollen, sollten Sie erst recht genau auf die Übersichtlichkeit und den Zusammenhang der Bedingungen achten. *Der Autor dieses Hinweises kann sich keine passende Situation vorstellen, aber vielleicht ist es auch einmal sinnvoll.*

19.5. Zusammenfassung

In diesem Kapitel lernten Sie die Verknüpfung von Tabellen über die JOIN-Klausel kennen.

- Mit einem INNER JOIN werden Datensätze abgefragt, bei denen ein Wert in je einer Spalte beider Tabellen vorhanden ist.
- In der ON-Klausel steht diese Verknüpfungsbedingung.
- In der WHERE-Klausel stehen die „normalen“ Auswahlbedingungen.

Genauso können mehrere Tabellen verknüpft werden.

19.6. Übungen

Übung 1 – Definition von JOINS

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 `ON Fahrzeug.ID >= Versicherungsvertrag.Fahrzeug_ID` eine zulässige Verknüpfungsbedingung.
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 EXTRACT(YEAR from
sf.Datum) = 2008;
```

Übung 2 – Definition von JOINS

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

```
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.Versicherungsnehmer_ID
join Versicherungsvertrag vv on vv.Fahrzeug_ID = zu.Fahrzeug_ID
where EXTRACT(YEAR from Datum) = 2008
order by Schadensfall_ID, Fahrzeug_ID;
```

Die folgenden Aufgaben entsprechen teilweise Aufgaben aus dem Kapitel „Einfache Tabellenverknüpfung“. Sie sollen jetzt an den passenden Stellen JOINS verwenden, anstatt die Tabellen einfach aufzulisten.

Übung 3 – Sinnvolle Verknüpfung von Tabellen

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.

Übung 4 – Sinnvolle Verknüpfung von Tabellen

Erweitern Sie die Abfrage von Aufgabe 3, sodass *Name* und *Vorname* des Mitarbeiters sowie das Fahrzeug-Kennzeichen eines jeden Vertrags angezeigt werden.

Übung 5 – Sinnvolle Verknüpfung von Tabellen

Ändern Sie die Abfrage von Aufgabe 4 so, dass 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.

19.7. Lösungen

Lösung zu Übung 1 – Definition von JOINS

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 – Definition von JOINS

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, zu.Fahrzeug_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 bei dieser Tabelle 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: Wegen der Klarheit sollte auch hier der Tabellen-Alias *sf* verwendet werden.
- 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 – Sinnvolle Verknüpfung von Tabellen

```
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 – Sinnvolle Verknüpfung von Tabellen

```
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 – Sinnvolle Verknüpfung von Tabellen

```
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 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-' ) );
```

19.8. Siehe auch

In diesem Kapitel werden Sachverhalte der folgenden Themen angesprochen:

- EINFACHE TABELLENVERKNÜPFUNG³
- UNTERABFRAGEN⁴
- GRUPPIERUNGEN⁵

3 Kapitel 18 auf Seite 193

4 Kapitel 25 auf Seite 283

5 Kapitel 24 auf Seite 271

20. 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.

20.1. Die Syntax von OUTER JOIN

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>
```

Als Spezialfälle des OUTER JOIN gibt es die JOIN-Typen **LEFT JOIN**, **RIGHT JOIN**, **FULL JOIN**.

Anstelle von <haupttabelle> und <zusatztabelle> wird bei OUTER JOIN von <linke tabelle> und <rechte tabelle> gesprochen, weil diese Tabellen unterschiedlich behandelt werden.

20.2. Allgemeine Hinweise zu OUTER JOIN

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 <Haupt-tabelle> genannt) und die unter JOIN genannte Tabelle die <rechte Tabelle> (bisher <Zusatz-tabelle> genannt). Bei Verknüpfungen mit mehreren Tabellen ist ebenfalls die unter JOIN genannte Tabelle die <rechte Tabelle>; die unmittelbar vorhergehende Tabelle ist die <linke Tabelle>.

Auch wenn die folgenden Beispiele so aussehen, als wenn die Datensätze sinnvoll sortiert wären, ist das Zufall; bitte denken Sie daran, dass SQL unsortierte Datenmengen liefert. Eine bestimmte Reihenfolge erhalten Sie erst durch ORDER BY.

20.3. LEFT OUTER 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>;
```

Für unser Beispiel sieht das dann so aus: ► **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;
```

(Auszug)

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			

Und wenn wir jetzt die beiden Tabellen vertauschen?

► **Aufgabe:** Dann erhalten wir alle Dienstwagen und dazu die Mitarbeiter, die zum jeweiligen Dienstwagen passen.

```
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;
```

(Auszug)

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

Bitte überlegen Sie selbst, wie sich WHERE-Klauseln auf das Ergebnis einer Abfrage auswirken.

20.4. 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

Nanu, dieses Ergebnis hatten wir doch gerade schon? Bei genauerem Überlegen wird das auch klar: Beim LEFT JOIN werden alle Datensätze der linken Tabelle mit Informationen der rechten Tabelle geliefert; gegenüber dem obigen Beispiel haben wir die beiden Tabellen vertauscht. Beim RIGHT JOIN werden alle Datensätze der rechten Tabelle mit Informationen der linken Tabelle verknüpft; das entspricht diesem Beispiel.

Ob wir also die beiden Tabellen vertauschen oder LEFT gegen RIGHT, bleibt sich zwangsläufig gleich. Kurz und „knackig“ formuliert kann man sich also merken:

UNKNOWN TEMPLATE "Kasten"

Bitte überlegen Sie, welches Ergebnis die Vertauschung der beiden Tabellen beim RIGHT JOIN liefert und welche Auswirkung WHERE-Klauseln haben.

20.5. 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;
```

(Auszug)

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;
```

(Auszug)

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

20.6. 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 *Fahrzeughersteller* eingebunden werden. Beim INNER JOIN war das kein Problem; probieren wir aus, wie es beim OUTER JOIN aussehen könnte.

20.6.1. Mehrere Tabellen parallel

► **Aufgabe:** 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;
```

(Auszug)

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						
	(E-Klasse)	6	13	DO-WB 111	16	W211	
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;

```

(Auszug)

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						
(C-Klasse)	6					W204	
			13	DO-WB 111	16	W211	
(E-Klasse)	6					Saab 9-3	
	8					S40	
	9					C30	
	9						

Versuchen wir eine 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.

Sie sehen: Es kommt genau auf die gewünschten und die tatsächlich vorhandenen Verknüpfungen an.

20.6.2. Gliederung durch Klammern

Für Verknüpfungen, die durch Klammern gegliedert werden, nehmen wir ein anderes Beispiel, nämlich „Mitarbeiter RIGHT JOIN Dienstwagen“, denn die Fahrzeugtypen sind eine Ergänzung zu den Dienstwagen, nicht zu den Mitarbeitern (auch wenn den Abteilungsleitern ein Mercedes zugestanden wird, aber das ist ein anderes Thema und hat nichts mit SQL zu tun).

```
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 Mitarbeiter mi
     right JOIN ( Dienstwagen dw
                join Fahrzeugtyp ft on ft.ID = dw.Fahrzeugtyp_id )
                ON dw.Mitarbeiter_ID = mi.ID;
```

(Auszug)

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						
			13	DO-WB 111	16	W211	
(E-Klasse)	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						

Auch hier erhalten wir ein vergleichbares Ergebnis. Prüfen wir zunächst die Abfrage in der Klammer:

- LEFT JOIN und INNER JOIN haben als Grundlage „alle Dienstwagen“, es wird also eine Datenmenge „alle Dienstwagen“ (mit Zusatzinformationen über die Fahrzeugtypen) erstellt.
- RIGHT JOIN und FULL JOIN gehen aus von „alle Fahrzeugtypen“, es wird also eine Datenmenge „alle Fahrzeugtypen“ (mit Zusatzinformationen über die Dienstwagen) erstellt.

Da der Ausdruck innerhalb der Klammern zuerst ausgewertet wird, wird diese Datenmenge anschließend mit den Mitarbeitern verknüpft, soweit es der Verknüpfungsbedingung auf der Basis von `dw.Mitarbeiter_ID` entspricht.

Mit diesen Erkenntnissen können wir nun auch den Hersteller mit seinem Namen anzeigen; dazu benutzen wir wegen der bisherigen Erkenntnisse das erste Beispiel:

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

(Auszug)

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYPID	TYP	HERSTELLER
80001	Schindler	Christina	8	DO-WB 428	14	A160	
	Mercedes-Benz						
90001	Janssen	Bernhard	9	DO-WB 429	14	A160	
	Mercedes-Benz						
100001	Grosser	Horst	10	DO-WB 4210	14	A160	
	Mercedes-Benz						
110001	Eggert	Louis	11	DO-WB 4211	14	A160	
	Mercedes-Benz						
120001	Carlsen	Zacharias	12	DO-WB 4212	14	A160	
	Mercedes-Benz						
			13	DO-WB 111	16	W211	
(E-Klasse)	Mercedes-Benz						
50002	Braun	Christian	14	DO-WB 352	2	Golf	
	Volkswagen						
50003	Polovic	Frantisek	15	DO-WB 353	3	Passat	
	Volkswagen						
50004	Kalman	Aydin	16	DO-WB 354	4	Kadett	
	Opel						
80002	Aliman	Zafer	17	DO-WB 382	2	Golf	
	Volkswagen						
80003	Langer	Norbert	18	DO-WB 383	3	Passat	
	Volkswagen						
80004	Kolic	Ivana	19	DO-WB 384	4	Kadett	
	Opel						

20.7. Zusammenfassung

In diesem Kapitel lernten Sie die Verwendung von OUTER JOIN kennen:

- Mit dieser Verknüpfung 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.

20.8. Übungen

Übung 1 – Allgemeines

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.

Übung 2 – Allgemeines

Was ist am folgenden SELECT-Befehl falsch und warum? Die Aufgabe dazu lautet: ► **Aufgabe:** 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 ;
```

Übung 3 – Sinnvollen SELECT-Befehl erstellen

Gesucht werden alle registrierten Versicherungsgesellschaften und (soweit vorhanden) deren Kunden mit Name, Vorname.

Übung 4 – Sinnvollen SELECT-Befehl erstellen

Gesucht werden die Dienstwagen, deren Fahrzeugtypen sowie die Hersteller. Die Liste der Typen soll vollständig sein.

Übung 5 – Sinnvollen SELECT-Befehl erstellen

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.

Übung 6 – Sinnvollen SELECT-Befehl erstellen

Gesucht werden alle registrierten Versicherungsgesellschaften sowie alle Kunden mit Name, Vorname, soweit der Nachname mit 'S' beginnt.

Übung 7 – RIGHT oder LEFT

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?

Übung 8 – SELECT-Befehl mit mehreren Tabellen

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.

Übung 9 – SELECT-Befehl mit mehreren Tabellen

Ergänzen Sie die Lösung zu Übung 8 insofern, dass nur Mitarbeiter der Abteilungen 1 bis 5 angezeigt werden; die Zeilen ohne Mitarbeiter sollen unverändert ausgegeben werden.

20.9. Lösungen

Lösung zu Übung 1 – Allgemeines

Die Aussagen 2, 3, 5 sind richtig, die Aussagen 1, 4, 6 sind falsch.

Lösung zu Übung 2 – Allgemeines

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.

Lösung zu Übung 3 – Sinnvollen SELECT-Befehl erstellen

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

Lösung zu Übung 4 – Sinnvollen SELECT-Befehl erstellen

```
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;
```

Lösung zu Übung 5 – Sinnvollen SELECT-Befehl erstellen

```
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.Abtteilung_ID <= 5 or mi.ID is null;
```

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

Lösung zu Übung 6 – Sinnvollen SELECT-Befehl erstellen

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

Lösung zu Übung 7 – RIGHT oder LEFT

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.

Lösung zu Übung 8 – SELECT-Befehl mit mehreren Tabellen

```
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.Hersteller_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).

Lösung zu Übung 9 – SELECT-Befehl mit mehreren Tabellen

```
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.Hersteller_id
where (mi.Abcteilung_id <= 5) or (mi.id is null);
```

21. Mehr zu JOIN




Die folgenden Ergänzungen zu JOIN sind in besonderen Situationen hilfreich.






21.1. Welcher JOIN passt wann?

Diese Frage stellt sich vor allem Anfängern sehr oft. Neben den (theoretischen) Überlegungen der vorigen Kapitel helfen oft besondere Beispiele; vielleicht ist dieses nützlich.

Eltern und ihre Kinder

Wir haben zwei Tabellen: *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*.

LEFT JOIN		Alle Paare und – falls sie Kinder haben – auch diese	<pre>SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key</pre>
INNER JOIN		Nur Paare, die Kinder haben	<pre>SELECT * FROM Paare INNER JOIN Kinder ON Paare.Key = Kinder.Key</pre>
RIGHT JOIN		Alle Kinder und – falls sie Eltern haben – auch diese	<pre>SELECT * FROM Paare RIGHT JOIN Kinder ON Paare.Key = Kinder.Key</pre>

LEFT JOIN IS NULL	 Abb. 5	Nur Paare, die keine Kinder haben	<pre>SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NULL</pre>
RIGHT JOIN IS NULL	 Abb. 6	Nur Waisenkinder	<pre>SELECT * FROM Paare RIGHT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Paare.Key IS NULL</pre>
FULL JOIN	 Abb. 7	Alle Paare und alle Kinder	<pre>SELECT * FROM Paare FULL JOIN Kinder ON Paare.Key = Kinder.Key</pre>
FULL JOIN IS NULL	 Abb. 8	Alle kinderlosen Paare und alle Waisenkinder	<pre>SELECT * FROM Paare FULL JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NULL OR Paare.Key IS NULL</pre>
Die zweite Variante mit dem INNER JOIN kann man auch so ausdrücken:			
LEFT JOIN IS NOT NULL	 Abb. 9	Alle Paare und – falls sie Kinder haben – auch diese, WOBEI es ein Kind geben <u>muss</u>	<pre>SELECT * FROM Paare LEFT JOIN Kinder ON Paare.Key = Kinder.Key WHERE Kinder.Key IS NOT NULL</pre>

Es zeigt sich, dass oft mehrere Wege möglich sind. Darauf hatten wir bereits im letzten Kapitel darauf hingewiesen: Das Ergebnis für "A LEFT JOIN B" unterscheidet sich nicht von "B RIGHT JOIN A".

21.2. SELF JOIN – Verknüpfung mit sich selbst

Eine Besonderheit sind Verknüpfungen einer Tabelle mit sich selbst. Dies wird immer dann nötig, wenn Informationen einer einzigen Spalte aus verschiedenen Datensätzen gesucht und verbunden werden. Dafür wird ein JOIN benutzt, der auf beiden Seiten dieselbe Tabelle <tabelle> benutzt; diese beiden

„Instanzen“ *müssen* durch einen Alias unterschieden werden.

```
SELECT <spaltenliste>
  FROM <tabelle> t1
     JOIN <tabelle> t2 ON <verknüpfung>
  WHERE <auswahlbedingungen>
```

Hinweis: Es funktioniert nicht, wenn eine der beiden Instanzen *mit Alias* und die andere *ohne Alias* benutzt wird. Dann kommt das DBMS erst recht durcheinander. In einem Forum stand einmal ein solches Problem mit einer sehr vertrackten Ausgabe, bei dem diese Ursache erst nach längerer Diskussion klar wurde.

Dies soll zunächst an zwei Beispielen umgesetzt werden.

21.2.1. Beispiel 1

► **Aufgabe:** Zeige zu jedem Fahrzeug andere Fahrzeuge aus dem gleichen Kreis.

Wir beschränken uns darauf, dass in einer Zeile jeweils zwei von allen möglichen Kombinationen angezeigt werden, auch wenn viele Angaben wiederholt werden.

```
select a.Kennzeichen, b.Kennzeichen
  from Fahrzeug a
     JOIN Fahrzeug b
       on SUBSTRING(a.Kennzeichen from 1 for 3)
          = SUBSTRING(b.Kennzeichen from 1 for 3)
  where a.Kennzeichen < b.Kennzeichen
  order by a.Kennzeichen;
```

```
KENNZEICHEN KENNZEICHEN1
BO-GH 102 BO-KL 678
BOR-NO 234 BOR-PQ 567
BOR-NO 234 BOR-RS 890
BOR-PQ 567 BOR-RS 890
GE-AB 123 GE-AC 246
GE-AB 123 GE-EG 892
GE-AC 246 GE-EG 892
RE-CD 456 RE-LM 901
RE-CD 456 RE-LM 902
RE-CD 456 RE-LM 903
```

Gesucht werden Kombinationen eines Fahrzeugs mit jeweils einem anderen Fahrzeug, wobei die Bedingung „gleicher Kreis“ erfüllt sein soll. Wir brauchen also innerhalb eines SELECT-Befehls zwei Zugriffe auf die Tabelle *Fahrzeug* mit

einer passenden Vergleichsbedingung. (Diese haben wir etwas ungenau formuliert, damit sie nicht unübersichtlich wird.)

Dies ist gleichzeitig ein Beispiel dafür, dass beliebige Bedingungen möglich sind. *Überlegen Sie bitte auch, warum unter WHERE die „kleiner als“-Bedingung benutzt wird.*

21.2.2. Beispiel 2

► **Aufgabe:** Zeige zu jedem Fahrzeug mit mehreren Schadensfällen den zeitlichen Abstand von einem Vorfall zum nächsten an.

Lösungsweg 1

Wir benötigen für jedes der Fahrzeuge aus der Tabelle *Schadensfall* zwei Einträge mit dem Datum sowie den Abstand, der einfach als Differenz benutzt wird und somit die Anzahl der Tage anzeigt. Die Fahrzeuge sind freilich erst über die Tabelle *Zuordnung_SF_FZ* zu finden und müssen zusätzlich verbunden werden. Außerdem sind die Fahrzeuge und die Reihenfolge der Datumsangaben zu kontrollieren.

Datumsvergleich als Teil der Auswahlbedingung

```
SELECT fz.ID, fz.Kennzeichen,
       sf1.Datum AS Datum1, sf2.Datum AS Datum2, sf2.Datum - sf1.Datum AS
Abstand
FROM Zuordnung_SF_FZ zu1
   join Zuordnung_SF_FZ zu2 on zu1.Fahrzeug_ID = zu2.Fahrzeug_ID
   join Schadensfall sf1 ON zu1.Schadensfall_ID = sf1.ID
   JOIN Schadensfall sf2 on zu2.Schadensfall_ID = sf2.ID
   JOIN Fahrzeug fz ON zu1.Fahrzeug_ID = fz.ID
where sf1.Datum < sf2.Datum
   and sf2.Datum = ( SELECT MIN(sf3.Datum)
                     FROM Schadensfall sf3
                     join Zuordnung_SF_FZ zu3 on
zu3.Schadensfall_ID = sf3.id
                     WHERE sf1.Datum < sf3.Datum
                     and zu3.Fahrzeug_ID = zu1.Fahrzeug_ID )
order by fz.ID, Datum1;
```

ID	KENNZEICHEN	DATUM1	DATUM2	ABSTAND
4	GE-AB 123	03.02.2007	05.10.2008	610
6	HER-EF 789	19.12.2007	21.06.2009	550
7	BO-GH 102	11.07.2007	13.03.2009	611
7	BO-GH 102	13.03.2009	01.08.2009	141

Lösungswege 2 und 3

Alternativen bieten die folgenden Lösungen:

Datumsvergleich als Teil der Verknüpfungsbedingungen

```

SELECT fz.ID, fz.Kennzeichen,
       sf1.Datum AS Datum1, sf2.Datum AS Datum2, sf2.Datum - sf1.Datum AS
Abstand
FROM Zuordnung_SF_FZ      zu1
  join Zuordnung_SF_FZ zu2 on zu1.Fahrzeug_ID      = zu2.Fahrzeug_ID
  join Schadensfall      sf1 ON zu1.Schadensfall_ID = sf1.ID
  JOIN Schadensfall      sf2 on zu2.Schadensfall_ID = sf2.Id
                                and sf1.Datum < sf2.Datum
                                and sf2.Datum = ( SELECT MIN(sf3.Datum)
                                                FROM Schadensfaelle
                                                sf3
                                                join
Zuordnung_SF_FZ zu3
                                                on
zu3.Schadensfall_ID = sf3.ID
                                                WHERE sf1.Datum <
sf3.Datum
                                                and zu3.Fahrzeug_ID =
zu1.Fahrzeug_ID )
  JOIN Fahrzeug          fz ON zu1.Fahrzeug_ID    = fz.ID
order by fz.ID, Datum1;
    
```

Datumsvergleich als Funktion bei den SELECT-Spalten

```

SELECT fz.ID, fz.Kennzeichen,
       sf1.Datum AS Datum1, MIN(sf2.Datum) AS Datum2, MIN(sf2.Datum -
sf1.Datum) AS Abstand
FROM Zuordnung_SF_FZ      zu1
  join Zuordnung_SF_FZ zu2 on zu1.Fahrzeug_ID      = zu2.Fahrzeug_ID
  join Schadensfall      sf1 ON zu1.Schadensfall_ID = sf1.ID
  JOIN Schadensfall      sf2 on zu2.Schadensfall_ID = sf2.Id
                                and sf1.Datum < sf2.Datum
  JOIN Fahrzeug          fz ON zu1.Fahrzeug_ID    = fz.ID
group by fz.ID, fz.Kennzeichen, sf1.Datum
order by fz.ID, Datum1;
    
```

Erläuterungen

In dieser Aufgabe stecken mehrere Probleme:

- Die Angaben aus der Spalte *Datum* der Tabelle *Schadensfall* müssen zweimal geholt werden.
- Zu jedem Schadensfall wird der Eintrag der Tabelle *Zuordnung_SF_FZ* benötigt, weil die Schadensfälle für jedes Fahrzeug gesucht werden.

- Das Datum, das zu sf1 gehört, muss immer „kleiner“ sein, also früher liegen als das Datum, das zu sf2 gehört.
- Außerdem benötigen wir „irgendwo“ die Einschränkung, dass zum Vergleich nur der jeweils folgende Schadensfall genommen werden darf, also das Minimum der späteren Einträge:
 - Die erste Lösung verwendet dafür eine Unterabfrage für eine Auswahlbedingung.
 - Die zweite Lösung arbeitet mit einer Unterabfrage bei der Verknüpfungsbedingung.
 - Die dritte Lösung benutzt das MINimum direkt als Aggregatfunktion und verlangt „zum Ausgleich“ eine GROUP BY-Klausel.

Die Lösung benötigt deshalb mehrfach verknüpfte Tabellen:

- Als Grundlage wird die Tabelle der Zuordnungen zwischen Schadensfällen und Fahrzeugen zu1 verwendet.
- Hauptverknüpfung ist der Self-Join zu2 auf dieselbe Tabelle, weil nur solche Einträge verknüpft werden sollen, die sich auf dasselbe Fahrzeug beziehen.
- Zu jedem Schadensfall aus zu1 werden die detaillierten Angaben aus sf1 geholt.
- Zu jedem Schadensfall aus zu2 werden die detaillierten Angaben aus sf2 geholt.
- Ergänzend benötigen wir das Kennzeichen des betreffenden Fahrzeugs, also einen JOIN auf *Fahrzeug*.
- Vor allem müssen die Vergleichsbedingungen für die Datumsangaben eingebaut werden.

Welche Lösung die Datenbank am wenigsten belastet, kann nicht generell gesagt werden, weil es von zu vielen Umständen abhängt.

21.2.3. Erweiterung durch einen OUTER JOIN

Bei diesen Lösungen stehen nicht alle Schadensfälle im Ergebnis, weil es nur um den zeitlichen Abstand ging. Wenn beispielsweise auch die Schadenshöhe gewünscht wird, müssen wir dafür sorgen, dass von sf1 oder sf2 alle Einträge angezeigt werden; wir brauchen also einen OUTER JOIN wie zum Beispiel (auf der Grundlage der letzten Version) so:

```
SELECT fz.ID, fz.Kennzeichen,
       sf1.Datum AS Datum1, MIN(sf2.Datum) AS Datum2, MIN(sf2.Datum -
       sf1.Datum) AS Abstand,
       sf1.Schadenshoehe
```

```

FROM Zuordnung_SF_FZ      zu1
  left join Zuordnung_SF_FZ zu2 on zu1.Fahrzeug_ID      =
zu2.Fahrzeug_ID
  left join Schadensfall    sf1 ON zu1.Schadensfall_ID = sf1.ID
  left JOIN Schadensfall    sf2 on zu2.Schadensfall_ID = sf2.Id
      and sf1.Datum < sf2.Datum
  left JOIN Fahrzeug        fz  ON zu1.Fahrzeug_ID      = fz.ID
group by fz.ID, fz.Kennzeichen, sf1.Datum, sf1.Schadenshoehe
order by fz.ID, Datum1;

```

ID	KENNZEICHEN	DATUM1	DATUM2	ABSTAND	SCHADENSHOEHE
3	RE-LM 903	27.05.2008			
1.438,75					
4	GE-AB 123	03.02.2007	05.10.2008	610	
1.234,50					
4	GE-AB 123	05.10.2008			
1.983,00					
5	RE-CD 456	11.07.2007			
2.066,00					
6	HER-EF 789	19.12.2007	21.06.2009	550	
3.715,60					
6	HER-EF 789	21.06.2009			
865,00					
7	BO-GH 102	11.07.2007	13.03.2009	611	
2.066,00					
7	BO-GH 102	13.03.2009	01.08.2009	141	
4.092,15					
7	BO-GH 102	01.08.2009			
2.471,50					

Wir nehmen es hin, dass dann alle Schadensfälle aufgeführt werden, auch für die Fahrzeuge, die nur einmal „aufgefallen“ sind. Dies ist eine Folge davon, dass Grundlage aller Verknüpfungen die Tabelle der Zuordnungen sein musste.

Bei allen solchen Situationen müssen Sie genau überlegen, wie die verschiedenen Instanzen miteinander verknüpft werden und wie die übrigen Bedingungen einzubinden sind. Oft führen erst mehrere Versuche zum Ziel. Hilfreich sind auch die Ausführungspläne, die ein DBMS anbieten kann.

21.2.4. Weitere Situationen

Zum Schluss sollen noch ein paar andere Beispiele erwähnt werden, bei denen ein Self-Join hilft.

- Wenn bei den Dienstwagen die privat gefahrenen Strecken abgerechnet werden sollen, können der km-Stand beim Fahrtantritt und beim Fahrtende in derselben Spalte, aber in getrennten Datensätzen gespeichert werden.
- Doppelte Adressen innerhalb einer Adressendatei können aufgespürt werden (siehe Übung 3).

- Wenn in der Tabelle *Mitarbeiter* zu einem Mitarbeiter der Leiter der Abteilung gesucht wird, benötigen wir wegen des doppelten Zugriffs auf dieselbe Tabelle ebenfalls einen Self-Join.

21.3. CROSS JOIN – das kartesische Produkt

Mit dieser speziellen Formulierung kann man deutlich machen, dass man wirklich ein kartesisches Produkt herstellen will und nicht etwa nur die JOIN-Bedingung vergessen hat:

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

Als Ergebnis wird tatsächlich jede Kombination eines Mitarbeiters mit einem Dienstwagen ausgegeben, also n mal m Sätze – wie beim allerersten Versuch im Kapitel *EINFACHE TABELLENVERKNÜPFUNG*¹. Man kann die Ergebnismenge auch einschränken durch eine WHERE-Klausel:

```
SELECT mi.Personalnummer AS MitNr,  
       mi.Name, mi.Vorname,  
       dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ  
FROM Mitarbeiter mi  
     CROSS JOIN Dienstwagen dw  
WHERE mi.Name like 'S%' and CHAR_LENGTH(dw.Kennzeichen) = 10;
```

Hinweis: Die DBMS verhalten sich bei einem CROSS JOIN unterschiedlich; teilweise ist ein CROSS JOIN mit WHERE-Klausel nichts anderes als ein INNER JOIN.

Der Nutzen des CROSS JOIN kann deshalb mit unserer sparsamen Beispieldatenbank nicht deutlich gemacht werden. Unter *Oracle* wäre folgendes Verfahren möglich und hilfreich: ► **Aufgabe:** Kontrollieren Sie mit der Tabelle *Fahrzeugauftrag*, welche Fahrzeuge am 2.12.2009 im Fuhrpark zur Verfügung stehen.

1 Kapitel 18 auf Seite 193

Oracle Quelltext

```

SELECT mi.Personalnummer AS MitNr,
       mi.Name, mi.Vorname,
       dw.ID AS DIW, dw.Kennzeichen, dw.Fahrzeugtyp_ID AS Typ
FROM Mitarbeiter mi
     CROSS JOIN Dienstwagen dw
     INNER JOIN Fahrzeugbuchung fb
           ON dw.Kennzeichen = fb.Kennzeichen
WHERE fb.Datum = to_date('02.12.2009','dd.mm.yyyy')
       AND fb.Status = 'noch nicht gebucht';

```

21.4. WITH – Inline-View

Oft kommt es vor, dass man die Daten aus einer Tabelle erst bearbeiten möchte, bevor man sie mit einer anderen Tabelle verknüpft. Beispiel:

```

SELECT Kuerzel, Bezeichnung, Anzahl_Mitarbeiter
FROM Abteilung
     INNER JOIN ( select Abteilung_ID, count(*) AS Anzahl_Mitarbeiter
                FROM Mitarbeiter
                GROUP BY Abteilung_ID
                ) MA_Anzahl
     ON Abteilung.ID = MA_Anzahl.Abteilung_ID
;

```

Dabei wird zunächst nach der Tabelle *Mitarbeiter* die Anzahl der Mitarbeiter für jede Abteilung bestimmt. Das Ergebnis wird wie eine Tabelle *MA_Anzahl* behandelt und über *Abteilung_ID* mit der Tabelle *Abteilung* verknüpft.

Diese Syntax ist ziemlich verschachtelt. Man kann sie auch so schreiben:

```

WITH MA_Anzahl as
( select Abteilung_ID, count(*) AS Anzahl_Mitarbeiter
  FROM Mitarbeiter
  GROUP BY Abteilung_ID
)
SELECT Kuerzel, Bezeichnung, Anzahl_Mitarbeiter
FROM Abteilung
     INNER JOIN MA_Anzahl
           ON Abteilung.ID = MA_Anzahl.Abteilung_ID
;

```

MA_Anzahl wird benutzt wie eine VIEW, die allerdings nicht permanent angelegt wird, sondern die nur für die Ausführung dieses einen SQL-Befehls gültig

ist. Der Unterschied liegt „nur“ darin, dass die Unterabfrage herausgelöst wird und durch WITH als separate temporäre Tabelle eingebunden wird.

Ob man die obere oder die untere Variante besser findet, ist sicher Geschmackssache und hat auch damit zu tun, welche Formulierung man gewöhnt ist. Wenn man sich die SQL-Beispiele im Kapitel FREMDSCHLÜSSEL-BEZIEHUNGEN² daraufhin ansieht, dann würde eine Formulierung ohne WITH viel länger und komplizierter werden.

21.5. Zusammenfassung

In diesem Kapitel lernten Sie einige weitere Möglichkeiten im Zusammenhang mit JOINS kennen.

- Für bestimmte Anforderungen sind Verknüpfungen einer Tabelle mit sich selbst sinnvoll oder notwendig.
- In diesen Fällen sind die Auswahl- und Verknüpfungsbedingungen besonders sorgfältig zu bestimmen.
- Durch WITH können Verknüpfungen über JOINS übersichtlicher werden.

21.6. Übungen

Übung 1 – Fragen zum Verständnis

Welche der folgenden Aussagen sind wahr, welche falsch?

1. Eine Tabelle kann mit sich selbst verknüpft werden.
2. SELF JOIN ist nur ein inhaltlicher Begriff, aber kein SQL-Schlüsselwort.
3. Bei einem SELF JOIN sind nur INNER JOINS erlaubt.
4. Eine bestimmte Tabelle darf in einem SELF JOIN nur zweimal verwendet werden.
5. Für einen SELF JOIN können Tabellen-Aliase benutzt werden, aber sie sind nicht überall erforderlich.
6. Ein CROSS JOIN ist eine Verknüpfung zweier Tabellen ohne Verknüpfungsbedingung.

2 Kapitel 28 auf Seite 343

7. Bei einem CROSS JOIN darf sich die WHERE-Klausel nicht auf die (rechte) Tabelle des JOINS beziehen.
8. Die Schreibweise mit WITH ist kein Sonderfall eines JOINS, sondern eine übersichtlichere Schreibweise, wenn mehrere Tabellen verknüpft werden.

Übung 2 – Verknüpfung einer Tabelle mit sich selbst

Suchen Sie zu jedem Mitarbeiter den Namen und Vornamen des Leiters der Abteilung. *Die Abteilungsleiter in unserer einfachen Firmenhierarchie haben keinen Vorgesetzten; sie sollen in der Liste deshalb nicht aufgeführt werden.*

Übung 3 – Doppelte Adressen suchen

Suchen Sie Einträge in der Tabelle *Versicherungsnehmer*, bei denen *Name, Vorname, PLZ, Strasse* übereinstimmen. Jeweils zwei dieser Adressen sollen mit ihrer ID und den übereinstimmenden Angaben aufgeführt werden.

Hinweis: Benutzen Sie einen JOIN, der sich *nicht* auf übereinstimmende IDs bezieht.

21.7. Lösungen

Lösung zu Übung 1 – Fragen zum Verständnis

Die Aussagen 1, 2, 6, 8 sind wahr, die Aussagen 3, 4, 5, 7 sind falsch.

Lösung zu Übung 2 – Verknüpfung einer Tabelle mit sich selbst

```
select mil.Abtteilung_ID as Abt, mil.Name, mil.Vorname, mi2.Name as
LtrName, mi2.Vorname as LtrVorn
  from Mitarbeiter mil
        join Abteilung  ab  on mil.Abtteilung_ID = ab.ID
        join Mitarbeiter mi2 on mi2.Abtteilung_ID = ab.ID
 where mi2.Ist_Leiter = 'J'
        and mil.Ist_Leiter = 'N'
```

Lösung zu Übung 3 – Doppelte Adressen suchen

```
select a.Name, a.Vorname, a.PLZ, a.Strasse, a.ID, b.ID
  from Versicherungsnehmer a
     JOIN Versicherungsnehmer b
       on a.Name = b.Name and a.Vorname = b.Vorname
          and a.PLZ = b.PLZ and a.Strasse = b.Strasse
 where a.ID < b.ID;
```

21.8. Siehe auch

Bei Wikipedia finden Sie weitere Erläuterungen:

- AUSWERTUNGSPLAN³, auch „Ausführungsplan“ genannt

³ [HTTP://DE.WIKIPEDIA.ORG/WIKI/AUSWERTUNGSPLAN](http://de.wikipedia.org/wiki/Auswertungsplan)

22. Nützliche Erweiterungen

In diesem Kapitel werden verschiedene Erweiterungen des SELECT-Befehls genauer behandelt.

Die Beispiele beziehen sich auch hier auf den Anfangsbestand der Beispieldatenbank; auf die Ausgabe der selektierten Datensätze wird wiederum weitgehend verzichtet. Bitte probieren Sie alle Beispiele aus und nehmen Sie verschiedene Änderungen vor, um die Auswirkungen zu erkennen.

22.1. DISTINCT – keine doppelten Einträge

Wenn Sie den **DISTINCT**-Parameter bei einem SELECT-Befehl angeben, erhalten Sie nur eindeutige Ergebnisse:

Liste der Fahrzeuge, für die Schadensfälle aufgetreten sind

```
SELECT distinct Fahrzeug_ID
from Zuordnung_SF_FZ;
```

```
Fahrzeug_ID
3
4
5
6
7
```

Bitte beachten Sie: Als „eindeutig“ gilt immer die gesamte Zeile, also alle Spalten zusammen. Die folgende Abfrage liefert alle Datensätze; Fahrzeuge mit mehreren Schadensfällen stehen auch mehrfach in der Liste.

Quelltext

Falsch

```
SELECT distinct Fahrzeug_ID, ID
from Zuordnung_SF_FZ;
```

Nur theoretisch DISTINCT, praktisch nicht

Als Gegenstück zu DISTINCT gibt es den **ALL**-Parameter, der ausdrücklich alle Datensätze abfragt. Dies ist der Standardwert, er kann also auch weggelassen werden und wird äußerst selten benutzt:

```
SELECT all Fahrzeug_ID
from Zuordnung_SF_FZ
```

22.2. Beschränkung auf eine Anzahl Zeilen

Häufig will man nicht sofort das gesamte Ergebnis sehen, sondern nur einen Teil der Zeilen.

Vor allem im Netzwerk kostet es seine Zeit, eine größere Menge von Datensätzen zu übertragen. Es ist deshalb oft praktisch, zunächst einen Teil des Ergebnisses zu holen und anzuzeigen. Während der Anwender sich mit diesem Teilergebnis beschäftigt, wird „im Hintergrund“ der nächste Abschnitt geholt usw.

Der SQL-Standard hat dafür (noch) kein Verfahren festgelegt. Jedes DBMS bietet eine eigene Lösung an; abweichend vom üblichen Vorgehen in diesem Buch erhalten Sie mehrere Lösungen.

Anstelle konstanter Werte (ohne Klammern) kann in allen folgenden Fällen auch ein SQL-Ausdruck (in Klammern) angegeben werden.

22.2.1. Firebird: FIRST SKIP oder ROWS

Möglichkeiten bei Firebird

Firebird bietet gleich zwei Lösungen an, die erste mit **FIRST / SKIP**:

```
SELECT [DISTINCT]
[ FIRST <value1> ]
[ SKIP <value2> ]
<select list> FROM ... /* usw. */
```

Mit dem FIRST-Parameter wird angegeben, wie viele Zeilen am Anfang angezeigt werden sollen; der SKIP-Parameter legt fest, wie viele Zeilen davor übersprungen werden sollen. Beide Parameter können einzeln oder zusammen

benutzt werden; sie folgen direkt als erste Klausel nach DISTINCT, noch vor der Spaltenliste. Einige Beispiele:

- Nur FIRST zeigt die ersten Zeilen.

Firebird Quelltext

```
select FIRST 10
      ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name;
```

- Nur SKIP überspringt die ersten Zeilen.

Firebird Quelltext

```
select SKIP 10
      ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name;
```

- FIRST zeigt die ersten 10 Zeilen an, aber wegen SKIP werden vorher 5 Zeilen übersprungen.

Firebird Quelltext

```
select FIRST 10 SKIP 5
      ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name;
```

Mit einem Ausdruck kann dafür gesorgt werden, dass etwa das erste Viertel der Datensätze abgerufen wird:

Firebird Quelltext

```
select FIRST ( (SELECT count(*) from Mitarbeiter) / 4 )
      ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name;
```

Der Wert für FIRST wird aus der Anzahl der Datensätze berechnet.

Die zweite Firebird-Variante benutzt mit **ROWS** direkt Zeilennummern:

```
SELECT ... FROM ... WHERE ...  
ORDER BY ...  
ROWS <value1> [ TO <value2> ]
```

Mit den ROWS-Parametern wird festgelegt, dass (nur) eine bestimmte Anzahl von Zeilen angezeigt werden sollen, die durch die Zeilennummern gekennzeichnet sind.

- Wenn nur ROWS benutzt wird, bezeichnet <value1> die Gesamtzahl der angezeigten Zeilen.
- Wenn ROWS zusammen mit TO benutzt wird, ist <value1> die erste Zeilennummer und <value2> die letzte Zeilennummer.

Einige Beispiele:

- Ausgabe der Zeilen 10 bis 20 (also insgesamt 11 Zeilen)

Firebird Quelltext

```
select ID, Name, Vorname, Abteilung_ID as Abt  
from Mitarbeiter  
order BY Name  
ROWS 10 TO 20;
```

- Der erste Datensatz gemäß Sortierung

Firebird Quelltext

```
select ID, Name, Vorname, Abteilung_ID as Abt  
from Mitarbeiter  
order BY Name  
ROWS 1;
```

- Der letzte Datensatz gemäß Sortierung

```
Firebird Quelltext
```

```
select ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name desc
  ROWS 1;
```

Vor allem das letzte Beispiel, bei dem mit DESC die Sortierung umgekehrt wird, ist oft sehr nützlich.

Bei einer Interbase-Datenbank sind auch prozentuale Angaben möglich; das ist bei Firebird entfallen.

22.2.2. Microsoft SQL: TOP

Möglichkeiten bei Microsoft-SQL

Der MS-SQL Server benutzt diese Syntax:

```
SELECT [DISTINCT]
  TOP ( <value> ) [PERCENT] [WITH TIES]
 <select list> FROM ... /* usw. */
```

Mit den **TOP**-Parametern wird festgelegt, dass (nur) eine bestimmte Anzahl von Zeilen angezeigt werden sollen, die durch die Zeilennummern gekennzeichnet sind. Diese Parameter folgen direkt als erste Klausel nach DISTINCT, noch vor der Spaltenliste.

- <value> bezeichnet die Gesamtzahl der angezeigten Zeilen. Es wird empfohlen, die Klammern immer zu setzen.
- Wenn TOP zusammen mit PERCENT benutzt wird, handelt es sich dabei um die prozentuale Angabe der Zeilenzahl.
- WITH TIES muss zusammen mit ORDER BY benutzt werden und liefert dann zusätzliche doppelte Zeilen, wenn der letzte Wert nach der Reihenfolge gleich ist den Werten in danach folgenden Zeilen.

Einige Beispiele:

- Nur TOP zeigt die ersten Zeilen.

MS-SQL Quelltext

```
select
  TOP 10
  ID, Name, Vorname, Abteilung_ID as Abt
from Mitarbeiter
order BY Name;
```

- TOP + PERCENT zeigt z. B. das erste Viertel an.

MS-SQL Quelltext

```
select
  TOP 25 PERCENT
  ID, Name, Vorname, Abteilung_ID as Abt
from Mitarbeiter
order BY Name;
```

- Der letzte Datensatz gemäß Sortierung

MS-SQL Quelltext

```
select
  TOP 1
  ID, Name, Vorname, Abteilung_ID as Abt
from Mitarbeiter
order BY Name desc;
```

Vor allem das letzte Beispiel, bei dem mit DESC die Sortierung umgekehrt wird, ist oft sehr nützlich.

22.2.3. MySQL und PostgreSQL: LIMIT

Möglichkeiten bei MySQL und PostgreSQL

Diese DBMS benutzen den **LIMIT**-Parameter. Dieser Parameter folgt nach **ORDER BY**, wobei die Sortierung nicht angegeben werden muss.

```
SELECT ... FROM ... WHERE ...
ORDER BY ...
LIMIT <value1> OFFSET <value2>
```

Dabei wird mit <value1> angegeben, wie viele Zeilen am Anfang angezeigt werden sollen. Mit <value2> kann nach dem Begriff **OFFSET** außerdem angegeben werden, wie viele Zeilen davor übersprungen werden sollen.

- Es werden die ersten 10 Zeilen angezeigt.

MySQL Quelltext

```
select ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name
LIMIT 10;
```

- Es werden die ersten 10 Zeilen angezeigt, aber vorher werden 5 Zeilen übersprungen.

MySQL Quelltext

```
select ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name
LIMIT 10 OFFSET 5;
```

- Der letzte Datensatz gemäß Sortierung

MySQL Quelltext

```
select ID, Name, Vorname, Abteilung_ID as Abt
  from Mitarbeiter
 order BY Name desc
LIMIT 1;
```

Vor allem das letzte Beispiel, bei dem mit DESC die Sortierung umgekehrt wird, ist oft sehr nützlich.

Es gibt noch eine andere Schreibweise für diesen Parameter ohne das Wort **OFFSET**:

```
SELECT ... FROM ... WHERE
ORDER BY ...
LIMIT [ <value2>, ] <value1>
```

Bei dieser Variante wird genauso mit <value1> angegeben, wie viele Zeilen am Anfang angezeigt werden sollen. Mit <value2> kann außerdem angegeben werden, wie viele Zeilen davor übersprungen werden sollen; dieser Wert wird jedoch zuerst angegeben und durch ein Komma von der gewünschten Zeilenzahl getrennt.

Die Bedeutung dieser Variante ist mit der ersten Variante identisch, sodass Beispiele nicht nötig sind. Es ist wohl Geschmackssache, welche Version „eingängiger“ ist.

22.2.4. Oracle: ROWNUM

Möglichkeiten bei Oracle

Bei dem DBMS Oracle gibt es bei jedem SELECT-Ergebnis eine implizite Spalte *Rownum*. Man kann diese Spalte mit ausgeben lassen. Solange man kein ORDER BY angibt, ist die Reihenfolge der ausgegebenen Sätze nicht festgelegt. Dieselbe Abfrage kann an einem anderen Tag durchaus eine andere Nummerierung der Sätze hervorbringen. Das liegt z.B. daran, dass jemand die Datensätze in der Zwischenzeit reorganisiert hat.

Rownum als implizite Spalte der Ergebnismenge

Oracle Quelltext

```
SELECT Name, rownum
FROM Mitarbeiter;
```

```
NAME  ROWNUM
Müller  1
Schneider  2
Meyer  3
Schmitz  4 /* usw. */
```

Wenn man diese Spalte *Rownum* nicht angibt, dann wird sie auch nicht ausgegeben. Ihr Vorteil ist, dass man sie auch bei WHERE verwenden kann:

Oracle Quelltext

```
SELECT Name
FROM Mitarbeiter
WHERE Rownum <= 2;
```



```
NAME  ROWNUM
Müller    1
Schneider 2
```

Es werden nur die ersten 2 Zeilen angezeigt.

Folgende Formulierung funktioniert allerdings nicht, wenn man nur den 10. Satz ausgeben will:

Oracle Quelltext

Falsch

```
SELECT Name
  FROM Mitarbeiter;
WHERE Rownum = 10;
```

Nur den 10. Satz anzeigen – so geht es nicht.

Das liegt daran, dass der Zähler *Rownum* nur die Zeilen zählt, die auch wirklich ausgegeben werden. Wenn die Tabelle 500 Sätze hat, dann wird bei jedem Satz geprüft, ob *Rownum* bereits den Wert 10 erreicht hat. Das ist jedoch nie der Fall, da der Zähler immer den Wert 0 behält.

Da muss man schon das DBMS zwingen, die Ergebnismenge mit der *Rownum* zwischenzuspeichern. Dann geht es:

Oracle Quelltext

```
SELECT Name
  FROM ( SELECT Name, rownum R
        FROM Mitarbeiter )
WHERE R = 10;
```

Nur den 10. Satz anzeigen – so klappt es.

Welcher Satz dabei ausgegeben wird, ist jedoch dem Zufall überlassen.

Wenn man die Ergebnismenge sortiert ausgeben will und dabei nur die ersten 6 Sätze ausgeben haben will, dann funktioniert die folgende Formulierung nicht:

Oracle Quelltext

Falsch

```
SELECT Name
  FROM Mitarbeiter;
WHERE Rownum <= 6
ORDER BY Name;
```

Die alphabetisch ersten 6 Namen ausgeben – so geht es nicht.

Man wird feststellen, dass zwar nur 6 Sätze ausgegeben werden, dass das aber keinesfalls immer die alphabetisch ersten 6 Namen sind. Das liegt daran, dass zunächst WHERE ausgewertet wird und danach erst sortiert wird. Es werden also – wie bei den vorangegangenen Beispielen beschrieben – beliebige 6 Sätze gelesen, und nur diese 6 Sätze werden sortiert.

Für die richtige Lösung muss die Datenbank schon etwas mehr tun. Sie muss nämlich alle vorhandenen Sätze sortieren und danach die ersten 6 Sätze ausgeben:

Oracle Quelltext

```
SELECT Name
  FROM ( SELECT Name
         FROM Mitarbeiter
         ORDER BY Name )
WHERE Rownum <= 6;
```

Die alphabetisch ersten 6 Namen ausgeben – so klappt es.

22.2.5. Sybase: ROWCOUNT

Möglichkeiten bei Sybase

Bei diesem DBMS wird zuerst die Anzahl der gewünschten Zeilen angegeben, danach folgt der SELECT-Befehl.

Sybase Quelltext

```
SET rowcount 10;
SELECT Name, rownum
  FROM Mitarbeiter;
```

22.3. Mehrere Abfragen zusammenfassen

22.3.1. UNION – Vereinigung

Mit der **UNION**-Klausel werden mehrere Abfragen verknüpft und als einheitliche Ergebnismenge geliefert. Dieses Verfahren ist Standard und steht in jedem DBMS zur Verfügung. Es ist vor allem in zwei Situationen sinnvoll:

- Die Daten stammen aus verschiedenen Tabellen mit ähnlicher Struktur und sollen gemeinsam angezeigt werden.
- Die Daten stammen aus derselben Tabelle; die Auswahlbedingungen sind so komplex, dass eine einzelne Abfrage nicht möglich, nicht sinnvoll oder zu unübersichtlich wäre.

Die Syntax einer solchen Verknüpfung sieht aus wie folgt, wobei auch mehr als zwei Abfragen verknüpft werden können:

```
SELECT <Spaltenliste1>
  FROM <Tabellenliste1>
 WHERE <Bedingungen1>
 UNION [ DISTINCT | ALL ]
 SELECT <Spaltenliste2>
  FROM <Tabellenliste2>
 WHERE <Bedingungen2>
```

Bei den einzelnen Abfragen können grundsätzlich alle Klauseln benutzt werden. Bitte beachten Sie folgende Bedingungen:

- Alle Einzelabfragen müssen in der Anzahl und Reihenfolge der Ergebnis-Spalten übereinstimmen. Die Datentypen müssen je nach DBMS und Version genau gleich sein oder zumindest so ähnlich, dass sie automatisch konvertiert werden können.
- Die Spaltennamen werden aus der ersten Spaltenliste übernommen, ggf. unter Berücksichtigung von Alias-Namen.
- Grundsätzlich werden doppelte Zeilen aus den Einzelabfragen unterdrückt, d. h. DISTINCT ist Standard und kann weggelassen werden. Wenn doppelte Zeilen angezeigt werden sollen, ist ALL anzugeben.
- Die Benutzung von Klammern sowie ORDER BY für das Gesamtergebnis wird je nach DBMS unterschiedlich geregelt.

► **Aufgabe:** Das folgende Beispiel ist eine einfache Zusammenfassung aller Fahrzeuge aus den Tabellen *Fahrzeuge* und *Dienstwagen*:

```
select ID, Kennzeichen, Farbe
  from Dienstwagen
 union
select ID, Kennzeichen, Farbe
  from Fahrzeug;
```

Im folgenden Beispiel werden als Spalte *Var* Konstanten eingetragen, die die Herkunft der Daten angeben, und es werden verschiedene Auswahlbedingungen benutzt.

```
select 'D' as Var, ID, Kennzeichen, Farbe
  from Dienstwagen
 where Fahrzeugtyp_ID <= 3
 union
select 'F', ID, Kennzeichen, Farbe
  from Fahrzeug
 where Kennzeichen LIKE 'B%';
```

Wenn die Ergebnismenge sortiert werden soll, werden je nach DBMS unterschiedliche Schreibweisen benötigt.

MySQL Quelltext

```
( select 'D' as Var, ID, Kennzeichen, Farbe
  from Dienstwagen
  where Fahrzeugtyp_ID <= 3 )
union
( select 'F', ID, Kennzeichen, Farbe
  from Fahrzeug
  where Fahrzeugtyp_ID <= 3 )
order by Kennzeichen;
```

MySQL benutzt Klammern um die Einzelabfragen herum, ORDER BY wird zusätzlich angegeben.

Firebird Quelltext

```
select * from
( select 'D' as Var, ID, Kennzeichen, Farbe
  from Dienstwagen
  where Fahrzeugtyp_ID <= 3
  union
  select 'F', ID, Kennzeichen, Farbe
```

```
from Fahrzeug
where Fahrzeugtyp_ID <= 3 )
order by Kennzeichen;
```

Firebird benutzt eine Hauptabfrage mit ORDER BY und die UNION-Verknüpfung als Unterabfrage.

Mit der folgenden Abfrage werden verschiedene Listen von Mitarbeitern zusammengefasst: zum einen diejenigen, die im September Geburtstag haben, zum anderen die Leiter mit einer Mobil-Rufnummer:

```
select Personalnummer, Name, Vorname, Geburtsdatum, Ist_Leiter as Leiter,
Mobil
  from Mitarbeiter
 where EXTRACT(month from Geburtsdatum) = 9
      union
select Personalnummer, Name, Vorname, Geburtsdatum, Ist_Leiter, Mobil
  from Mitarbeiter
 where Ist_Leiter = 'J' and Mobil <> '';
```

Offensichtlich kann diese Abfrage auch ohne UNION erreicht werden, nämlich durch: WHERE <bedingung1> OR (<bedingung2> AND <bedingung3>). Sie können sich aber sicherlich noch viel kompliziertere Situationen vorstellen, vor allem wenn die WHERE-Bedingungen auf Informationen in unterschiedlichen weiteren Tabellen zugreifen müssen.

22.3.2. Andere Varianten

Unter den gleichen Voraussetzungen wie UNION gibt es weitere Wege, um Einzelabfragen zusammenzufassen. Diese stehen aber nur vereinzelt zur Verfügung; wir beschränken uns deshalb auf Hinweise.

Mit **INTERSECT** (MS-SQL) wird der Durchschnitt der Teilabfragen zurückgegeben.

```
SELECT <Spaltenliste1> FROM <Tabellenliste1> WHERE
<Bedingungen1>
INTERSECT
SELECT <Spaltenliste2> FROM <Tabellenliste2> WHERE
<Bedingungen2>
```

Die Ergebnismenge besteht aus allen Zeilen, die sowohl zum ersten SELECT-Befehl als auch zum zweiten SELECT-Befehl gehören.

Mit **EXCEPT** (MS-SQL) oder **MINUS** (Oracle) wird die Differenz der Teilabfragen zurückgegeben.

```
SELECT <Spaltenliste1> FROM <Tabellenliste1> WHERE
<Bedingungen1>
    EXCEPT | MINUS
SELECT <Spaltenliste2> FROM <Tabellenliste2> WHERE
<Bedingungen2>
```

Die Ergebnismenge besteht aus allen Zeilen, die zum ersten SELECT-Befehl gehören, aber beim zweiten SELECT-Befehl nicht enthalten sind.

Wie bei UNION können auch solche Verknüpfungen durch sehr komplizierte JOINS und WHERE-Klauseln nachgebildet werden.

22.4. CASE WHEN – Fallunterscheidungen

Immer wieder möchte man innerhalb eines SQL-Befehls je nach Situation unterschiedliche Werte erhalten. Das einfachste Beispiel ist die Tabelle *Mitarbeiter*: Anstelle des Feldinhalts *Ist_Leiter* mit 'J' oder 'N' kann 'Leiter' oder eine leere Zeichenkette angezeigt werden. Dafür ist der **CASE**-Ausdruck vorgesehen, den es in zwei Varianten gibt.

Als <expression> können wahlweise konstante Werte oder komplexe Ausdrücke verwendet werden. Der CASE-Ausdruck ist nicht nur für SELECT, sondern auch für Speichern-Befehle geeignet und nicht nur (wie in den meisten Beispielen) als Teil der Spaltenliste, sondern auch in der WHERE-Klausel oder an anderen Stellen, an denen entsprechende Werte benötigt werden.

Der Wert, den der CASE-Ausdruck zurückgibt, erhält einen passenden Datentyp, der sich aus den Result-Werten ergibt.

22.4.1. Simple Case – die einfache Fallunterscheidung

Die einfache Variante hat folgende Syntax:

```
CASE <expression>
  WHEN <expression1> THEN <result1>
  [ WHEN <expression2> THEN <result2> ] /* usw. */
  [ ELSE <default> ]
END
```

Bei dieser Version wird der Wert, der sich durch den ersten Ausdruck beim CASE ergibt, nacheinander mit den Ausdrücken nach WHEN verglichen. Sobald eine Gleichheit festgestellt wird, wird der Wert, der sich durch den Result-Ausdruck ergibt, als Wert für den CASE-Ausdruck übernommen. Wenn es keine Übereinstimmung gibt, wird der Wert des Default-Ausdrucks übernommen; wenn ELSE nicht vorhanden ist, wird NULL als Wert genommen. Beispiele:

► **Aufgabe:** Ein vorhandener Feldinhalt, nämlich 'J' oder 'N', wird für die Ausgabe durch andere Texte ersetzt:

```
select Personalnummer as Pers, Name, Vorname, Geburtsdatum as Geb,
       CASE Ist_Leiter
         WHEN 'J' THEN 'Leiter'
         ELSE ""
       END as Leiter,
       Mobil
from Mitarbeiter;
```

PERS	NAME	VORNAME	GEB	LEITER	MOBIL
40001	Langmann	Matthias	28.03.1976	Leiter	
40002	Peters	Michael	15.11.1973		
50001	Pohl	Helmut	27.10.1980	Leiter	(0171)
4123456					
50002	Braun	Christian	05.09.1966		(0170)
8351647					

Dieselbe Lösung ohne ELSE-Zweig ändert nur die Anzeige:

```
select Personalnummer as Pers, Name, Vorname, Geburtsdatum as Geb,
       CASE Ist_Leiter
         WHEN 'J' THEN 'Leiter'
       END as Leiter,
       Mobil
from Mitarbeiter;
```

PERS	NAME	VORNAME	GEB	LEITER	MOBIL
40001	Langmann	Matthias	28.03.1976	Leiter	
40002	Peters	Michael	15.11.1973	<null>	
50001	Pohl	Helmut	27.10.1980	Leiter	(0171)
4123456					
50002	Braun	Christian	05.09.1966	<null>	(0170)
8351647					

► **Aufgabe:** Bei der Art der Versicherungsverträge möchten wir die Varianten im „Klartext“ lesen:

```
select Vertragsnummer as Vertrag, Abschlussdatum as Datum,
       CASE Art
         WHEN 'VK' THEN 'Vollkasko'
         WHEN 'TK' THEN 'Teilkasko'
         WHEN 'HP' THEN 'Haftpflicht'
       END
from Versicherungsvertrag;
```

VERTRAG	DATUM	CASE
DB-04	25.01.2008	Haftpflicht
RH-01	11.12.1976	Vollkasko
RD-02	29.01.1988	Haftpflicht
RM-03	13.01.1996	Haftpflicht
RD-04	23.11.2006	Haftpflicht
RR-05	29.06.1990	Teilkasko

22.4.2. Searched Case – die komplexe Fallunterscheidung

Die komplexe Variante hat keinen Ausdruck hinter dem CASE; sie arbeitet nach folgender Syntax:

```
CASE
  WHEN <bedingung1> THEN <result1>
  [ WHEN <bedingung2> THEN <result2> ] /* usw. */
  [ ELSE <default> ]
END
```

Bei dieser Variante werden nacheinander Bedingungen geprüft. Sobald eine Bedingung als WAHR festgestellt wird, wird der Wert, der sich durch den Resultat-Ausdruck ergibt, als Wert für den CASE-Ausdruck übernommen. Wenn es keine Übereinstimmung gibt, wird der Wert des Default-Ausdrucks übernommen; wenn ELSE nicht vorhanden ist, wird NULL als Wert genommen. Beispiele:

► **Aufgabe:** In einer einzigen Zeile wird angegeben, wieviele Versicherungsverträge innerhalb eines Jahrzehnts abgeschlossen wurden.

```
select
  SUM( CASE WHEN EXTRACT(YEAR from Abschlussdatum) between 1970 and
1979
        THEN 1
        ELSE 0
      END ) as S_197_,
  SUM( CASE WHEN EXTRACT(YEAR from Abschlussdatum) between 1980 and
1989
        THEN 1
        ELSE 0
```



```

        END ) as S_198_,
        SUM( CASE WHEN EXTRACT(YEAR from Abschlussdatum) between 1990 and
1999
                THEN 1
                ELSE 0
        END ) as S_199_,
        SUM( CASE WHEN EXTRACT(YEAR from Abschlussdatum) >= 2000
                THEN 1
                ELSE 0
        END ) as S_200_
from Versicherungsvertrag;

```

S_197_	S_198_	S_199_	S_200_
6	6	6	5

Dazu wird für jedes Jahrzehnt eine Spalte vorgesehen. Jede Spalte enthält einen CASE-Ausdruck mit einer WHEN-Bedingung. Wenn für eine Zeile diese Bedingung TRUE ergibt, wird 1 zur Summe dieses Jahrzehnts addiert, andernfalls 0.

► **Aufgabe:** In den folgenden Versuchen wird der CASE-Ausdruck innerhalb von ORDER BY für eine unterschiedliche Sortierung, abhängig von der Art der Versicherung, verwendet:

```

select ID, Art,
        Abschlussdatum as Datum, Vertragsnummer as Vertr,
        Mitarbeiter_ID as Mit, Fahrzeug_ID as FZG
from Versicherungsvertrag
order by Art,
        CASE WHEN Art = 'TK' THEN ID
              WHEN Art = 'VK' THEN Mitarbeiter_ID
              WHEN Art = 'HP' THEN Fahrzeug_ID
        END;

```

ID	ART	DATUM	VERTR	MIT	FZG
14	HP	15.03.1998	KG-03	9	16
18	HP	17.05.2000	HG-03	9	17
19	HP	21.09.2004	HB-04	9	19
10	HP	23.11.2006	RD-04	9	20
6	HP	25.01.2008	DB-04	9	21
15	HP	27.03.1988	KV-04	10	22
11	TK	29.06.1990	RR-05	9	23
12	TK	14.02.1978	KB-01	10	6
21	VK	20.06.1982	XH-02	9	8
23	VK	19.07.2002	XO-04	9	18
7	VK	11.12.1976	RH-01	10	5
22	VK	05.06.1992	XW-03	10	13

In diesem Beispiel wurden alle Abschnitte numerisch sortiert. Auch die Sortierung eines Teils der Datensätze nach einem Datum funktioniert, muss aber unter Umständen überarbeitet werden:

SQL Quelltext

Falsch

```
select ID, Art,
       Abschlussdatum as Datum, Vertragsnummer as Vertr,
       Mitarbeiter_ID as Mit, Fahrzeug_ID as FZG
from Versicherungsvertrag
order by Art,
       CASE WHEN Art = 'TK' THEN ID
            WHEN Art = 'VK' THEN Mitarbeiter_ID
            WHEN Art = 'HP' THEN Abschlussdatum
       END;
```

```
Invalid token.
SQL error code = -104.
Datatypes are not comparable in expression CASE.
```

Unter Firebird so nicht möglich

Auch wenn Firebird (wie im ersten Beispiel) Namen von Spalten akzeptiert, werden die Datentypen der Spalten verglichen. Diese sind offensichtlich nicht kompatibel; also bricht Firebird diese Abfrage ab. Aber in der ORDER BY-Klausel können auch die Nummern der Spalten angegeben werden; dann klappt es:

Firebird Quelltext

```
select ID, Art,
       Abschlussdatum as Datum, Vertragsnummer as Vertr,
       Mitarbeiter_ID as Mit, Fahrzeug_ID as FZG
from Versicherungsvertrag
order by Art,
       CASE WHEN Art = 'TK' THEN ID
            WHEN Art = 'VK' THEN Mitarbeiter_ID
            WHEN Art = 'HP' THEN 2
       END;
```

Mit Nummern der Spalten anstelle der Namen.

Bitte wundern Sie sich nicht über das identische Ergebnis wie oben bei der Sortierung nach Fahrzeug_ID: Die Fahrzeuge wurden in der gleichen zeitlichen Reihenfolge erfasst wie die Verträge; also stimmen in der ersten Zeit beide Sortierungen überein.

Die Sortierung kann nur einheitlich festgelegt werden, aber nicht mal so, mal so:

Quelltext

Falsch

```
select ID, Art,
       Abschlussdatum as Datum, Vertragsnummer as Vertr,
       Mitarbeiter_ID as Mit, Fahrzeug_ID as FZG
from Versicherungsvertrag
order by Art,
       CASE
       WHEN Art = 'TK' THEN ID           ASC
       WHEN Art = 'VK' THEN Mitarbeiter_ID ASC
       WHEN Art = 'HP' THEN 2           DESC
END;
```

```
Invalid token.
Dynamic SQL Error.
SQL error code = -104.
Token unknown - line 7, column 39. ASC.
```

Versuch, nur nach dem Abschlussdatum absteigend zu sortieren.

```
select ID, Art,
       Abschlussdatum as Datum, Vertragsnummer as Vertr,
       Mitarbeiter_ID as Mit, Fahrzeug_ID as FZG
from Versicherungsvertrag
order by Art,
       CASE
       WHEN Art = 'TK' THEN ID
       WHEN Art = 'VK' THEN Mitarbeiter_ID
       WHEN Art = 'HP' THEN 2
END DESC;
```

Einheitliche Sortierung DESC sollte möglich sein

Zumindest unter Firebird gelingt dennoch keine korrekte Sortierung: nur der Abschnitt 'TK' wird absteigend sortiert, die beiden anderen nicht. Sie müssen aber sowieso selbst ausprobieren, was in welchem DBMS möglich ist und was nicht.

22.4.3. CASE-Ausdruck beim Speichern

Wie beim SELECT-Befehl kann ein CASE-Ausdruck auch beim Speichern an verschiedenen Stellen benutzt werden, vor allem bei den Zuweisungen der Werte für INSERT und UPDATE sowie bei der WHERE-Klausel für UPDATE und DELETE. Da sich diese Kapitel vor allem mit Abfragen beschäftigen, beschränken wir uns auf ein Beispiel.

► **Aufgabe:** Die Zuständigkeit der Mitarbeiter für die Versicherungsverträge wird neu geregelt:

- Vollkasko-Verträge gehören zum Aufgabenbereich des Abteilungsleiters (Mitarbeiter-ID 9).
- Teilkasko-Verträge gehören zum Aufgabenbereich des Mitarbeiters mit ID 12.
- Die Haftpflicht-Verträge werden in Abhängigkeit von der ID aufgeteilt auf die Mitarbeiter 10 und 11.

```
update Versicherungsvertrag
set Mitarbeiter_id =
  CASE WHEN Art = 'VK' THEN 9
        WHEN Art = 'TK' THEN 12
        WHEN Art = 'HP' THEN 10 + MOD(ID, 2)
END;
```

Es wird also eine normale SET-Anweisung geschrieben. Die CASE-Anweisung liefert die benötigten Werte, wobei für den Fall „Haftpflicht“ für gerade IDs der Wert (10+0) und für ungerade IDs der Wert (10+1) gesetzt wird. Eine WHERE-Klausel ist nicht erforderlich, weil alle Verträge neu zugeordnet werden sollen.

22.5. Zusammenfassung

In diesem Kapitel lernten Sie einige nützliche Erweiterungen vor allem für Abfragen kennen:

- Auf den DISTINCT-Parameter zur Auswahl wird genauer eingegangen.
- Je nach DBMS wird unterschiedlich geregelt, wenn nur eine gewisse Anzahl von Zeilen gewünscht wird.
- Mit UNION wird das Ergebnis von zwei oder mehr Abfragen in einer gemeinsamen Ergebnistabelle zusammengefasst.
- Mit CASE WHEN sind Fallunterscheidungen für gewisse Werte möglich; dies ist auch beim Speichern von Datensätzen sowohl für die Werte als auch für die WHERE-Klausel hilfreich.

22.6. Übungen

Übung 1 – Richtig oder falsch?

Welche der folgenden Aussagen sind richtig, welche sind falsch?

1. Es ist Standard, den Parameter ALL bei einem SELECT-Befehl anzugeben.
2. Bei SELECT DISTINCT werden nur solche Zeilen angezeigt, die sich in mindestens einer Spalte unterscheiden.
3. Es ist äußerst selten erforderlich, das Ergebnis einer Abfrage nur in Teilen zu holen.
4. Für die Anzeige oder Verwendung einer ROW_NUMBER hat der SQL-Standard (2003) ein Verfahren vorgeschrieben.
5. Ein solches Verfahren wird von den meisten DBMS verwendet, ist aber in unterschiedlicher Weise verwirklicht.
6. Bei UNION, INTERSECT usw. müssen die ausgewählten Spalten von der Anzahl her übereinstimmen.
7. Bei UNION, INTERSECT usw. müssen die ausgewählten Spalten vom Datentyp her genau übereinstimmen.

Übung 2 – DISTINCT und ALL

Bitte überprüfen Sie die folgenden Befehle:

```
-- Variante 1
select distinct COUNT(*)
  from Mitarbeiter
 group by Abteilung_ID
-- Variante 2
select all COUNT(*)
  from Mitarbeiter
 group by Abteilung_ID
```

Worin unterscheidet sich die Ausgabe? Welche wichtige Information fehlt vor allem bei Variante 2?

Übung 3 – UNION, INTERSECT usw.

Was ist an den folgenden Befehlen falsch oder fragwürdig?

```
-- Befehl 1
select Name, Vorname                from Mitarbeiter
      union
select Name, Vorname, Geburtsdatum  from Versicherungsnehmer
-- Befehl 2
select Name, Vorname, Abteilung_ID  from Mitarbeiter
      union
select Name, Vorname, Geburtsdatum  from Versicherungsnehmer
```

Zusatzfrage: Wie kann man das DBMS dazu bringen, den „fragwürdigen“ Befehl auf jeden Fall auszuführen?

Übung 4 – UNION, INTERSECT usw.

Erstellen Sie eine Abfrage für die Tabellen *Versicherungsvertrag*, *Fahrzeug*, *Zuordnung_SF-FZ*, in der folgende Bedingungen berücksichtigt werden:

- Es sollen Vertragsnummer, Abschlussdatum, Kennzeichen sowie ggf. anteilige Schadenshöhe angezeigt werden.
- Fahrzeuge mit einem Schaden sollen vollständig angezeigt werden.
- Fahrzeuge ohne Schaden sollen nur angezeigt werden, wenn der Vertrag vor 1990 abgeschlossen wurde.
- Das Ergebnis soll nach Schadenshöhe und Kennzeichen sortiert werden.

Benutzen Sie UNION zur Verknüpfung der Verträge mit Schaden und ohne Schaden.

Übung 5 – Fallunterscheidung für Nachschlagewerte

Schreiben Sie einen SELECT-Befehl, bei dem die korrekte Briefanrede für die *Mitarbeiter* erstellt wird.

Hinweis: Benutzen Sie CONCAT zum Verknüpfen mehrerer Zeichenketten.

Übung 6 – Fallunterscheidung für Bereiche

Zur Tabelle *Fahrzeug* soll aus dem Kennzeichen die regionale Herkunft abgeleitet und angezeigt werden. Schreiben Sie eine Abfrage für diese Spalten *Kennzeichen* und *Herkunft*.

Hinweis: Benutzen Sie POSITION zur Feststellung des Bindestrichs sowie SUBSTRING.

Zusatzfrage: Wie müsste ein solches Problem sinnvollerweise gelöst werden, falls eine solche Zuordnung häufiger und allgemeiner benötigt wird?

Übung 7 – Fallunterscheidung für mehrere Varianten

Aus der Tabelle *Versicherungsnehmer* sollen Name, Vorname und Anschrift angezeigt werden. Außerdem soll jede Adresse eine Markierung bekommen: F=(eigene) Firmenkunden, P=(eigene) Privatkunden, X=eXterne Verträge

(d. h. Kunden fremder Versicherungsgesellschaften). Begründen Sie die Reihenfolge der WHEN-ELSE-Bedingungen.

Übung 8 – Fallunterscheidung beim Speichern

Schreiben Sie einen UPDATE-Befehl, der nach Ablauf eines Versicherungsjahres den *Prämiensatz* für das nächste Jahr ändert. Berücksichtigen Sie dabei folgende Bedingungen:

- Neue Verträge, für die noch kein Prämiensatz gilt, werden auf 200 [Prozent] gesetzt.
- Verträge mit einem Prämiensatz von mindestens 100 werden um 20 reduziert.
- Verträge mit einem Prämiensatz von weniger als 100 werden um 10 reduziert.
- Der Mindestsatz von 30 darf nicht unterschritten werden.

Ignorieren Sie dabei zunächst, dass dies nur im Fall von Schadensfreiheit gelten darf und innerhalb eines Jahres nur einmal neu berechnet werden darf.

Übung 9 – Fallunterscheidung beim Speichern

Bei genauerer Untersuchung von Übung 8 sind weitere Bedingungen im UPDATE-Befehl erforderlich.

1. In welcher Weise könnte die Schadensfreiheit berücksichtigt werden? Abhängig von der Höhe des Schadens soll sich der Prämiensatz gar nicht, wenig oder mehr erhöhen.
2. Wie kann unter Verwendung des Datums *Prämienänderung* gesichert werden, dass die Neuberechnung nur einmal jährlich stattfinden darf?

Versuchen Sie, dies im selben UPDATE-Befehl zu berücksichtigen. Sie sollen keinen Befehl schreiben, sondern die notwendigen Klauseln erwähnen und erläutern.

22.7. Lösungen

Lösung zu Übung 1 – Richtig oder falsch?

Die Aussagen 2, 5, 6 sind richtig. Die Aussagen 1, 3, 4, 7 sind falsch.

Lösung zu Übung 2 – DISTINCT und ALL

Bei Variante 1 wird jede Anzahl von Mitarbeitern pro Abteilung genau einmal aufgeführt.

Bei Variante 2 gibt es diese Anzeige für jede Abteilung einzeln. Dabei fehlt vor allem die Angabe der *Abteilung_ID*, ohne die die Ausgabe ziemlich sinnlos ist.

Lösung zu Übung 3 – UNION, INTERSECT usw.

Bei Befehl 1 werden unterschiedlich viele Spalten ausgewählt, das ist unzulässig.

Bei Befehl 2 unterscheiden sich die Datentypen der dritten Spalte; es ist unsicher, ob das DBMS die unterschiedlichen Spalten einheitlich als Zeichenkette (zulässig) oder nicht-kompatibel (unzulässig) interpretieren will.

Zur Zusatzfrage: Durch einen CAST beider Spalten auf VARCHAR werden die Datentypen kompatibel.

Lösung zu Übung 4 – UNION, INTERSECT usw.

Eine mögliche Variante lautet so (Firebird-Version für ORDER BY):

```
select * from (
  select Vertragsnummer, Abschlussdatum, Kennzeichen, Schadenshoehe
    from Versicherungsvertrag vv
      inner join Fahrzeug      fz on fz.ID = vv.Fahrzeug_ID
      right join Zuordnung_SF_FZ zu on fz.ID = zu.Fahrzeug_ID
  union
  select Vertragsnummer, Abschlussdatum, Kennzeichen, 0
    from Versicherungsvertrag vv
      inner join Fahrzeug      fz on fz.ID = vv.Fahrzeug_ID
  where Abschlussdatum < '01.01.1990'
    and not EXISTS ( select ID
                      from Zuordnung_SF_FZ zu
                      where zu.Fahrzeug_ID = fz.ID )
) order by Schadenshoehe, Kennzeichen;
```

Der zweite SELECT kann auch so geschrieben werden:

```
select Vertragsnummer, Abschlussdatum, Kennzeichen, 0
  from Versicherungsvertrag vv
    inner join Fahrzeug      fz on fz.ID = vv.Fahrzeug_ID
    left  join Zuordnung_SF_FZ zu on fz.ID = zu.Fahrzeug_ID
  where Abschlussdatum < '01.01.1990'
    and zu.ID is null
```


Lösung zu Übung 5 – Fallunterscheidung für Nachschlagewerte

```
select CONCAT( 'Sehr geehrte',
              CASE Geschlecht
                WHEN 'M' THEN 'r Herr '
                WHEN 'W' THEN ' Frau '
                ELSE '/r Frau/Herr '
              END,
              Name ) as Anrede
from Mitarbeiter;
```

Hinweise: Es kann passieren, dass CONCAT oder die String-Verknüpfung eine falsche Länge des ersten Teils erzeugt. Das kann mit TRIM „repariert“ werden; dann muss aber das abschließende Leerzeichen manuell hinzugefügt werden. Der ELSE-Zweig ist überflüssig, weil *Geschlecht* laut Definition nicht NULL sein kann; er wird nur der Vollständigkeit halber genannt.

Lösung zu Übung 6 – Fallunterscheidung für Bereiche

```
select Kennzeichen,
       CASE SUBSTRING(Kennzeichen from 1 for POSITION('-',
Kennzeichen)-1 )
         WHEN 'RE' THEN 'Kreis Recklinghausen'
         WHEN 'GE' THEN 'Stadt Gelsenkirchen'
         WHEN 'E'  THEN 'Stadt Essen'
         WHEN 'BO' THEN 'Stadt Bochum'
         WHEN 'HER' THEN 'Stadt Herne'
         ELSE 'unbekannt'
       END as Herkunft
from Fahrzeug
order by 2, 1;
```

Zur Zusatzfrage: Es sollte eine Tabelle *Region* o. ä. angelegt werden; zur Anzeige würde ein LEFT JOIN über den o.g. SUBSTRING verwendet. Man könnte auch einen INNER JOIN benutzen, dann würden aber Fahrzeuge fehlen, deren Kennzeichen in der Tabelle *Region* fehlen.

Lösung zu Übung 7 – Fallunterscheidung für mehrere Varianten

```
select Name, Vorname, PLZ, Strasse, Hausnummer,
       CASE
         WHEN Eigener_Kunde = 'N' THEN 'X'
         WHEN Vorname is null or Fuehrerschein is null THEN 'F'
```

```
ELSE 'P'  
END as Markierung  
from Versicherungsnehmer;
```

Zur Reihenfolge: Bei den Fremdverträgen wird nicht zwischen Firmen- und Privatkunden unterschieden; dies muss also zuerst geprüft werden. Bei Firmenkunden sind weder Vorname noch Führerschein-Datum registriert, das wird als nächstes geprüft (eines der WHEN-Kriterien würde genügen). Alle übrigen Fälle sind Privatkunden.

Lösung zu Übung 8 – Fallunterscheidung beim Speichern

```
update Versicherungsvertrag  
set Praemiensatz = CASE  
    WHEN Praemiensatz is null THEN 200  
    WHEN Praemiensatz >= 100 THEN Praemiensatz  
- 20  
    WHEN Praemiensatz >= 40 THEN Praemiensatz  
- 10  
    ELSE 30  
END  
where -- passende Bedingungen
```

Lösung zu Übung 9 – Fallunterscheidung beim Speichern

1. Es wird eine geschachtelte CASE-Anweisung verwendet. Zunächst wird unterschieden, ob überhaupt Schadensfälle aufgetreten sind. Im ersten Zweig „kein Schaden“ wird die Lösung von Übung 7 benutzt. Im zweiten Zweig „mit Schaden“ wird eine ähnliche CASE-Anweisung eingefügt, die abhängig von der Schadenshöhe den Prämienersatz erhöht.
2. In der WHERE-Klausel wird geprüft, ob die letzte Prämienänderung überschritten ist. Gleichzeitig mit der Neuberechnung wird dieses Datum um ein Jahr weitersetzt.

Hinweis: Dieses Verfahren wird ausführlich im Kapitel PROZEDUREN: AUTOMATISCHES UPDATE¹ behandelt. Bei Interesse können Sie im dortigen Code den abschließenden UPDATE-Befehl vergleichen.

1 Kapitel 31 auf Seite 399

23. Berechnete Spalten

Mit Abfragen kann man nicht nur Spalten auswählen, sondern auch völlig neue Spalten aus anderen Spalten oder mit Funktionen erstellen.

23.1. Allgemeine Regel

Eine neue Spalte wird als Teil einer Abfrage wie folgt erstellt:

```
<Ausdruck> [AS] <Name der neuen Spalte>
```

Ausdruck ist allgemein der Hinweis auf etwas, das an der betreffenden Stelle verwendet wird: ein konstanter Wert, der Inhalt einer oder mehrerer Spalten, eine Berechnung mit diesen Spalten, das Ergebnis einer Funktion oder das Ergebnis einer Unterabfrage. Der Spalte, die das Ergebnis aufnimmt, wird mit **AS** ein eigener Name, ein **Alias** zugewiesen; das **AS** kann auch entfallen.

In manchen DBMS ist bei berechneten Spalten die Angabe eines Alias Pflicht, und das Vergessen führt zu einem Fehler, manche erzeugen automatisch einen zufälligen Namen, wenn kein Alias eingegeben wird, und andere meckern erst, wenn man versucht, eine physische Tabelle mit einer namenlosen Spalte zu füllen.

Wenn eine Unterabfrage als berechnete Spalte benutzt wird, ist es unbedingt wichtig, dass sie – wie bei einer Funktion, Berechnung oder Verknüpfung – für jede Zeile der Hauptabfrage genau einen Wert ergibt. In dem untenstehenden Beispiel wird dies durch eine passende WHERE-Bedingung erreicht.

Hinweis: Fast alles, was in diesem Abschnitt über Ausdrücke, Funktionen und Konvertierungen gesagt wird, ist auch bei der Manipulation oder Verarbeitung von Daten wichtig. Die Funktionen und Konvertierungen wurden bereits in den Kapiteln FUNKTIONEN¹ und FUNKTIONEN (2)² behandelt.

1 Kapitel 13 auf Seite 121

2 Kapitel 15 auf Seite 157

23.2. Ergebnis von Berechnungen

Alle Werte einer Spalte werden für eine Berechnung verwendet, vorzugsweise eine der Grundrechenarten. Das folgende Beispiel errechnet für alle Zeilen der Tabelle *Schadensfall* aus dem Feld *Schadenshoehe* den Nettobetrag (ohne MWSt) und fügt alle weiteren Felder an:

```
SELECT Schadenshoehe / 1.19 AS Netto,
       *
FROM Schadensfall;
```

Dieses Ergebnis kann auch für weitere Berechnungen verwendet werden, wenn auch nicht direkt mit dem Alias:

Quelltext

Falsch

```
SELECT Schadenshoehe           AS Brutto,
       Schadenshoehe / 1.19    AS Netto,
       Schadenshoehe - Netto  AS MWSt
FROM Schadensfall;
```

Der Alias-Name der neuen Spalte Netto kann nicht nochmals verwendet werden.

```
SELECT Schadenshoehe           AS Brutto,
       Schadenshoehe / 1.19    AS Netto,
       Schadenshoehe - (Schadenshoehe / 1.19) AS MWSt
FROM Schadensfall;
```

Aber die Berechnung kann erneut zugewiesen werden.

Man kann davon ausgehen, dass das DBMS die Abfrage soweit optimiert, dass die Berechnung tatsächlich nur einmal ausgeführt werden muss.

23.3. Zeichenketten verbinden und bearbeiten

Ebenso wie Zahlen für Berechnungen verwendet werden, gibt es die Bearbeitung von Zeichenketten. Das einfachste ist die Verknüpfung (Verkettung) von Strings durch den **Operator** `||` bzw. `+`:

```
SELECT Name || ', ' || Vorname AS Gesamtname  
FROM Mitarbeiter;
```

```
Gesamtname  
Müller, Kurt  
Schneider, Daniela
```

In diesem Beispiel wird zwischen die Spalten noch das Komma und ein Leerzeichen als konstanter Text gesetzt.

Bei Textspalten mit fester Feldlänge, die wir in der Beispieldatenbank nicht haben, sollten überzählige Leerzeichen mit der Funktion TRIM abgeschnitten werden:

```
SELECT TRIM(Name) || ', ' || TRIM(Vorname) AS Gesamtname  
FROM Mitarbeiter;
```

Auf diese Weise kann man auch Teile von Texten übernehmen und den Rest abschneiden:

```
SELECT Name || ', ' || TRIM( SUBSTRING(Vorname FROM 1 FOR 1) ) || '.' AS  
Name_kurz  
FROM Mitarbeiter;
```

```
Name_kurz  
Müller, K.  
Schneider, D.
```

Hinweis: Nach SQL-Standard liefert das Ergebnis von SUBSTRING einen Text von gleicher Länge wie der ursprüngliche Text; die jetzt zwangsläufig folgenden Leerzeichen müssen mit TRIM entfernt werden.

23.4. Ergebnis von Funktionen

In den bisherigen Beispielen werden Feldinhalte direkt modifiziert und ausgegeben. In den beiden Kapiteln über „Funktionen“ werden neue Informationen berechnet und unabhängig von vorhandenen Zeilen und Spalten ausgegeben:

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

Bei diesem Beispiel mit einer Aggregatfunktion besteht das Ergebnis aus einer einzelnen Zahl in einer einzigen Zeile. Dies muss aber nicht so sein; vor allem mit GRUPPIERUNGEN³ gibt es viele Varianten.

Viele andere Funktionen können besondere Abfragen ermöglichen oder stark erleichtern. EXTRACT nach Monat liefert die Liste aller Geburtstage, sortiert nach Monat:

```
SELECT Name, Vorname,
       EXTRACT(MONTH FROM Geburtsdatum) AS Monat,
       Geburtsdatum
FROM Mitarbeiter
ORDER BY Monat;
```

NAME	VORNAME	MONAT	GEBURTSDATUM
Agenau	Karolin	1	02.01.1950
Langer	Norbert	1	13.01.1968
Wagner	Gaby	1	18.01.1970
Müller	Kurt	1	05.01.1977
Kolic	Ivana	2	14.02.1971
Schneider	Daniela	2	16.02.1980 // usw.

23.5. Unterabfragen

Diese werden ausführlich im Kapitel UNTERABFRAGEN⁴ besprochen. Hier soll deshalb ein Beispiel genügen, wie es für eine berechnete Spalte geeignet ist. ►

Aufgabe: Gesucht sind alle Abteilungen zusammen mit der Anzahl ihrer Mitarbeiter:

```
SELECT ab.ID,
       ab.Kuerzel,
       ab.Ort,
       ( SELECT COUNT(*)
         FROM Mitarbeiter mi
         WHERE mi.Abteilung_ID = ab.ID
       ) AS Mitarbeiterzahl
FROM Abteilung ab
```

3 Kapitel 24 auf Seite 271

4 Kapitel 25 auf Seite 283

;

Wir betrachten diese Abfrage von innen nach außen: In den Klammern werden bei jeder ab.ID die Anzahl der Einträge in *Mitarbeiter* mit dieser Abteilung gezählt; dieser Wert wird mit den anderen Spalten von *Abteilung* zusammengefasst. Weil die Spalte *ID* in beiden Tabellen vorkommt, muss die gewünschte Tabelle ausdrücklich erwähnt werden.

23.6. Zusammenfassung

In diesem Kapitel bekamen wir verschiedene Erläuterungen dafür, wie aus Berechnungen, Verkettung von Zeichenketten oder als Ergebnis von Skalar- oder Spaltenfunktionen neue Spalten für die Ergebnistabelle entstehen.

23.7. Übungen

Übung 1 – Zusatzspalten durch Berechnung

Erstellen Sie zur Tabelle *Versicherungsvertrag* eine Abfrage, die (neben der Versicherungsnummer) die Basisprämie und den Prämienatz angibt sowie die aktuelle Prämie berechnet.

Übung 2 – Zusatzspalten durch Berechnung

Geben Sie (unter Verwendung der Lösung von Übung 1) die Gesamtzahl der Versicherungsverträge sowie den Gesamtbetrag aller aktuellen Prämien an.

Übung 3 – Zusatzspalten durch String-Verknüpfung

Erstellen Sie zur Tabelle *Versicherungsnehmer* eine Abfrage, die die druckfertige Anschrift zurückgibt:

- Zeile 1 mit Anrede (basierend auf der Spalte *Geschlecht* mit den Inhalten 'W' bzw. 'M') und der ID am rechten Rand – insgesamt 22 Zeichen für die Anrede und 8 Zeichen für die ID (rechtsbündig)
- Zeile 2 mit Vorname und Name
- Zeile 3 mit Straße und Hausnummer

- Zeile 4 mit PLZ und Ort

Benutzen Sie *CASE*, *CAST*, *RPAD* und *CONCAT*; auf *TRIM* können Sie verzichten.

Übung 4 – Neue Spalten durch Spaltenfunktion

Bestimmen Sie, wie viele Fahrzeuge in Bochum ('BO') und wie viele in Gelsenkirchen ('GE') angemeldet sind.

Gruppierungen werden erst im nächsten Kapitel behandelt; verwenden Sie stattdessen *UNION*.

23.8. Lösungen

Lösung zu Übung 1 – Zusatzspalten durch Berechnung

```
select Vertragsnummer,
       Basispraemie,
       Praemiensatz,
       Basispraemie * Praemiensatz / 100 as Aktuell
from Versicherungsvertrag;
```

Lösung zu Übung 2 – Zusatzspalten durch Berechnung

```
select COUNT(*) as Gesamtzahl,
       SUM(Basispraemie * Praemiensatz / 100) as Praemienummer
from Versicherungsvertrag;
```

Lösung zu Übung 3 – Zusatzspalten durch String-Verknüpfung

```
select CONCAT( CAST( ( CASE Geschlecht
                       WHEN 'M' THEN 'Herrn'
                       WHEN 'W' THEN 'Frau'
                       ELSE ''
                     END ) AS CHAR(22) ),
              RPAD( CAST( ID as VARCHAR(8) ), 8 )
            ) as Zeile1,
       CASE
         WHEN Vorname is null THEN Name
```



```
        ELSE CONCAT (Vorname, ' ', Name)
      END
    CONCAT (Strasse, ' ', Hausnummer) as Zeile3,
    CONCAT (PLZ, ' ', Ort)          as Zeile4
  from Versicherungsnehmer vn;
```

Lösung zu Übung 4 – Neue Spalten durch Spaltenfunktion

```
select COUNT(*) as Anzahl, 'BO' as Kreis
  from Fahrzeug
 where Kennzeichen STARTING WITH 'BO-'
        union
select COUNT(*), 'GE'
  from Fahrzeug
 where Kennzeichen STARTING WITH 'GE-';
```

Für **STARTING WITH** gibt es Alternativen; wir haben verschiedentlich **SUBSTRING** mit **POSITION** o. a. verwendet.

24. 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.

24.1. Syntax von **GROUP BY**

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.

24.2. Gruppierung bei einer Tabelle

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

Aufgabe: 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	ANZAHL
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. ► **Aufgabe:** Berechne die mittlere Schadenshöhe für die Schadensfälle mit und ohne Personenschäden.

```
SELECT Verletzte,  
       AVG(Schadenshoehe) AS Mittlere_Schadenshoehe  
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.

24.3. Gruppierung über mehrere Tabellen

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

► **Aufgabe:** 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 Fahrzeughersteller 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 *Fahrzeughersteller* 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,
       EXTRACT(YEAR FROM 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).

Siehe dazu die Erläuterungen unter *Einschränkungen*.

24.4. Einschränkungen

Wie das letzte Beispiel zeigt, muss die GROUP BY-Klausel gewisse Bedingungen erfüllen. Auch dafür gilt: Jedes DBMS weicht an manchen Punkten 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, der abgeleiteten Tabellen oder der 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.

Dies entspricht der gleichen Einschränkung, die bei den unter „Ergebnis von Berechnungen“ im Kapitel **BERECHNETE SPALTEN**¹ genannt ist.

Mit der Beispieldatenbank sind keine passenden Beispiele möglich; wir müssen uns deshalb auf fiktive Tabellen und Spalten beschränken:

Die folgenden Abfragen sind nicht zulässig.

Quelltext	Falsch
<pre>SELECT Spalte1, Spalte2 FROM T GROUP BY Spalte1 + Spalte2; SELECT Spalte1 + constant + Spalte2 FROM T GROUP BY Spalte1 + Spalte2;</pre>	

Die folgenden Abfragen sind zulässig.

<pre>SELECT Spalte1, Spalte2 FROM T GROUP BY Spalte1, Spalte2; SELECT Spalte1 + Spalte2 FROM T GROUP BY Spalte1, Spalte2; SELECT Spalte1 + Spalte2 FROM T GROUP BY Spalte1+ Spalte2; SELECT Spalte1 + Spalte2 + constant FROM T GROUP BY Spalte1, Spalte2;</pre>	
--	--

- GROUP BY kann nicht benutzt werden in einem SELECT-Befehl mit folgenden Bedingungen:
 - Der Befehl enthält eine INTO-Klausel (d. h. er wird benutzt, um einzelne Zeilen zu verarbeiten – dies wird in dieser Einführung nicht erläutert).
 - Der Befehl enthält eine Unterabfrage oder eine VIEW, die selbst mit einer GROUP BY- oder HAVING-Klausel arbeitet.
- Jeder SELECT-Befehl darf einschließlich aller Unterabfragen höchstens eine GROUP BY-Klausel enthalten.

Zu dieser Bedingung ist den Autoren kein vernünftiges Beispiel eingefallen. Eines, das dafür konstruiert wurde, hat Firebird trotz klarer Verletzung ausgeführt,

¹ Kapitel 23 auf Seite 263

wahrscheinlich weil es sich wegen der anderen Bedingungen sowieso nur um jeweils eine Zeile handelte und keine Gruppierung erforderlich war.

24.5. Erweiterungen

24.5.1. Zwischensummen mit CUBE

Diese Erweiterung steht nur in manchen DBMS zur Verfügung. Es soll deshalb ein kurzer Hinweis genügen.

```
GROUP BY CUBE (<spaltenliste>)
```

Mit diesem Befehl werden zusätzlich zu den normalerweise von GROUP BY erstellten Zeilen auch Zwischensummen in das Ergebnis aufgenommen. Für jede mögliche Kombination von Spalten in der <spaltenliste> wird eine eigene GROUP BY-Summenzeile zurückgegeben.

Erläuterungen und Beispiele sind zu finden z.B. unter [GROUP BY \(MICROSOFT\)](#)² und weiteren Links.

24.5.2. Gesamtsummen mit ROLLUP

Auch diese Erweiterung steht nur in manchen DBMS zur Verfügung. Es soll wiederum ein kurzer Hinweis genügen.

```
GROUP BY <spaltenliste> WITH ROLLUP
```

Mit diesem Befehl werden zusätzlich zu den von GROUP BY erstellten Zeilen auch Gesamtsummen in das Ergebnis aufgenommen.

Erläuterungen und Beispiele sind zu finden z. B. unter [GROUP BY \(MYSQL\)](#)³.

2 [HTTP://MSDN.MICROSOFT.COM/DE-{}DE/LIBRARY/MS177673%28SQL.90%29.ASPX](http://msdn.microsoft.com/de-{}de/library/ms177673%28SQL.90%29.aspx)

3 [HTTP://DEV.MYSQL.COM/DOC/REFMAN/5.1/DE/GROUP-{}BY-{}MODIFIERS.HTML](http://dev.mysql.com/doc/refman/5.1/de/group-{}by-{}modifiers.html)

24.5.3. Das Ergebnis wird eingeschränkt mit 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 (siehe oben unter „Gruppierung bei einer Tabelle“) 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;
```

```
Herne      1
```

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

Hinweis: Die HAVING-Klausel wird als letzter Teil des SELECT-Befehls ausgeführt. Es ist deshalb zu empfehlen, alle möglichen Einschränkungen vorher zu regeln, z. B. als Teil von WHERE-Bedingungen. Nur wenn – wie bei Aggregatfunktionen – diese Einschränkung erst am Schluss geprüft werden kann, ist HAVING zu benutzen.

Quelltext

Falsch

```
SELECT Abteilung_ID, COUNT(*)
  FROM MITARBEITER
  group by Abteilung_ID
  having Abteilung_ID < 3;
```

Möglich, aber nicht günstig.

SQL Quelltext

```
SELECT Abteilung_ID, COUNT(*)  
FROM MITARBEITER  
group by Abteilung_ID  
where Abteilung_ID < 3;
```

Besseres Verfahren, weil weniger Datensätze geprüft werden müssen.

24.6. Zusammenfassung

In diesem Kapitel lernten wir Einzelheiten über die Gruppierung von Abfrageergebnissen.

- Dies wird meistens gleichzeitig mit ORDER BY und in Verbindung mit Aggregatfunktionen verwendet.
- Die Gruppierung ist auch über mehrere Tabellen hinweg möglich.
- Einige wichtige Einschränkungen sind zu beachten; vor allem sind die Felder aus der Spaltenliste in der Regel auch unter GROUP BY aufzuführen.

Die HAVING-Klausel kann das Abfrageergebnis einschränken, sollte aber zurückhaltend benutzt werden.

24.7. Übungen

Übung 1 – Definitionen

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 dem *PLZ-Bereich*, nämlich 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.

Übung 2 – Gruppierung für 1 Tabelle

Bestimmen Sie die Anzahl der Kunden (Versicherungsnehmer) in jedem Briefzentrum (d. h. die Ziffern 1 und 2 der PLZ).

Übung 3 – Gruppierung für 1 Tabelle

Bestimmen Sie, wie viele Fahrzeuge in jedem Kreis angemeldet sind.

Übung 4 – Gruppierung für mehrere Tabellen

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*.

Übung 5 – Gruppierung für mehrere Tabellen

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.

24.8. Lösungen

Lösung zu Übung 1 – Definitionen

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

Lösung zu Übung 2 – Gruppierung für 1 Tabelle

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

Hinweis: Hierbei handelt es sich um die korrekte Version zur Frage 5 aus Übung 1.

Lösung zu Übung 3 – Gruppierung für 1 Tabelle

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

Hinweis: Hierbei handelt es sich um die vollständige Version der letzten Übung zu „berechneten Spalten“.

Lösung zu Übung 4 – Gruppierung für mehrere Tabellen

```
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 Fahrzeughersteller fh
    on fh.ID = temp.ID
 order by Name;
```

Lösung zu Übung 5 – Gruppierung für mehrere Tabellen

```
select EXTRACT(YEAR from sf.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*.

24.9. Siehe auch

Ergänzende Informationen sind in diesen Kapiteln zu finden:

- DATENTYPEN⁴ informiert auch über besondere Sortierungen einer einzelnen Spalte.
- MEHRERE TABELLEN⁵ verknüpfen

4 Kapitel 12 auf Seite 107

5 Kapitel 17 auf Seite 187

25. Unterabfragen

Immer wieder werden zur Durchführung einer Abfrage oder eines anderen Befehls Informationen benötigt, die zuerst durch eine eigene Abfrage geholt werden müssen. Diese „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 immer in Klammern gesetzt wird. Auch wenn ein DBMS das nicht verlangen sollte, ist es wegen der Übersichtlichkeit dringend zu empfehlen.

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

25.1. Ergebnis als einzelner Wert

25.1.1. Ergebnisse einfacher Abfragen

Immer wieder kennt der Anwender den Namen eines Objekts, benötigt aber für Abfragen die ID. Diese holt er sich mit einer Unterabfrage und übergibt das Ergebnis an die eigentliche Abfrage.

- **Aufgabe:** Nenne alle Mitarbeiter der Abteilung „Schadensabwicklung“.
- Lösung Teil 1: Hole die ID dieser Abteilung anhand des Namens.
 - Lösung Teil 2: Hole die Mitarbeiter dieser 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

Teil 1 der Lösung ist der SELECT-Befehl innerhalb der Klammern. Das Ergebnis ist eine einzelne ID. Diese kann anstelle einer konkreten Zahl in die WHERE-Klausel der eigentlichen Abfrage übernommen werden. *Das Wörtchen „Diese“ hat in diesem Fall sprachlich eine doppelte Bedeutung: zum einen steht es für die Unterabfrage, zum anderen für die ID als Ergebnis.*

Hinweis: Dies funktioniert nur deshalb auf einfache Weise, weil die Kurzbezeichnung faktisch eindeutig ist und deshalb genau eine ID geliefert wird. Wenn wir uns darauf nicht verlassen wollen oder können oder wenn das DBMS „empfindlich“ ist und die Eindeutigkeit des Ergebnisses nicht erkennt, können wir daraus bewusst einen einzelnen Wert machen:

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

Eine solche Aufgabe kann auch zweimal dieselbe Tabelle benutzen. ► **Aufgabe:** 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.

Zur Sicherheit prüfen wir auch die Eigenschaft *Ist_Leiter*.

```
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' );

```

PERSONALNUMMER	NAME	VORNAME
80002	Aliman	Zafer
80003	Langer	Norbert
80004	Kolic	Ivana

25.1.2. Ergebnisse von Spaltenfunktionen

Häufig werden Ergebnisse von Aggregatfunktionen als Teil der WHERE-Klausel benötigt. ► **Aufgabe:** Nenne alle Schadensfälle mit unterdurchschnittlicher Schadenshöhe.

- Lösung Teil 1: Berechne den Durchschnitt der Schadenshöhe aller Schadensfälle.
- Lösung Teil 2: Übernimm dieses Ergebnis als Vergleichswert in die Hauptabfrage.

```

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

```

ID	DATUM	ORT	SCHADENSHOEHE
1	03.02.2007	Recklinghausen, Bergknappenstr. 144	1.234,50
2	11.07.2007	Haltern, Hauptstr. 46	2.066,00
4	27.05.2008	Recklinghausen, Südgrabenstr. 23	1.438,75
5	05.10.2008	Dorsten, Oberhausener Str. 18	1.983,00
7	21.06.2009	Recklinghausen, Bergknappenstr. 144	865,00

► **Aufgabe:** 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 Jahr innerhalb 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 EXTRACT(YEAR from sf2.Datum) =
EXTRACT(YEAR from sf.Datum)
                             )
        ) <= 300;
```

ID	DATUM	SCHADENSHOEHE	JAHR
2	11.07.2007	2.066,00	2007
4	27.05.2008	1.438,75	2008
5	05.10.2008	1.983,00	2008
8	01.08.2009	2.471,50	2009

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 ist eine der JOIN-Varianten oder eine der Lösungen im Abschnitt „Ergebnis in Form einer Tabelle“ (siehe unten).

25.2. Ergebnis als Liste mehrerer Werte

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

► **Aufgabe:** Bestimme alle Fahrzeuge eines bestimmten Herstellers.

- Lösung Teil 1: Hole die ID des gewünschten Herstellers.
- Lösung Teil 2: Hole alle IDs der Tabelle *Fahrzeugtyp*, die zu dieser Hersteller-ID gehören.

- Lösung Teil 3: Hole alle Fahrzeuge, die zu dieser Liste von Fahrzeugtypen-IDs gehören.

```
select ID, Kennzeichen, Fahrzeugtyp_ID as TypID
  from Fahrzeug
 where Fahrzeugtyp_ID in
    ( select ID
      from Fahrzeugtyp
     where Hersteller_ID =
       ( select ID
         from Fahrzeughersteller
        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, wie es im Kapitel MEHRERE TABELLEN¹ erläutert wird; es ist auch ein Verfahren möglich, wie es unten im Abschnitt „Ergebnis in Form einer Tabelle“ erläutert wird.

Das obige Beispiel mit der durchschnittlichen Schadenshöhe kann auch so gebaut werden: ► **Aufgabe:** 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
```

1 Kapitel 17 auf Seite 187

Unterabfragen

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

ID	DATUM	ORT	BESCHREIBUNG	SCHADENSHOEHE	VERLETZTE	MIT-ID
4	27.05.2008	Recklinghausen,	Fremdes parkendes			
1.438,75		N 16				
		Südgrabenstr. 23	Auto gestreift			
5	05.10.2008	Dorsten,	beim Ausparken hat			
1.983,00		N 14				
		Oberhausener Str. 18	ein fremder Wagen die			
			Vorfahrt missachtet			

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.

Ganz offensichtlich ist dieses Beispiel konstruiert: Weil immer dieselbe Tabelle verwendet wird, kann die WHERE-Klausel der Unterabfrage in der äußersten Klammer auch als WHERE-Klausel der Hauptabfrage verwendet werden (die Einrückungen wurden zum besseren Vergleich nicht geändert):

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

25.3. Ergebnis in Form einer Tabelle

Das Ergebnis einer Abfrage kann innerhalb der eigentlichen Abfrage 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 selbständig 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.

Überarbeiten wir jetzt, wie oben angekündigt, einige Beispiele. Dabei wird die Unterabfrage, die bisher zur WHERE-Klausel gehörte, als Tabelle in die FROM-Klausel eingebaut.

► **Aufgabe:** 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 zur obigen Lösung: Die Durchschnittswerte werden einmalig zusammengestellt und nur noch abgerufen; sie müssen nicht bei jedem Datensatz neu (und ständig wiederholt) berechnet werden.

► **Aufgabe:** 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.

Hinweis: Mit den Möglichkeiten des nächsten Kapitels ERSTELLEN VON VIEWS² ergeben sich wesentliche Verbesserungen: Mit einer VIEW lassen sich Unterabfragen, die – wie die Liste von Typen und Herstellern – immer wieder benötigt werden, dauerhaft bereitstellen. Und mit einer Inline-View werden verschachtelte Abfragen deutlich übersichtlicher.

25.4. Verwendung bei Befehlen zum Speichern

Bisher hatten wir Abfragen als Teil von anderen Abfragen benutzt; deshalb wird das Ganze auch als „Unterabfrage“ bezeichnet. Man kann das Ergebnis einer Abfrage aber auch zum Speichern verwenden: sowohl für einen einzelnen Wert als auch als vollständigen Datensatz.

25.4.1. Verwendung bei INSERT INTO ... SELECT

Eine Abfrage in einen INSERT-Befehl einzubinden, ist eines der Standardverfahren für INSERT:

```
INSERT INTO <zieltabelle>
  ( <spaltenliste> )
SELECT <spaltenliste>
  FROM <quelltabelle/n>
  [ <weitere Festlegungen> ]
  [ WHERE <bedingungen> ]
```

Der SELECT-Befehl kann dabei beliebig aufgebaut sein: Daten aus einer oder mehreren Tabellen holen, mit oder ohne Einschränkungen, mit oder ohne weitere Festlegungen. Lediglich drei Punkte sind zu beachten:

- Die Spaltenliste in der Zieltabelle muss mit den Spalten in der SELECT-Auflistung genau übereinstimmen; genauer: Die Datentypen müssen zueinander passen, also gleich sein oder automatisch konvertiert werden können.
- Es ist möglich, Daten aus der Zieltabelle zu holen und somit zu verdoppeln. Dann muss die ID automatisch vergeben werden können; und es ist unbedingt mit WHERE zu arbeiten, weil es andernfalls zu einer Endlosschleife kommen kann (siehe das erste nachfolgende Beispiel).
- ORDER BY kann nicht sinnvoll genutzt werden, weil das Ergebnis in die Zieltabelle eingefügt wird, wodurch eine Sortierung sowieso verlorenginge.

Um weitere Testdaten zu erhalten, könnte so verfahren werden: ► **Aufgabe:** Kopiere die vorhandenen Schadensfälle.

```
INSERT into Schadensfall
  ( Datum, Ort, Beschreibung, Schadenshoehe, Verletzte, Mitarbeiter_ID
  )
SELECT Datum, Ort, Beschreibung, Schadenshoehe, Verletzte, Mitarbeiter_ID
  from Schadensfall
 where ID < 10000;
```

Auf die doppelte Angabe der Spalten kann nicht verzichtet werden, weil *ID* nicht benutzt werden darf (sie soll automatisch neu vergeben werden) und das DBMS wissen muss, welche Werte wie zugeordnet werden sollen. Auf diese Weise kann man ganz leicht 100 oder 1000 Testdatensätze erzeugen. Für den produktiven Betrieb wird man diese Syntax wohl eher seltener brauchen.

In einem ersten Versuch fehlte die WHERE-Bedingung; erwartet wurde, dass nur die vor dem Befehl vorhandenen Datensätze bearbeitet würden. Tatsächlich hatte Firebird endlos kopiert, bis mit **Strg** + **Alt** + **Entf** der „Stecker gezogen“ wurde. Danach war die Datenbank von 3 MB auf 740 MB aufgebläht worden und (natürlich) beschädigt, sodass auf diese Tabelle nicht mehr richtig zugegriffen werden konnte. Weitere Versuche mit WHERE-Bedingung arbeiteten wie vorgesehen: nur die vorhandenen Datensätze wurden einmalig kopiert.

Die „neuen“ Daten können auch aus einer anderen Tabelle geholt und mit konstanten Werten gemischt werden, wie es in der Beispieldatenbank geschieht: ► **Aufgabe:** Jeder Abteilungsleiter erhält einen persönlichen Dienstwagen.


```
INSERT INTO Dienstwagen
  ( Kennzeichen, Farbe, Fahrzeugtyp_ID, Mitarbeiter_ID )
SELECT 'DO-WB 42' || Abteilung_ID,
      'elfenbein', 14, ID
FROM Mitarbeiter
WHERE Ist_Leiter = 'J';
```

Die Spaltenliste der Zieltabelle *Dienstwagen* enthält alle Spalten mit Ausnahme der ID; diese wird automatisch vergeben. Diesen Spalten werden die Spalten der SELECT-Abfrage zugewiesen:

- Die *Mitarbeiter_ID* ist das einzige Feld aus der Quelltablelle *Mitarbeiter*, das unbedingt benötigt wird und übernommen werden muss.
- Die Farbe wird als Konstante eingetragen: Alle Abteilungsleiter bekommen den gleichen Wagentyp.
- Der Fahrzeugtyp wird ebenso als Konstante eingetragen. Er könnte auch durch eine Unterabfrage wie oben als Ergebnis einer einfachen Abfrage bestimmt werden.
- Für das Kennzeichen ist eine String-Verknüpfung vorgesehen, bei der zusätzlich die *Abteilung_ID* übernommen wird. *Bitte benutzen Sie die für Ihr DBMS richtige Art der String-Verknüpfung.*

Die WHERE-Bedingung sichert, dass nur Abteilungsleiter „automatisch“ einen Dienstwagen erhalten.

25.4.2. Verwendung bei INSERT INTO ... VALUES

Die andere Version des INSERT-Befehls arbeitet mit einer Liste von Werten, die direkt angegeben werden:

```
INSERT INTO <zieltabelle>
  [ ( <spaltenliste> ) ]
VALUES ( <werteliste> )
```

Ein einzelner Befehl sieht dabei wie folgt aus: ► **Aufgabe:** Der Mitarbeiter 2 bekommt einen gelben Dienstwagen Typ 2 mit einem bestimmten Kennzeichen.

```
INSERT INTO Dienstwagen
  ( Kennzeichen, Farbe, Fahrzeugtyp_ID, Mitarbeiter_ID )
VALUES ( 'DO-WB 202', 'gelb', 2, 2 );
```

Versuchen wir, diesen Befehl variabel zu gestalten, sodass *Mitarbeiter_ID* und *Kennzeichen* (wie oben) aus einer Abfrage kommen können. Dazu setzen wir diese beiden Spalten hintereinander und ordnen diesen das „Ergebnis als Liste mehrerer Werte“ zu unter Verwendung der LPAD-Funktion (siehe FUNKTIONEN (2)³).

Quelltext

Falsch

```
INSERT INTO Dienstwagen
      ( Kennzeichen, Mitarbeiter_ID, Farbe, Fahrzeugtyp_ID )
VALUES ( ( SELECT 'DO-WB 2' || LPAD(ID, 2, '0'), ID
          FROM Mitarbeiter
          WHERE ID = 2 ),
        'gelb', 2 );
```

```
SQL error code = -804.
Count of read-write columns does not equal count of values.
```

Aha, der SELECT-Befehl wird nur als *ein* Wert interpretiert, obwohl er zwei passende Werte liefert. Können wir die Abfrage an beiden Stellen verwenden?

```
INSERT INTO Dienstwagen
      ( Kennzeichen, Mitarbeiter_ID, Farbe, Fahrzeugtyp_ID )
VALUES ( ( SELECT 'DO-WB 2' || LPAD(ID, 2, '0') FROM Mitarbeiter WHERE ID
        = 2 ),
        ( SELECT ID
          FROM Mitarbeiter WHERE ID
        = 2 ),
        'gelb', 2 );
```

So funktioniert es, aber das ist natürlich nicht schön, wenn eine fast identische Abfrage doppelt auftauchen muss (auch wenn davon auszugehen ist, dass ein DBMS einen vernünftigen Ausführungsplan erstellt). Dann ist die obige Version mit INSERT INTO ... SELECT mit einer Mischung aus Konstanten und Tabellenspalten die bessere Fassung.

25.4.3. Verwendung bei UPDATE

Schauen wir uns die grundsätzliche Struktur eines UPDATE-Befehls an:

```
UPDATE <tabellenname>
  SET <spalte1> = <wert1> [ ,
      <spalte2> = <wert2> ]
  WHERE <bedingungsliste>;
```

Daraus ergibt sich, dass Abfragen benutzt werden können, um einen oder mehrere Werte zu speichern oder um Vergleichswerte für die Bedingungsliste zu liefern (ebenso wie in verschiedenen früheren Beispielen).

► **Aufgabe:** Alle Mitarbeiter mit Dienstort „Bochum“ bekommen eine neue Telefonnummer, die neben der Zentrale die *Abteilung* und die *Personalnummer* enthält.

- Lösung 1: Die WHERE-Bedingung muss die betreffenden Datensätze (genauer: die IDs) der Tabelle *Abteilung* prüfen und vergleichen.
- Lösung 2: Der zugeordnete neue Wert muss passend gestaltet werden. (LPAD und SUBSTRING werden nur verwendet, damit mit festen Längen gearbeitet werden kann.)

```
update Mitarbeiter
  set Telefon = '0234/66'
      || LPAD(Abteilung_ID, 3, '0')
      || SUBSTRING(LPAD(Personalnummer, 6, '0') from 4 for 3)
  where Abteilung_ID in ( SELECT ID
                        from Abteilung
                        where Ort = 'Bochum' );
```

Diese Lösung enthält nichts Neues: Die Abfrage wird mit der IN-Abfrage in die WHERE-Klausel eingebunden; die neuen Werte werden aus den vorhandenen (Abteilung und Personalnummer) gebildet.

► **Aufgabe:** Die Abteilung „Ausbildung“ soll dorthin umziehen, wo die Abteilung „Personalverwaltung“ sitzt.

- Lösung: Der gewünschte Ort wird aus einer Abfrage geholt.

```
update Abteilung
  set Ort = ( SELECT Ort
            from Abteilung
            where Kuerzel = 'Pers' )
  where Kuerzel = 'Ausb';
```

Bitte wundern Sie sich nicht über ein solch konstruiertes Beispiel; der neue Ort könnte natürlich im Klartext angegeben werden. Bei der vorgegebenen Datenstruktur ist es manchmal schwer, sinnvolle Beispiele zu entwickeln. Wichtig ist, dass Sie die möglichen Zusammenhänge zwischen einer Abfrage und einem Speichern-Befehl erkennen.

Man kann sich in der Unterabfrage auch auf Spalten beziehen, die aus der Tabelle stammen, die geändert werden soll. Beispiel (nur teilweiser Bezug auf die Beispieldatenbank):

Quelltext

Falsch

```
update Abteilung
  set Ort = ( SELECT Ort
              from Adressbuch
              where Abteilung.PLZ = Adressbuch.PLZ )
;
```

Das Problem dieser Abfrage ist, dass sie nur dann funktioniert, wenn es im Adressbuch nur exakt einen einzigen Eintrag zu einer bestimmten PLZ gibt. Sobald man zu einer PLZ mehrere Adressen notiert hat, findet die Unterabfrage mehrere Sätze; das ist bei dieser Unterabfrage nicht zulässig. Damit der Update auch in solchen Fällen funktioniert, muss ein DISTINCT eingefügt werden oder man verwendet die MAX-oder die MIN-Funktion:

```
update Abteilung
  set Ort = ( SELECT MAX(Ort)
              from Adressbuch
              where Abteilung.PLZ = Adressbuch.PLZ )
;
```

Bei solchen Unterabfragen mit einem Bezug zu dem Satz, der verändert werden soll, kann es vorkommen, dass die Ausführung dieser Anweisung ziemlich lange dauert. Das liegt daran, dass alle Sätze aus der Tabelle Abteilung verändert werden sollen. Für jeden Satz muss die Unterabfrage erneut ausgeführt werden. Wenn eine einzelne Ausführung dieser Unterabfrage eine Sekunde dauert, und wir haben z. B. 1000 Sätze in der Tabelle Abteilung, dann dauert die Ausführung des gesamten Statements 1000 Sekunden, also ca. 16 Minuten.

Man kann auch mehrere Spalten aus Unterabfragen befüllen. Beispiel:

```
update Abteilung
  set Ort = ( SELECT MAX(Ort)
```

```
        from Telefonbuch
        where PLZ = '12345' ),
Leiter = ( SELECT Manager
        from Mitarbeiter
        where Kuerzel = 'A073' )
where Kuerzel = 'Ausb';
;
```

Wenn man mehrere Werte aus derselben Unterabfrage übernehmen will, dann könnte man dieselbe Unterabfrage mehrfach angeben. Aber oft kann das Datenbanksystem nicht erkennen, dass es sich immer wieder um dieselbe Unterabfrage handelt, und müsste sie mehrfach ausführen. Einfacher, übersichtlicher und dann auch schneller ist die folgende Variante, die aber nicht jedes DBMS kennt.

► **Aufgabe:** In der Tabelle *Abteilung* werden die Spalten *Ort*, *Leiter*, *Telefon* geändert.

```
update Abteilung
  set ( Ort, Leiter, Telefon )
    = ( SELECT Ort, Name, Telefon
        from Adressbuch
        where Adressbuch.Personalnummer = Abteilung.Chef_Personalnummer
    )
```

Hier werden alle Sätze in der Tabelle *Abteilung* aktualisiert unter Verwendung der Tabelle *Adressbuch*; die Personalnummer des Abteilungsleiters kann wie oben bestimmt werden.

25.4.4. Verwendung bei DELETE

Schauen wir uns noch die grundsätzliche Struktur eines DELETE-Befehls an:

```
DELETE FROM <tabellenname>
[ WHERE <bedingungsliste> ];
```

Daraus ergibt sich, dass Abfragen nur für Vergleichswerte der Bedingungsliste sinnvoll sind (ebenso wie in verschiedenen früheren Beispielen).

► **Aufgabe:** Die Abteilung „Forschung und Entwicklung“ wird ausgelagert; alle zugeordneten Mitarbeiter werden in der Datenbank gelöscht.

- Lösung: Suche in einer Abfrage die benötigte ID dieser Abteilung.

```
delete from Mitarbeiter
where Abteilung_ID in ( SELECT ID
                        from Abteilung
                        where Bezeichnung = 'Forschung und Entwicklung'
                      );
```

25.5. Zusammenfassung

In diesem Kapitel benutzten wir Unterabfragen:

- Sowohl einzelne Werte als auch Listen als Ergebnis einer Abfrage können als Vergleichswerte in der WHERE-Klausel verwendet werden.
- Eine Tabelle als Ergebnis von Abfragen kann wie jede „echte“ Tabelle als Teil der FROM- oder JOIN-Klausel verwendet werden.

Ähnlich können Ergebnisse von Abfragen beim Speichern genutzt werden:

- Ganze Datensätze werden mit INSERT in eine Tabelle eingefügt werden.
- Einzelne Werte werden in der WHERE-Klausel oder in der SET-Anweisung genutzt.

25.6. Übungen

Übung 1 – Definitionen

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.

Übung 2 – Ein einzelner Wert

Welche Verträge (mit einigen Angaben) hat der Mitarbeiter „Braun, Christian“ abgeschlossen? *Ignorieren Sie die Möglichkeit, dass es mehrere Mitarbeiter dieses Namens geben könnte.*

Übung 3 – Ein einzelner Wert

Zeigen Sie alle Verträge, die zum Kunden „Heckel Obsthandel GmbH“ gehören. *Ignorieren Sie die Möglichkeit, dass der Kunde mehrfach gespeichert sein könnte.*

Übung 4 – Eine Liste von Werten

Ändern Sie die Lösung von Übung 3, sodass auch mehrere Kunden mit diesem Namen als Ergebnis denkbar sind.

Übung 5 – Eine Liste von Werten

Zeigen Sie alle Fahrzeuge, die im Jahr 2008 an einem Schadensfall beteiligt waren.

Übung 6 – Eine Liste von Werten

Zeigen Sie alle Fahrzeugtypen (mit ID, Bezeichnung und Name des Herstellers), die im Jahr 2008 an einem Schadensfall beteiligt waren.

Übung 7 – Eine Tabelle als Ergebnis

Bestimmen Sie alle Fahrzeuge eines bestimmten Herstellers mit Angabe des Typs.

Hinweis: Es handelt sich um das letzte Beispiel aus dem Abschnitt „Ergebnis in Form einer Tabelle“. Benutzen Sie jetzt JOIN.

Übung 8 – Eine Tabelle als Ergebnis

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.

Übung 9 – Speichern mit Unterabfrage

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-PROGRAMMIERUNG⁴ möglich.) Bei dieser Änderungsdatei handelt es sich nur um fiktive Daten, keine echten Änderungen.

25.7. Lösungen

Lösung zu Übung 1 – Definitionen

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

Lösung zu Übung 2 – Ein einzelner Wert

4 Kapitel 29 auf Seite 363


```
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 – Ein einzelner Wert

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

Lösung zu Übung 4 – Eine Liste von Werten

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

Lösung zu Übung 5 – Eine Liste von Werten

```
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 – Eine Liste von Werten

```
SELECT distinct ft.ID as TypID, ft.Bezeichnung as Typ, fh.Name as
Hersteller
```

```
FROM Fahrzeugtyp ft
  inner join Fahrzeughersteller 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 – Eine Tabelle als Ergebnis

```
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 Fahrzeughersteller
            WHERE Name = 'Volkswagen' )
        ) Typen on fz.Fahrzeugtyp_ID = Typen.ID;
```

Lösung zu Übung 8 – Eine Tabelle als Ergebnis

```
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.Abtteilung_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 – Speichern mit Unterabfrage

```
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 );
```


26. Erstellen von 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.

26.1. Allgemeine Hinweise

Bei einer View wird die Abfrage in der Datenbank gespeichert, aber nicht das Ergebnis. Bei jedem neuen Aufruf der View wird die dahinterliegende Abfrage neu ausgeführt, denn sie soll ja das Ergebnis anhand der aktuellen Daten bestimmen.

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.

Views dienen dazu, die Komplexität der Abfragen für den Anwender zu reduzieren. Die View wird mit ihrer (komplexen) Abfrage angelegt, und die Nutzer können die Daten dieser View abfragen.

Weiterhin können Views genutzt werden, um den Zugriff auf bestimmte Daten einzuschränken. Nutzer können Zugriff nur auf bestimmte Views bekommen.

Somit lässt sich der Zugriff für einzelne Nutzer auf bestimmte Daten (Spalten und Datensätze) beschränken.

26.2. Eine View anlegen und benutzen

Views werden mit dem Befehl **CREATE VIEW** mit folgender Syntax angelegt.

```
CREATE VIEW <View-Name>
  [ ( <Spaltennamen> ) ]
  AS <Select-Ausdruck> ;
```

Zu dieser Definition gehören folgende Bestandteile:

- **CREATE VIEW** kennzeichnet den Befehl.
- Unter **<View-Name>** ist eine Bezeichnung anzugeben, unter der die View in einem **SELECT**-Befehl angesprochen wird. Dieser Name muss eindeutig sein und darf auch kein Name einer „echten“ Tabelle sein.
- Als **<Select-Ausdruck>** wird ein (beliebiger) **SELECT**-Befehl eingetragen.
- Es wird empfohlen, möglichst bei allen Spalten mit einem Alias zu arbeiten.
- Diese können wahlweise vor dem **AS** in Klammern angegeben werden oder (wie üblich) Teil des **<Select-Ausdruck>**s sein.

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. */
```

26.2.1. Eine einfache View

Im einfachsten Fall greifen wir auf eine einfache Verknüpfung zweier Tabellen zu und verbinden dies mit einer festen Suchbedingung. ► **Aufgabe:** 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 Fahrzeughersteller fh on ft.Hersteller_ID = fh.ID
WHERE fh.Land = 'Deutschland';
```

Die Abfrage basiert auf den beiden Tabellen *Fahrzeugtyp* und *Fahrzeughersteller*. 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 kann dann wie eine „normale“ Tabellen in Abfragen genutzt werden.

```
SELECT * FROM Deutscher_Fahrzeugtyp order by Hersteller;
```

FAHRZEUGTYP	HERSTELLER
A3	Audi
A4	Audi
325	BMW
525	BMW
Z3	BMW
Fiesta	Ford
Focus	Ford /* usw. */

In diesem Fall kann die ORDER BY-Klausel ebensogut Teil der View sein; das untersuchen wir später noch.

26.2.2. Eine View mit variabler Selektion

Es klappt leider nicht, in eine View eine WHERE-Klausel einen (variablen) Parameter einzubauen, der erst im SELECT mit einem konkreten Wert versehen wird. ► **Aufgabe:** Gesucht wird eine Abfrage über die Mitarbeiter einer Abteilung; am Anfang soll der Abteilungsleiter stehen, danach alphabetisch die betreffenden Mitarbeiter. Die Nummer der Abteilung soll nicht fest vorgegeben werden, sondern variabel sein.

Quelltext

Falsch

```
create view Mitarbeiter_in_Abteilung
as select Personalnummer, Name, Vorname, Geburtsdatum
from Mitarbeiter
WHERE Abteilung_ID = ?
```

```
order by Ist_Leiter, Name, Vorname;
```

```
Commit nicht möglich  
Invalid token.
```

Auch Alternativen für das Fragezeichen führen nicht zum Ziel. Es bleibt nur ein kleiner Umweg, nämlich die *Abteilung_ID* in der View zu berücksichtigen und später für WHERE zu nutzen:

```
create view Mitarbeiter_in_Abteilung  
  ( Pers, Name, Vorname, Geburtsdatum, Abt )  
as select Personalnummer, Name, Vorname, Geburtsdatum, Abteilung_ID  
   from Mitarbeiter  
   order by Ist_Leiter, Name, Vorname;
```

Damit können alle Angaben einer bestimmten Abteilung geholt werden; die Spalte *Abt* bleibt zur Verdeutlichung stehen:

```
select * from Mitarbeiter_in_Abteilung  
where Abt = 5;
```

PERS	NAME	VORNAME	GEBURTSDATUM	ABT
50001	Pohl	Helmut	27.10.1980	5
50002	Braun	Christian	05.09.1966	5
50004	Kalman	Aydin	17.12.1976	5
50003	Polovic	Frantisek	26.11.1961	5

Und siehe da: zuerst kommt der Abteilungsleiter, danach die anderen Mitarbeiter in alphabetischer Reihenfolge.

Hinweis: Eine Alternative zu einer VIEW mit variabler WHERE-Bedingung ist eine „StoredProcedure“, die diese Abfrage enthält und einen Wert als Parameter entgegennimmt; sie wird in einem späteren Kapitel behandelt.

26.2.3. Probleme mit der Sortierung

Ändern wir die obige View deutscher Fahrzeuge dahin, dass die Sortierung nach Hersteller fest eingebaut wird.

- Bitte beachten Sie: Wenn Sie oben die View *Deutscher_Fahrzeugtyp* fest gespeichert haben, müssen Sie in diesem Abschnitt einen anderen Namen ver-

wenden oder stattdessen etwas wie CREATE OR ALTER (siehe die DBMS-Dokumentation) benutzen.

```
CREATE VIEW Deutscher_Fahrzeugtyp ( Typ, Firma )
AS SELECT DISTINCT ft.Bezeichnung, fh.Name as Firma
   FROM Fahrzeugtyp ft
      join Fahrzeughersteller fh on ft.Hersteller_ID = fh.ID
 WHERE fh.Land = 'Deutschland'
 order by Firma;
```

Bitte beachten Sie, dass in diesem Fall der Spalten-Alias *Firma* auch Teil des SELECT-Befehls sein muss, damit er in der ORDER BY-Klausel bekannt ist.

Jetzt kann diese Liste je nach Wunsch mit oder ohne Sortierung abgerufen werden:

```
SELECT * FROM Deutscher_Fahrzeugtyp;           /* automatisch
sortiert nach Firma */
SELECT * FROM Deutscher_Fahrzeugtyp ORDER BY Typ; /* speziell
sortiert nach Typ */
```

26.2.4. Views in Verbindung mit JOIN

Die obige Verknüpfung „Fahrzeugtyp plus Hersteller“ benötigen wir in der Praxis ständig, nicht nur in der konkreten Abfrage nach deutschen Herstellern. Bisher – zum Beispiel mit OUTER JOIN – haben wir beide Tabellen separat per JOIN eingebunden, mussten aber immer auf die Art des JOINS aufpassen. Das kann man einmalig durch eine fiktive Tabelle *Fahrzeugart*, also eine VIEW mit den benötigten Informationen steuern.

Eine solche VIEW erfüllt mehrere Wünsche:

- Die eigentlichen Informationen werden getrennt gespeichert; es ist nicht nötig, bei jedem Fahrzeugtyp den Hersteller und sein Herkunftsland aufzuführen. *Wie wir aus der Wirtschaftspolitik des Jahres 2009 wissen, kann sich ein Herkunftsland durchaus ändern; nach den Regeln der Normalisierung ist die separate Tabelle der Hersteller nicht nur sinnvoll, sondern notwendig.*
- Bei jeder Abfrage des Fahrzeugtyps erhalten wir sofort auch den Hersteller.
- Jede solche Abfrage wird einfacher, weil eine Tabelle weniger benötigt wird.
- Das DBMS kennt seine VIEWS und hat sie „von Haus aus“ optimiert; also wird jede solche Abfrage auch schneller ausgeführt.

Diese Aussage gilt nicht unbedingt bei jeder Abfrage und jedem DBMS. Aber nach allen Erkenntnissen über interne Datenbankstrukturen kann man davon ausgehen.

Das obige „einfache Beispiel“ der VIEW müssen wir also nur ein wenig umschreiben: ► **Aufgabe:** Bereite eine (fiktive) Tabelle *Fahrzeugart* vor mit allen relevanten Informationen aus den Tabellen *Fahrzeugtyp* und *Fahrzeughersteller*.

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

Für den Anwender sieht es tatsächlich so aus, als hätten wir eine einfache Tabelle mit allen Angaben:

```
select * from Fahrzeugart
order by Land, Hersteller, Bezeichnung;
```

ID	BEZEICHNUNG	HERSTELLER	LAND
19	C30	Volvo	
18	S40	Volvo	
12	A3	Audi	Deutschland
13	A4	Audi	Deutschland
9	325	BMW	Deutschland
10	525	BMW	Deutschland
11	Z3	BMW	Deutschland

Damit kann das letzte der Beispiele zu OUTER JOIN vereinfacht werden.

► **Aufgabe:** 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;
```

MITNR	NAME	VORNAME	DIW	KENNZEICHEN	TYPID	TYP	HERSTELLER
80001	Schindler	Christina	8	DO-WB 428	14	A160	
	Mercedes-Benz						
90001	Janssen	Bernhard	9	DO-WB 429	14	A160	
	Mercedes-Benz						
100001	Grosser	Horst	10	DO-WB 4210	14	A160	
	Mercedes-Benz						
110001	Eggert	Louis	11	DO-WB 4211	14	A160	
	Mercedes-Benz						
120001	Carlsen	Zacharias	12	DO-WB 4212	14	A160	
	Mercedes-Benz						
			13	DO-WB 111	16	W211	
(E-Klasse)	Mercedes-Benz						
50002	Braun	Christian	14	DO-WB 352	2	Golf	
	Volkswagen						
50003	Polovic	Frantisek	15	DO-WB 353	3	Passat	
	Volkswagen						
50004	Kalman	Aydin	16	DO-WB 354	4	Kadett	
	Opel						
80002	Aliman	Zafer	17	DO-WB 382	2	Golf	
	Volkswagen						
80003	Langer	Norbert	18	DO-WB 383	3	Passat	
	Volkswagen						
80004	Kolic	Ivana	19	DO-WB 384	4	Kadett	
	Opel						

Einige kleine Änderungen machen alles einfacher: Die Tabelle *Fahrzeugtyp* wird durch die View *Fahrzeugart* ersetzt; die JOIN-Verknüpfung auf *Fahrzeughersteller* kann ersatzlos entfallen. Lediglich zur Verdeutlichung werden Tabellen-Alias und Spaltennamen geändert.

26.3. Eine View ändern oder löschen

26.3.1. Änderungen

Eine VIEW zu ändern, wird unterschiedlich gehandhabt.

- Üblich ist das „normale“ **ALTER VIEW**.
- Firebird behandelt eine Änderung mit **RECREATE** als Löschung und anschließende Neuaufnahme.

26.3.2. Löschungen

Views werden mit dem Befehl **DROP VIEW** gelöscht.

```
DROP VIEW Deutscher_Fahrzeugtyp;
```

Hierbei wird nur die View als Objekt in der Datenbank gelöscht. Die Tabellen und Daten in den Tabellen, auf denen die View basiert, werden davon nicht beeinflusst – sie werden nicht gelöscht.

26.4. Zusammenfassung

- Views sind Abfragen, die in der Datenbank als Objekt gespeichert werden.
- Views können die Komplexität für den Anwender reduzieren.
- Views können für eine detaillierte Zugriffskontrolle genutzt werden.
- Views werden mit CREATE VIEW erstellt; sie werden in Abfragen wie jede Tabelle benutzt.
- Views werden mit DROP VIEW gelöscht.

26.5. Übungen

Die Formulierung „eine View kontrollieren“ meint: Mit einer geeigneten Abfrage soll überprüft werden, ob die View richtig erstellt worden ist.

Übung 1 – Definitionen

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.

Übung 2 – Eine View benutzen

Skizzieren Sie eine Abfrage, durch die eine beliebige View benutzt werden kann.

Übung 3 – Eine einfache View erstellen

Bei der Suche nach Dienstwagen sollen mit der View *Dienstwagen_Anzeige* immer auch angezeigt werden:

- Name und Vorname des Mitarbeiters
- ID und Bezeichnung seiner Abteilung
- 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 );
```

Übung 4 – Mehrere Tabellen und Views verbinden

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.

Dies ist ein Beispiel dafür, dass eine View bei Abfragen genauso wie eine „echte“ Tabelle benutzt werden kann.

Übung 5 – Eine VIEW auf mehrere Tabellen

Erstellen Sie eine Sicht *Vertrag_Anzeige*, bei der zu jedem Vertrag angezeigt werden:

- ID, Vertragsnummer, Abschlussdatum, Art (als Text)
- Name, Vorname des Mitarbeiters
- Name, Vorname des Versicherungsnehmers

- Kennzeichen des Fahrzeugs

Übung 6 – Eine VIEW auf mehrere Tabellen

Erweitern Sie die vorstehende View so, dass mit Hilfe der View *Fahrzeugart* auch Bezeichnung, Hersteller und Land angezeigt werden.

Übung 7 – Eine View abschnittsweise kontrollieren

Erstellen Sie eine Abfrage, sodass für einen Teil der Verträge die vorstehende View kontrolliert wird.

Übung 8 – Eine weitere VIEW auf mehrere Tabellen

Erstellen Sie eine Sicht *Schaden_Anzeige*, bei der zu jedem an einem Schadensfall beteiligten Fahrzeug angezeigt werden:

- ID, Datum, Gesamthöhe eines Schadensfalls
- Kennzeichen und Typ des beteiligten Fahrzeugs
- Anteiliger Schaden
- ID des Versicherungsvertrags

Übung 9 – Eine weitere VIEW auf mehrere Tabellen

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.

Übung 10 – Eine View zur Auswertung einer View

Erstellen Sie eine weitere View so, dass die vorstehende View für alle Schadensfälle des aktuellen Jahres benutzt wird.

26.6. Lösungen

Lösung zu Übung 1 – Definitionen

Richtig sind die Aussagen 1, 3, 4, 6, 8. Falsch sind die Aussagen 2, 5, 7, 9.

Lösung zu Übung 2 – Eine View benutzen

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

Lösung zu Übung 3 – Eine View erstellen

```
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.Abcteilung_ID,
         ab.Bezeichnung
from Dienstwagen dw
  left join Mitarbeiter mi
         on mi.ID = dw.Mitarbeiter_ID
  left join Abteilung ab
         on ab.ID = mi.Abcteilung_ID;
```

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

Lösung zu Übung 4 – Mehrere Tabellen und Views verbinden

```
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.Abcteilung_ID,
         ab.Bezeichnung
from Dienstwagen dw
  left join Mitarbeiter mi
         on mi.ID = dw.Mitarbeiter_ID
  left join Abteilung ab
         on ab.ID = mi.Abcteilung_ID
  inner join Fahrzeugart fa
         on fa.ID = dw.Fahrzeugtyp_ID;
```

Lösung zu Übung 5 – Eine VIEW auf mehrere Tabellen

```
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;
```

Hinweis: Weil die Zusatzangaben Pflicht sind, können wir einheitlich mit INNER JOIN arbeiten.

Lösung zu Übung 6 – Eine VIEW auf mehrere Tabellen

```
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;
```


Lösung zu Übung 7 – Eine View abschnittsweise kontrollieren

```
SELECT * FROM Vertrag_Anzeige
WHERE EXTRACT(YEAR from Abschlussdatum) <= 1990;
```

Lösung zu Übung 8 – Eine weitere VIEW auf mehrere Tabellen

```
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;
```

Lösung zu Übung 9 – Eine weitere VIEW auf mehrere Tabellen

```
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;
```

Lösung zu Übung 10 – Eine View zur Auswertung einer View

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

26.7. Siehe auch

Ergänzende Informationen gibt es in den folgenden Kapiteln:

- [STOREDPROCEDURE](#)¹
- [VERKNÜPFUNG MEHRERER TABELLEN BEI OUTER JOIN](#)²

¹ [Kapitel 31 auf Seite 399](#)

² [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%C3%9C
HRUNG%20IN%20SQL%3A%
20OUTER%20JOIN%23VERKN%
FCPFUNG%20MEHRERER%20TABELLEN](http://de.wikibooks.org/wiki/Einf%C3%9Chrung%20in%20SQL%3A%20Outer%20Join%23Verkn%C3%9Cpfung%20mehrerer%20Tabellen)

Teil IV.

Erweiterungen

27. DDL – Einzelheiten

In diesem Kapitel werden einige Befehle der Data Definition Language (DDL) vertieft behandelt.

Wegen des Umfangs mancher Befehle und Optionen werden die Abschnitte sachlich gegliedert, nicht nach einem einzelnen Befehl.

27.1. Definition einer Tabelle

Um eine **Tabelle** zu erzeugen, sind sehr umfangreiche Angaben nötig.

```
CREATE TABLE <Tabellenname>
(
  <Spaltenliste>
  [ , <Einschränkungen> ]
);
```

Zum Erstellen einer Tabelle gehören folgende Angaben:

- der **Name** der Tabelle, mit dem die Daten über die DML-Befehle gespeichert und abgerufen werden

Dazu kommen – in Klammern gesetzt – die weiteren Angaben:

- die Liste der **Spalten** (Felder), und zwar vor allem mit den jeweiligen Datentypen.
- Angaben wie der **Primärschlüssel** (PRIMARY KEY, PK) oder weitere Indizes

Jede Spalte und Einschränkung wird mit einem Komma abgeschlossen; dieses entfällt vor der schließenden Klammer. Die Einschränkungen – CONSTRAINTs – werden häufig nicht sofort festgelegt, sondern durch anschließende ALTER TABLE-Befehle; sie werden deshalb getrennt besprochen.

Notwendig sind: der Name des Befehls, der Name der Tabelle, die runden Klammern, mindestens eine Spalte mit Name und Typ. Eine solche „Minimalversion“ gibt es aber nur für Code-Beispiele; in der Praxis gehören immer mehrere Spalten und der PK dazu.

Beim **Entwurf einer Datenbank** und ihrer Tabellen sollten Sie immer beachten:

- Datentypen, Art und Umfang der Zusatzangaben hängen vom DBMS ab.
- Primärschlüssel: Manche DBMS verlangen ausdrücklich einen PK – meistens eine ID o. ä., nur selten aus mehreren Spalten zusammengesetzt. Auch wenn es nicht verlangt wird, ist ein PK dringend zu empfehlen; eine Tabelle ohne PK ist selten sinnvoll. Dessen Inhalte müssen eindeutig sein und dürfen sich nicht wiederholen.
- Dafür wird meistens eine automatische Fortzählung (AUTO_INCREMENT) angeboten, was ohne Entwicklungsaufwand die Bedingungen eines Primärschlüssels erfüllt.



Sie müssen bei Art und Umfang aller Angaben immer die Besonderheiten Ihres DBMS beachten!

Die Code-Auszüge stammen überwiegend aus den Skripten zur Beispieldatenbank. Die wichtigsten Bestandteile ersehen Sie aus dem folgenden Beispiel; weitere Bestandteile werden in den späteren Abschnitten behandelt.

MySQL Quelltext

```
create table Abteilung
( ID          INTEGER      not null auto_increment primary key,
  Kuerzel     VARCHAR(10)  not null,
  Bezeichnung VARCHAR(30)  not null,
  Ort        VARCHAR(30)
);
```

Die Tabelle *Abteilung* wird mit vier Spalten erzeugt:

- *ID* ist eine ganze Zahl, die nicht NULL sein darf, die Werte durch die automatische Zählung erhält und die als PRIMARY KEY benutzt wird.
- *Kuerzel* ist eine Zeichenkette mit variabler Länge (höchstens 10 Zeichen), die nicht NULL sein darf.
- *Bezeichnung* und *Ort* sind Zeichenketten mit höchstens 30 Zeichen. Der *Ort* darf NULL sein, die *Bezeichnung* nicht.

27.2. Definition einer einzelnen Spalte

Jede einzelne **Spalte** wird wie folgt definiert.

```
<Spaltenname> <Datentyp> [ <Optionen> ]
```

Jede der Optionen wird mit einem Schlüsselwort und der erforderlichen Angabe hinzugefügt.

Die Optionen können in der Regel kombiniert werden; die Reihenfolge muss beachtet werden. Verschiedene der Einschränkungen (siehe unten) können auch bei einer einzelnen Spalte angegeben werden.

27.2.1. COLLATE – Sortierungsregel

Jede Spalte kann eine eigene Regel für die alphabetische Sortierung erhalten – abweichend von der Regel der Tabelle oder Datenbank bzw. von der Standard-sortierung gemäß Zeichensatz. ► **Aufgabe:** In der obigen Definition der Tabelle *Abteilung* soll die Bezeichnung nicht nach den allgemein für die Datenbank gültigen Regeln sortiert werden, sondern nach denen für kanadisches Französisch:

```
Firebird Quelltext
```

```
create /* usw. bis */
      Bezeichnung VARCHAR(30) not null COLLATION FR_CA,
/* usw. */
```

Achtung: So funktioniert der Befehl nicht. Der obige CREATE-Befehl stammt aus der MySQL-Version, diese COLLATION-Ergänzung aber aus Firebird.

27.2.2. NULL-Werte zulässig oder nicht

NULL bzw. NOT NULL legt ausdrücklich fest, ob NULL-Werte in der Spalte zulässig sind oder nicht. Der Standardwert ist „zulässig“, das NULL kann deshalb entfallen.

Im obigen CREATE-Befehl gilt:

- Die Spalten *ID*, *Kuerzel*, *Bezeichnung* dürfen keine NULL-Werte enthalten. Die Informationen in diesen Spalten sind wesentlich für einen Datensatz; eine Speicherung ohne einen dieser Werte wäre sinnlos.
- Die Spalte *Ort* darf dagegen NULL-Werte enthalten. Diese Angabe ist nur eine zusätzliche Information; die Abteilung steht auch dann eindeutig fest, wenn der Sitz nicht bekannt oder noch nicht festgelegt ist.

27.2.3. DEFAULT – Vorgabewert

Mit DEFAULT <Wert> wird ein Standardwert festgelegt (als konstanter Wert oder als Ergebnis einer Funktion); dieser wird immer dann verwendet, wenn bei einer Neuaufnahme für diese Spalte kein Wert angegeben ist.

► **Aufgabe:** In der Tabelle *Mitarbeiter* erhält die Spalte *Ist_Leiter* den Vorgabewert 'N'; denn ein Mitarbeiter ist normalerweise kein Abteilungsleiter:

```
create table Mitarbeiter
( ID          INTEGER      not null auto_increment primary key,
  /* usw. bis */
  Ist_Leiter  CHAR(1)     default 'N',
  Abteilung_ID INTEGER     not null
);
```

Weit verbreitet sind dabei Standardwerte, mit denen Datum/Zeit einer Änderung und die Bearbeiterin registriert werden:

- CURRENT_TIMESTAMP als aktuelles Datum und Uhrzeit
- CURRENT_USER

27.2.4. AUTO_INCREMENT – automatischer Zähler

AUTO_INCREMENT legt fest, dass die Werte in dieser Spalte automatisch vom DBMS hochgezählt werden – siehe das obige Beispiel unter „Definition einer Tabelle“.

Die folgenden SQL-Systeme bieten einen automatischen Zähler als Teil der Spaltendefinition an. Teilweise hängt es von der Version ab, was möglich ist und was nicht.

DBMS	Schlüsselwort	Beispiel	Bemerkungen
------	---------------	----------	-------------

MS-SQL IDENTITY

```
ALTER TABLE dbo.doc_
exe
ADD column_
b INT IDENTITY CONSTRAINT column_
b_pk PRIMARY KEY,
```

MySQL AUTO_-
INCREMENT

```
CREATE TABLE Abteilung
( ID INTEGER NOT NULL AUTO_-
INCREMENT PRIMARYKEY, /* usw. */
);
```

Oracle AUTO INCREMENT

```
CREATE TABLE Fahrzeug
(ID INTEGER NOT NULL AUTOINCREMENT, /* usw. */
CONSTRAINT Fahrzeug_-
PK PRIMARY KEY (ID)
);
```

nur möglich bei
der Lite-Version für
mobile Computer
"AUTO INCREMENT"
wird hier mit Leerze-
ichen geschrieben

Bei den folgenden SQL-Systemen werden andere Verfahren angeboten.

DBMS	Schlüsselwörter	Beispiel	Bemerkungen
Firebird	SEQUENCE TRIGGER	<pre> /* Definition */ CREATE SEQUENCE Fahrzeug_ ID; /* Benutzung im Trigger */ CREATE OR ALTER TRIGGER Fahrzeug_ BEFORE INSERT FOR Fahrzeug ACTIVE BEFORE INSERT POSITION 0 AS begin IF ((new.ID IS NULL) OR (new.ID = 0)) then new.ID = next value FOR Fahrzeug_ ID; end </pre>	<p>Es wird ein Zähler definiert (wahlweise je Tabelle oder pauschal). In einem Before-Insert-Trigger wird der nächste Wert abgerufen und eingefügt.</p>
Firebird	GENERATOR <i>veraltet</i> TRIGGER	<pre> /* Definition */ CREATE GENERATOR Fahrzeug_ ID; SET GENERATOR Fahrzeug_ ID TO 0; /* Benutzung im Trigger */ CREATE OR ALTER TRIGGER Fahrzeug_ BEFORE INSERT FOR Fahrzeug ACTIVE BEFORE INSERT POSITION 0 AS begin IF ((new.ID IS NULL) OR (new.ID = 0)) then new.ID = GEN_ ID(Fahrzeug_ID, 1); end </pre>	<p>Es wird ein Zähler definiert (wahlweise je Tabelle oder pauschal). In einem Before-Insert-Trigger wird der nächste Wert abgerufen und eingefügt.</p> <p>Das Verfahren mit GENERATOR sollte für neue Datenbanken nicht mehr benutzt werden.</p>

Oracle SEQUENCE
TRIGGER

```

/* Definition */
CREATE SEQUENCE Fahrzeug_
ID;
/* Benutzung im Trigger */
CREATE OR REPLACE TRIGGER BeforeInsert
BI
BEFORE INSERT ON Fahrzeug
REFERENCING NEW AS NEW OLD AS OLD
FOR EACH ROW
WHEN (:NEW.ID IS NULL)
begin
  /* in früheren Oracle-Versionen: */
  SELECT Fahrzeug_
ID.NEXTVAL INTO :NEW.ID FROM DUAL;
  /* ab Version 11g direkte Zuweisung: */
  :NEW.ID := Fahrzeug_
ID.NEXTVAL;
end

```

Es wird ein Zähler definiert (wahlweise je Tabelle oder pauschal). In einem Before-Insert-Trigger wird der nächste Wert abgerufen und eingefügt.

Oracle SEQUENCE
INSERT

```

/* Definition */
CREATE SEQUENCE Fahrzeug_
ID;
/* Benutzung im INSERT-Befehl */
INSERT INTO Fahrzeug
( ID, Kennzeichen /* usw. */ )
VALUES
( Fahrzeug_
ID.NEXTVAL, 'BO-CR 123' /* usw. */ )
;

```

Es wird ein Zähler definiert (wahlweise je Tabelle oder pauschal). Im INSERT-Befehl wird der nächste Wert abgerufen und übernommen. Die Version mit Trigger ist vorzuziehen.

Wie schon früher erwähnt, gehört der automatische Zähler gemäß AUTO_INCREMENT genau zu der betreffenden Tabelle; eine „Sequenz“ bezieht sich dagegen auf die gesamte Datenbank. Es ist ohne weiteres möglich, eine einzige Sequenz AllMyIDs zu definieren und die neue ID einer jeden Tabelle daraus abzuleiten. In der Beispieldatenbank benutzen wir getrennte Sequenzen, weil sie für die verschiedenen DBMS „ähnlich“ aussehen soll.

Für die meisten Varianten gibt es Parameter zur genaueren Steuerung, nachzulesen in der DBMS-Dokumentation.

27.2.5. COMMENT – Beschreibung verwenden

Damit wird der Inhalt der Spalte beschrieben. Dies ist nützlich für alle Spalten, sofern ein Bezeichner nicht ganz eindeutig ist oder der Inhalt besondere Bedin-

gungen erfüllen soll. Damit erleichtern Sie sich selbst und anderen „Betreuern“ der Datenbank die Arbeit.

```
create table Schadensfall
( ID      INTEGER      not null auto_increment primary key,
  Datum  DATE          not null,
  Ort    VARCHAR(200) not null
        COMMENT 'nicht nur mit dem Ortsnamen, sondern auch mit
  Straße und äußeren Umständen'
  , /* usw. */
  );
```

27.3. Tabelle ändern

Mit **ALTER TABLE** wird die Struktur einer Tabelle geändert:

```
ALTER TABLE <Tabellenname> <Aufgabenliste> ;
```

Es können also mehrere Aufgaben mit einem ALTER-Befehl ausgeführt werden. Die möglichen Aufgaben sind in den einzelnen Abschnitten beschrieben.

Der Begriff COLUMN ist nicht immer Teil des Befehls: Bei manchen DBMS *kann* er weggelassen werden, bei manchen *darf* er nicht benutzt werden.

27.3.1. Stop – Aufgabe kann nicht ausgeführt werden

Eine Spalte kann oft nicht geändert oder gelöscht werden, wenn sie an anderer Stelle benutzt wird. Das gilt vor allem dann, wenn diese Spalte beim PRIMARY KEY, einem INDEX, einem FOREIGN KEY oder in einer CHECK-Einschränkung für die Tabelle benutzt wird. In diesen Fällen muss zunächst die „abhängige“ Konstruktion gelöscht, deaktiviert oder geändert werden. Erst danach kann die Änderung in der Tabelle ausgeführt werden.

Der Datentyp kann durch ALTER COLUMN nur dann geändert werden, wenn die „alten“ Werte automatisch (implizit) in den neuen Datentyp konvertiert werden können.

27.3.2. ADD COLUMN – Spalte hinzufügen

Diese Aufgabe fügt der Tabelle eine weitere Spalte hinzu.

► **Aufgabe:** Die Tabelle *Versicherungsvertrag* wird um Spalten zur Berechnung und Anpassung der Versicherungsprämie erweitert.

Firebird-Version hinsichtlich Schreibweise (ohne "COLUMN") und Reihenfolge der Optionen

Firebird Quelltext

```
alter table Versicherungsvertrag
  add Basispraemie          DECIMAL DEFAULT 500 not null,
  add Praemiensatz          INTEGER DEFAULT 100 not null,
  add Praemienaenderung     DATE;
```

Die bisherigen Inhalte der Tabelle bleiben unverändert. In den neuen Spalten wird der DEFAULT-Wert eingetragen, andernfalls NULL.

27.3.3. ALTER COLUMN – Spalte ändern

Diese Aufgabe ändert eine Spalte dieser Tabelle. Dies kann eine Änderung des Datentyps, ein anderer DEFAULT-Wert oder eine andere Einschränkung sein.

► **Aufgabe:** In der Tabelle *Abteilung* ist die Spalte *Kuerzel* mit VARCHAR(10) definiert, die einzelnen Werte sind aber immer genau 4 Zeichen lang. Die Spaltendefinition soll an die Realität angepasst werden.

Firebird Quelltext

Falsch

```
alter table Abteilung
  ALTER COLUMN Kuerzel TYPE CHAR(4);
```

```
Firebird-Fehlermeldung: This operation is not defined for system
tables.
unsuccessful metadata update.
New size specified for column KUERZEL must be at least 40
characters.
```

Maßnahme kann nicht unbedingt ausgeführt werden.

Mit der Option TYPE wird der Datentyp geändert. Vorhandene Inhalte werden dabei möglichst implizit konvertiert. Beispielsweise MySQL hat keine Probleme

damit, den Text entsprechend abzuschneiden. Firebird weigert sich aber, obwohl die tatsächlichen Inhalte passen, sondern bringt eine völlig verwirrende Fehlermeldung, die gar nicht zu dieser Situation passt.

In solchen Fällen gibt es einen **Umweg**:

- Erzeugen Sie eine neue, temporäre Spalte.
- Kopieren Sie alle Inhalte durch einen UPDATE-Befehl aus der „alten“ Spalte, die geändert werden soll, in die temporäre Spalte.
- Löschen Sie die „alte“ Spalte.
- Erzeugen Sie eine neue Spalte unter dem „alten“ Namen mit den „neuen“ Eigenschaften.
- Kopieren Sie alle Inhalte durch einen UPDATE-Befehl aus der temporären Spalte in die neue Spalte, wobei sie passend konvertiert werden müssen.
- Löschen Sie die temporäre Spalte.

Dieser Weg wird im Kapitel **ÄNDERUNG DER DATENBANKSTRUKTUR**¹ benutzt.

27.3.4. DROP COLUMN – Spalte entfernen

Diese Aufgabe entfernt eine Spalte aus der Tabelle, z. B. die eben erwähnte temporäre Spalte.

```
alter table Abteilung
DROP COLUMN Temp;
```

Durch Löschen einer Spalte wird nicht der Speicherplatz der Spalte freigegeben. Dafür muss (sofern erforderlich) ein vollständiges Backup + Restore der Datenbank ausgeführt werden. Aber das machen Sie ja sowieso regelmäßig.

27.3.5. ADD CONSTRAINT – Einschränkung hinzufügen

Diese Aufgabe erweitert die Bedingungen, die für die Daten der Tabelle gelten sollen.

```
ALTER TABLE <Tabellenname>
ADD [ CONSTRAINT <constraint name> ] <Inhalt> ;
```

1 Kapitel 34 auf Seite 443

Einzelheiten zum <Inhalt> stehen unten im Abschnitt CONSTRAINTs – EINSCHRÄNKUNGEN².

27.3.6. DROP CONSTRAINT – Einschränkung entfernen

Diese Aufgabe löscht eine Bedingung, die für die Daten der Tabelle gültig war.

```
ALTER TABLE <Tabellenname>
DROP CONSTRAINT <constraint name>;
```

27.4. CONSTRAINTs – Einschränkungen

Dabei handelt es sich um Bedingungen, denen ein Datensatz entsprechen muss. Wenn eine der aktuell gültigen Bedingungen verletzt wird, wird der betreffende Datensatz nicht gespeichert. Die Bedingungen können Folgendes betreffen:

- die Schlüssel innerhalb der Tabelle: PRIMARY KEY, INDEX
- die Beziehungen zu anderen Tabellen: FOREIGN KEY
- die Werte innerhalb der Spalte: UNIQUE, CHECK

Ein CONSTRAINT kann mit oder ohne eigenen Namen festgelegt werden. *Wir empfehlen die Benutzung eines Namens, weil dies die Arbeit übersichtlicher macht: Bei Verletzung einer Regel wird dieser Name meistens angegeben; anhand des Namens ist das Löschen direkt möglich. Aber das ist Geschmackssache und hängt wohl auch vom DBMS ab.* Das Schlüsselwort CONSTRAINT selbst ist nur erforderlich bei Verwendung des Namens; ansonsten würden die Schlüsselwörter der einzelnen Bedingungen ausreichen.

Ein CONSTRAINT wird auf eine der folgenden Arten festgelegt:

- im CREATE TABLE-Befehl bei einer einzelnen Spalte als Bedingung für diese Spalte
- im CREATE TABLE-Befehl als Bedingung für die Tabelle, also in der Liste der <Einschränkungen>
- im ALTER TABLE-Befehl durch ADD CONSTRAINT

² [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23CONSTRAINTs%20%20EINSCHR%20%20E4NKUNGEN](http://de.wikibooks.org/wiki/%23CONSTRAINTs%20%20EINSCHR%20%20E4NKUNGEN)

Ein CONSTRAINT wird wie folgt gelöscht:

```
ALTER TABLE <Tabellenname>  
DROP CONSTRAINT <constraint name>;
```

Ein CONSTRAINT kann nicht geändert werden. Es ist nur Löschen und erneute Festlegung möglich.

Welche Wörter der Schlüsselbegriffe optional sind (z. B. KEY), hängt von der konkreten Situation ab.

27.4.1. PRIMARY KEY – Primärschlüssel der Tabelle

Der Primärschlüssel – **PRIMARY KEY** mit PK als gängiger Abkürzung – ist das wichtigste Mittel, mit dem die Datenbank alle Einträge verwaltet. Ohne PK sind weder Änderungen noch Löschungen einzelner Datensätze möglich, ohne alle Spalten anzugeben. Im praktischen Einsatz haben Tabellen ohne Primärschlüssel keinen Sinn. Fremdschlüssel (FOREIGN KEYS, FK) wären ohne Primärschlüssel nicht möglich.

Als Primärschlüssel geeignet sind folgende Arten von Spalten:

- der Datentyp GUID
- eine Spalte mit einem INTEGER-Datentyp, der als AUTO_INCREMENT verwendet wird oder ersatzweise durch eine SEQUENCE bestimmt wird

Die Beispieldatenbank benutzt ausnahmslos eine solche Spalte namens *ID*.

- eine Spalte mit einem INTEGER-Datentyp, sofern die Werte nach Lage der Dinge eindeutig sind und während der „Lebenszeit“ der Datenbank nicht mehr geändert werden.

Die Beispieldatenbank enthält in der Tabelle *Mitarbeiter* die Spalte *Personalnummer*. Diese ist eigentlich eindeutig und dürfte deshalb als PK verwendet werden. Da die Firma aber ihre interne Struktur ändern und die Personalnummern anpassen könnte, scheidet diese Spalte als PK aus.

- eine Kombination aus zwei Spalten, von denen jede dem PK jeweils einer anderen Tabelle entspricht, wenn die „neue“ Tabelle nur die Zuordnungen zwischen den beiden anderen Tabellen darstellt.

Die Tabelle *Zuordnung_SF_FZ* der Beispieldatenbank enthält die Zuordnungen Fahrzeuge/Schadensfälle; anstelle einer eigenen ID wäre auch ein Primärschlüssel aus *Fahrzeug_ID* plus *Schadensfall_ID* möglich und sinnvoll.

Als Primärschlüssel ungeeignet oder unmöglich sind folgende Arten von Spalten:

- Unmöglich sind sämtliche Spalten (wie eine PLZ), bei denen mehrere Datensätze mit dem gleichen Wert vorkommen können.
- Unmöglich ist eine Kombination von Name/Vorname bei allen Tabellen mit Namen, weil über kurz oder lang ein „Müller, Hans“ doppelt vorkommen dürfte.
- Auch eine Kombination von Name/Vorname/Geburtstag scheidet aus dem gleichen Grund aus.
- Eine Kombination von Name/Geburtstag/Region/lfd.Nr. (ähnlich wie bei der Versicherungsnummer der deutschen Rentenversicherung) ist zwar eindeutig, aber als Kombination von vier Spalten äußerst unpraktisch.

Der Primärschlüssel kann wie folgt festgelegt werden:

- im CREATE TABLE-Befehl bei einer einzelnen Spalte als Zuordnung für diese Spalte

MySQL Quelltext

```
create table Abteilung
( ID          INTEGER      not null auto_increment primary key,
  Kuerzel     VARCHAR(10)  not null,
  Bezeichnung VARCHAR(30)  not null,
  Ort         VARCHAR(30)
);
```

Die Spalte *ID* wird direkt als (einzige) Spalte des PK definiert. Dies ist implizit ein CONSTRAINT, bekommt aber keinen eigenen Namen.

- im CREATE TABLE-Befehl in der Liste der <Einschränkungen>

Firebird Quelltext

```
CREATE TABLE Abteilung
( ID          INTEGER,
  Kuerzel     VARCHAR(10) not null,
  Bezeichnung VARCHAR(30) not null,
  Ort         VARCHAR(30)
  CONSTRAINT Abteilung_PK PRIMARY KEY (ID)
```

```
);
```

Der PK bekommt als CONSTRAINT einen eigenen Namen, der Vermerk in Klammern führt die Spalten auf, die als PK verwendet werden (hier wie meistens handelt es sich um eine einzelne Spalte).

- im ALTER TABLE-Befehl durch ADD CONSTRAINT

MySQL-Version mit zwei aufeinanderfolgenden Befehlen

MySQL Quelltext

```
create table Abteilung
( ID          INTEGER      not null auto_increment,
  Kuerzel     VARCHAR(10)  not null,
  Bezeichnung VARCHAR(30)  not null,
  Ort        VARCHAR(30)
);
alter Table Abteilung
add CONSTRAINT Abteilung_PK Primary Key (ID);
```

Die Tabelle erhält zunächst noch keinen PK, auch wenn das durch *AUTO_INCREMENT* suggeriert und vorbereitet wird. Vielmehr wird der PK anschließend (mit eigenem Namen) definiert; der Vermerk in Klammern führt die Spalten auf, die als PK verwendet werden.

27.4.2. UNIQUE – Eindeutigkeit

Ein **UNIQUE KEY** sorgt dafür, dass innerhalb einer Spalte bzw. einer Kombination von Spalten kein Wert doppelt auftreten kann. Beispiele sind in der Tabelle *Mitarbeiter* die Spalte *Personalnummer* und in der Tabelle *Fahrzeug* die Spalte *Kennzeichen*.

Eine solche Eindeutigkeitsbedingung kann wie folgt festgelegt werden:

- im CREATE TABLE-Befehl bei einer einzelnen Spalte als Zuordnung für diese Spalte

```
create table Fahrzeug
( ID          INTEGER      not null auto_increment primary key,
  Kennzeichen VARCHAR(10)  not null                unique,
  Farbe       VARCHAR(30),
  Fahrzeugtyp_ID INTEGER    not null
);
```

- im CREATE TABLE-Befehl in der Liste der <Einschränkungen> mit Bezug auf eine oder mehrere Spalten

```
CREATE TABLE Fahrzeug
( ID          INTEGER      not null auto_increment,
  Kennzeichen VARCHAR(10) not null,
  Farbe       VARCHAR(30),
  Fahrzeugtyp_ID INTEGER    not null,
  CONSTRAINT Fahrzeug_PK PRIMARY KEY (ID),
  CONSTRAINT Fahrzeug_Kz UNIQUE (Kennzeichen)
);
```

- im ALTER TABLE-Befehl durch ADD CONSTRAINT

```
alter TABLE Fahrzeug
ADD [ CONSTRAINT Fahrzeug_Kz ] UNIQUE (Kennzeichen);
```

27.4.3. INDEX – Suche beschleunigen

Ein **INDEX** ist ein Verfahren innerhalb einer Datenbank, mit dem schnell auf Datensätze zugegriffen werden kann. Vor allem der PK benutzt selbst einen Index. (Intern arbeiten die DBMS unterschiedlich; für den Nutzer sieht es immer so aus, als wenn der PK ein Index *ist*.) Sämtliche Spalten bzw. Kombinationen von Spalten, nach denen in SELECT-Befehlen häufiger gesucht oder sortiert wird, sollten mit einem Index versehen werden. Beispiele:

- Name/Vorname sowie PLZ und separat PLZ/Name (vielleicht auch PLZ/Straße) in Tabellen mit Adressen
- solche Spalten, die für den Nutzer wie ein PK aussehen, es aber nicht sind, z. B. in der Tabelle *Mitarbeiter* die Spalte *Personalnummer* und in der Tabelle *Fahrzeug* die Spalte *Kennzeichen*

Die Angabe UNIQUE für eine Spalte sorgt bereits für einen Index; eine doppelte Festlegung ist nicht nötig.

- das Datum in der Tabelle *Schadensfall*

Wenn vorzugsweise die größeren Werte zuerst benötigt werden, ist mit DESC ein absteigender Index (wie im folgenden Beispiel) sinnvoll. In manchen Fällen sind durchaus zwei getrennte Indizes auf dieselbe Spalte angebracht – der eine ASC, der andere DESC.

Ein Index ist nicht sinnvoll, wenn eine Spalte nur wenige verschiedene Werte enthalten kann wie in der Tabelle *Versicherungsvertrag* die Spalte *Art*.

Ein Index kann wie folgt festgelegt werden:

- im CREATE TABLE-Befehl in der Liste der <Einschränkungen> mit Bezug auf eine oder mehrere Spalten

MySQL Quelltext

```
CREATE TABLE Schadensfall
  ( ID      INTEGER      NOT NULL AUTO_INCREMENT PRIMARY KEY,
    Datum  DATE          NOT NULL,
    Ort    VARCHAR(200) NOT NULL,
    /* usw. */
    INDEX Schadensfall_Datum (Datum DESC)
  );
```

In der Tabelle wird sofort ein Index (mit eigenem Namen) für die Spalte *Datum* angelegt, und zwar sollen die Werte absteigend verwaltet werden. (Das bedeutet, dass die Daten mit dem „größten“ Wert, also die aktuellsten Werte zuerst gefunden werden.) Der Standardwert, der nicht angegeben werden muss, ist ASC (= aufsteigend).

- mit einem zusätzlichen CREATE INDEX-Befehl in folgender Syntax:

```
CREATE [ UNIQUE ] INDEX <Indexname>
  ON <Tabellenname> ( <Spaltenliste> );
```

„Irgendwo“ (unterschiedlich nach DBMS) kann außerdem ASC bzw. DESC festgelegt werden.

Das vorige Beispiel sieht dann unter Firebird (DESC vorgezogen) so aus:

Firebird Quelltext

```
CREATE DESC INDEX Schadensfall_Datum
  ON Schadensfall (Datum);
```

Die Eindeutigkeitsbedingung UNIQUE benutzt intern (vermutlich immer) ebenfalls einen INDEX; dieser kann auch ausdrücklich angegeben werden:

```
ALTER TABLE Fahrzeug
  ADD CONSTRAINT Fahrzeug_Kennzeichen UNIQUE (Kennzeichen)
```

```
USING INDEX Fahrzeug_Kennzeichen_UK;
```

27.4.4. FOREIGN KEY – Fremdschlüssel

Ein **FOREIGN KEY** (FK) regelt die logischen Verbindungen zwischen zwei Tabellen: Ein Datensatz in einer Tabelle darf in einer bestimmten Spalte nur solche Werte benutzen, die in einer anderen Tabelle als PK registriert sind. Beispiele:

- In der Tabelle *Mitarbeiter* darf als *Abteilung_ID* nur eine gültige ID der Tabelle *Abteilung* stehen.
- In der Tabelle *Fahrzeug* darf als *Fahrzeugtyp_ID* nur eine gültige ID der Tabelle *Fahrzeugtyp* stehen.
- In der Tabelle *Fahrzeugtyp* darf als *Hersteller_ID* nur eine gültige ID der Tabelle *Fahrzeughersteller* stehen.

Ein Fremdschlüssel kann wie folgt festgelegt werden:

- im CREATE TABLE-Befehl bei einer einzelnen Spalte als Bedingung für diese Spalte
- im CREATE TABLE-Befehl als Bedingung für die Tabelle, also in der Liste der <Einschränkungen>
- im ALTER TABLE-Befehl durch ADD CONSTRAINT

Einzelheiten werden im Kapitel **FREMDSCHLÜSSEL-BEZIEHUNGEN**³ behandelt.

27.4.5. CHECK – Werteprüfungen

Ein **CHECK** ist eine Prüfung, ob die Werte, die für einen Datensatz gespeichert werden sollen, bestimmten Regeln entsprechen. Diese Prüfung wird sowohl bei INSERT als auch bei UPDATE vorgenommen; sie kann für eine einzelne Spalte oder für die Tabelle festgelegt werden.

Als Bedingung in der CHECK-Klausel kann im Wesentlichen alles stehen, was für die WHERE-Klausel vorgesehen ist.

Eine solche Prüfung kann wie folgt festgelegt werden:

- im CREATE TABLE-Befehl bei einer einzelnen Spalte als Bedingung für diese Spalte

3 Kapitel 28 auf Seite 343

► **Aufgabe:** In der Tabelle *Schadensfall* sind als *Schadenshoehe* natürlich keine negativen Zahlen zulässig. Die Spalte *Verletzte* ist als CHAR(1) definiert; sinnvollerweise sind nur die Werte 'J' und 'N' zulässig.

```
create table Schadensfall
( ID          INTEGER          not null auto_increment primary key,
  Datum       DATE             not null,
  Ort         VARCHAR(200)     not null,
  Beschreibung VARCHAR(1000)  not null,
  Schadenshoehe DECIMAL(16,2) check(Schadenshoehe >= 0),
  Verletzte   CHAR(1)         not null check(Verletzte = 'J' OR
Verletzte = 'N'),
  Mitarbeiter_ID INTEGER      not null
);
```

- im CREATE TABLE-Befehl als Bedingung für die Tabelle, also in der Liste der <Einschränkungen>

► **Aufgabe:** Wenn ein Versicherungsnehmer eine Person ist, müssen Vorname, Geburtsdatum und Führerschein registriert sein. Außerdem muss er mindestens 16 Jahre alt sein.

```
CREATE TABLE Versicherungsnehmer
( ID          INTEGER          NOT NULL auto_increment Primary
Key,
  Name        VARCHAR(30) NOT NULL ,
  Vorname     VARCHAR(30) NOT NULL ,
  Geburtsdatum DATE ,
  Fuehrerschein DATE ,
  /* usw. für alle anderen Spalten, danach: */
  CONSTRAINT Versicherungsnehmer_CheckDatum
CHECK( ( (Geburtsdatum is null)
        and (Fuehrerschein is null)
        and (Vorname is null or Vorname = "") )
or (Fuehrerschein >= Geburtsdatum + 365*16) ));
```

Die ersten Bedingungen prüfen, ob es sich um eine Person handelt; wenn nicht, sind Führerscheinprüfung und Geburtsdatum irrelevant, und der Datensatz kann gespeichert werden. Wenn es sich um eine Person handelt, wird auch die letzte Bedingung benötigt; diese wird „näherungsweise“ geprüft und berücksichtigt, dass das DBMS ein Datum intern als ganze Zahl speichert. *Alternativ könnten auch mit EXTRACT() Tag, Monat, Jahr getrennt verglichen werden. Dieses Verfahren wäre aber deutlich umständlicher; deshalb sollte es hier nicht stehen.*

- im ALTER TABLE-Befehl durch ADD CONSTRAINT

► **Aufgabe:** In der Tabelle *Versicherungsvertrag* sind als *Art* nur bestimmte Werte zulässig: 'VK' (= Vollkasko), 'TK' (= Teilkasko incl. Haftpflicht), 'HP' (= Haftpflicht).

```
alter table Versicherungsvertrag
add CONSTRAINT Vertrag_CheckArt
check (Art in ('VK', 'TK', 'HP') );
```

27.5. Zusammenfassung

In diesem Kapitel lernten wir mehrere Verfahren kennen, mit denen einzelne Spalten und ganze Tabellen genauer festgelegt werden:

- Zu einer Spalte gehören nicht nur der Datentyp, sondern auch die Vorgabe von Werten und Wertebereichen.
- Für eine Spalte können Einschränkungen wie „Eindeutigkeit“ oder „Teil des Primärschlüssels“ oder „Teil eines Index“ gelten.
- Für eine Tabelle können Wertebereiche über mehrere Spalten geprüft werden.
- Eigentlich immer gehört zu einer Tabelle ein Primärschlüssel.
- Außerdem können Indizes und Fremdschlüssel festgelegt werden.

27.6. Übungen

Übung 1 – Definitionen

Welche der folgenden Aussagen sind wahr, welche sind falsch?

1. Zur Definition einer Tabelle gehört unbedingt die Definition der Spalten.
2. Zur Definition einer Tabelle gehört unbedingt die Definition des Primärschlüssels.
3. Zur Definition einer Tabelle gehören unbedingt die Klammern.
4. Die Definition von Einschränkungen ist während des CREATE-Befehls oder durch einen ALTER-Befehl möglich.
5. Als UNIQUE darf nur eine Spalte festgelegt werden.
6. Jede Spalte kann als NOT NULL festgelegt werden.
7. Für jede Spalte können Vorgabewerte festgelegt werden.
8. Es gibt Situationen, in denen die Definition einer Spalte nicht geändert werden kann.

9. Der Begriff CONSTRAINT gehört zur Definition einer Einschränkung.
10. Ein Primärschlüssel kann über beliebig viele Spalten festgelegt werden.
11. Es ist üblich, dass der Wert eines Primärschlüssels immer wieder einmal geändert werden.

Übung 2 – Tabellendefinition

Bitte geben Sie an, welche Bestandteile der folgenden Definition falsch sind bzw. welche Angaben fehlen.

```
create table Computer
constraint ComputerID primary key (Nummer)
unique Name,
Name not null varchar collation Win1252
Nummer integer primary Key
Hersteller varchar(30)
Herstellung date
Festplatte long default 320*1024*1024*1024
Ram_Groesse long,
;
```

Hinweis: Bei den folgenden Definitionen verwenden Sie bitte für alle Einschränkungen geeignete Namen.

Übung 3 – Einschränkungen allgemein definieren

Erstellen Sie die Definition für eine Tabelle mit internationalen Postleitzahlen: laufende Nummer, Land, Code, Ortsname. Legen Sie für jede Spalte möglichst viele Einzelheiten fest; bei der Reihenfolge der Einzelheiten müssen Sie wegen der Unterschiede der DBMS nur auf die CONSTRAINTS achten.

Übung 4 – Spalten mit Einschränkungen hinzufügen

Ergänzen Sie die Tabelle *Versicherungsvertrag* um folgende Spalten:

- Basisprämie für einen Betrag, Vorgabewert 500, keine negativen Beträge
- Prämienatz für eine Zahl, Vorgabewert 100, Minimalwert 10 [%]

Übung 5 – Einschränkung und Index hinzufügen

Ändern Sie die Tabelle *Versicherungsvertrag* so, dass die Spalte *Vertragsnummer* eindeutig ist und einen (ausdrücklich angegebenen) Index benutzt.

Übung 6 – Einschränkung hinzufügen

Ändern Sie die Tabelle *Versicherungsnehmer* so, dass die Spalte *Eigener_Kunde* nur die Werte 'J' und 'N' annehmen darf.

27.7. Lösungen

Lösung zu Übung 1 – Definitionen

Die Aussagen 1, 3, 4, 6, 7, 8, 10 sind wahr, die Aussagen 2, 5, 9, 11 sind falsch.

Lösung zu Übung 2 – Tabellendefinition

- Es fehlen die Klammern um die gesamte Auflistung aller Einzelheiten, und es fehlen Kommata nach jedem einzelnen Bestandteil. Dagegen ist das letzte Komma falsch.
- Zeilen 2/3: Diese Zeilen gehören an das Ende: Zuerst müssen die Spalten festgelegt werden, dann kann darauf Bezug genommen werden.
- Zeile 4: Es fehlt die Größenangabe für die Zeichenkette. Ob die Reihenfolge der Teile passt, hängt vom DBMS ab.
- Zeile 2 und 5: Doppelte Festlegung des Primary Key. Es bezieht sich zwar in beiden Fällen auf dieselbe Spalte, es sind aber wegen des CONSTRAINT-Namens unterschiedliche Definitionen.

Lösung zu Übung 3 – Einschränkungen allgemein definieren

```
create Table PLZ_Codes
( ID INTEGER not null auto_increment
  CONSTRAINT PLZ_PK primary key,
  Land CHAR ( 2) not null default 'DE',
  Code VARCHAR(10) not null,          -- auch CHAR(10) ist
  denkbar
  Ort VARCHAR(30) not null,
  CONSTRAINT PLZ_UK unique (Land, Code)
);
```

Lösung zu Übung 4 – Spalten mit Einschränkungen hinzufügen

```
ALTER TABLE Versicherungsvertrag
  ADD [COLUMN] Basispraemie DECIMAL
  DEFAULT 500 NOT NULL
  CONSTRAINT Vertrag_Basispraemie_Check CHECK(Basispraemie > 0),
  ADD [COLUMN] Praemiensatz INTEGER
  DEFAULT 100 NOT NULL
  CONSTRAINT Vertrag_Praemiensatz_Check CHECK(Praemiensatz >= 10);
```

Lösung zu Übung 5 – Einschränkung und Index hinzufügen

```
ALTER TABLE Versicherungsvertrag
  ADD CONSTRAINT Versicherungsvertrag_Nummer UNIQUE (Vertragsnummer)
  USING INDEX Versicherungsvertrag_Nummer_UK;
```

Lösung zu Übung 6 – Einschränkung hinzufügen

```
ALTER TABLE Versicherungsnehmer
  ADD CONSTRAINT Versicherungsnehmer_Eigener_Kunde
  CHECK( Eigener_Kunde = 'J' OR Eigener_Kunde = 'N' );
```

27.8. Siehe auch

In den folgenden Kapiteln sind Einzelheiten zu finden:

- DATENTYPEN⁴
- TABELLENSTRUKTUR DER BEISPIELDATENBANK⁵
- WHERE-KLAUSEL IM DETAIL⁶

Bei Wikipedia gibt es grundlegende Erläuterungen:

- GLOBALLY UNIQUE IDENTIFIER⁷ (GUID) als eindeutige Kennung
- VERSICHERUNGSNUMMER⁸ der deutschen Rentenversicherung

4 Kapitel 12 auf Seite 107

5 Kapitel 36 auf Seite 469

6 Kapitel 16 auf Seite 173

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/GLOBALLY%20UNIQUE%20IDENTIFIER](http://de.wikipedia.org/wiki/Globally%20Unique%20Identifier)

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/VERSICHERUNGSNUMMER](http://de.wikipedia.org/wiki/Versicherungsnummer)

28. Fremdschlüssel-Beziehungen

In diesem Kapitel werden wir die Verknüpfungen zwischen Tabellen über Fremdschlüssel behandeln.

28.1. Problemstellung

In der Beispieldatenbank gibt es viele Verweise von einer Tabelle auf andere Tabellen:

- Zu jedem Eintrag der Tabelle *Versicherungsvertrag* gehört genau ein Eintrag der Tabelle *Fahrzeug*: Verträge und Fahrzeuge gehören unmittelbar zusammen.
- Zu jedem Eintrag der Tabelle *Versicherungsvertrag* gehört ein Eintrag der Tabelle *Versicherungsnehmer*: Ein Vertrag ohne einen Kunden ist sinnlos.
- Einem Eintrag der Tabelle *Versicherungsnehmer* sind Einträge der Tabelle *Versicherungsvertrag* zugeordnet: Ein Kunde kann einen oder mehrere Verträge halten; zu ihm können auch null Verträge gehören, wenn er nämlich aktuell keinen Vertrag hat, aber als potenzieller Kunde weiterhin registriert ist.
- Zu jedem Vertrag gehört ein Eintrag der Tabelle *Mitarbeiter*: Jeder Vertrag wird hauptsächlich von einem bestimmten Mitarbeiter bearbeitet. *Das ist zwar in der Praxis nicht so eng zu sehen, wird aber in unserer theoretischen Firma so geregelt.*
- Umgekehrt gehört nicht zu jedem Eintrag der Tabelle *Mitarbeiter* ein Vertrag – z. B. bei der Abteilung „Forschung und Entwicklung“.

Es wäre viel Aufwand, wenn ein Anwender alle diese Bedingungen immer selbständig beachten müsste. In einem Büro kann man immer einmal gestört werden; und schon fehlt eine der unbedingt erforderlichen Informationen: Ein neuer Vertrag wird gespeichert, aber der Kunde fehlt noch; dann kommt ein Telefonat dazwischen; dann macht die Mitarbeiterin mit dem nächsten Vertrag weiter. Und wohin soll die Rechnung zum vorigen Vertrag gehen?

Viel sinnvoller ist es, wenn das DBMS in die Lage versetzt wird, diese Bedingungen direkt zu berücksichtigen. Dies wird über die Fremdschlüssel – englisch: ForeignKeys (FK) – geregelt.

28.2. Grundsätze der Lösung

28.2.1. Beziehungen beschreiben

Bei all diesen Beziehungen geht es darum, dass die Verbindungen zwischen den Tabellen und Querverweise immer auf dem aktuellen Stand sind und die Werte in allen Tabellen korrekt zusammenpassen. Das wird insgesamt als „Referentielle Integrität“ bezeichnet und durch die „Fremdschlüssel“ gewährleistet. *Weiter unten wird es als „interne Datensicherheit“ bezeichnet: Die Daten sollen innerhalb der Datenbank insofern sicher sein, dass keine Widersprüche zwischen den Daten in verschiedenen Tabellen auftreten.*

Durch die Fremdschlüssel werden Beziehungen zwischen den Tabellen definiert. Das DBMS sorgt dann für die richtigen Verknüpfungen:

- Kein neuer Vertrag kann ohne Fahrzeug-ID, Versicherungsnehmer-ID und Mitarbeiter-ID eingetragen werden.
- Die entsprechenden Datensätze in den zugeordneten Tabellen müssen zuerst gespeichert werden, bevor ein Vertrag neu aufgenommen werden kann.
- Kein Versicherungsnehmer kann gelöscht werden, solange noch Verträge vorliegen.
- Bei entsprechender Festlegung können zunächst alle Verträge eines Kunden und sofort der Kundensatz selbst gelöscht werden.

Ergänzend sind weitere Bedingungen denkbar:

- Wenn Kundennummern geändert werden, müssen die abhängigen Verträge ebenso geändert werden (Update-Weitergabe); oder Änderungen an Kundennummern werden erst gar nicht zugelassen (Update-Restriktion).

Die letzten Bedingungen können vernachlässigt werden, da die Beziehungen über die IDs geregelt werden. Nach den Regeln der relationalen Datenbanken hat die ID keine andere Bedeutung als die Identifizierung der Datensätze; es gibt niemals die Notwendigkeit, sie zu ändern.

28.2.2. Beziehungen definieren

In der Tabellenstruktur der Beispieldatenbank sind bei den einzelnen Tabellen in der Spalte *Erläuterung* die Verknüpfungen aufgeführt, beispielsweise:

- Die Tabelle *Versicherungsvertrag* enthält die folgenden Verweise:
 - Die Spalte *Versicherungsnehmer-ID* verweist auf eine ID der Tabelle *Versicherungsnehmer*.
 - Die Spalte *Fahrzeug-ID* verweist auf eine ID der Tabelle *Fahrzeug*.
 - Die Spalte *Mitarbeiter-ID* verweist auf eine ID der Tabelle *Mitarbeiter*.
- Die Tabelle *Zuordnung_SF_FZ* enthält die folgenden Verweise:
 - Die Spalte *Schadensfall-ID* verweist auf eine ID der Tabelle *Schadensfall*.
 - Die Spalte *Fahrzeug-ID* verweist auf eine ID der Tabelle *Fahrzeug*.
- Die Tabelle *Versicherungsnehmer* enthält den folgenden Verweis:
 - Die Spalte *Versicherungsgesellschaft-ID* verweist optional (nämlich nur bei Fremdkunden) auf eine ID der Tabelle *Versicherungsgesellschaft*.

In allen Fällen bedeuten die Verweise: In einem Datensatz der Haupttabelle stehen keine Einzelheiten der betreffenden Spalte. Stattdessen steht dort eine ID; die Angaben dazu sind im Datensatz der zugeordneten Tabelle unter der genannten ID zu finden. Soweit es sich um einen optionalen Verweis handelt, kann in der Haupttabelle auch der NULL-Wert stehen; ob ein solcher Verweis mit einem Fremdschlüssel automatisiert gesteuert werden kann, hängt vom DBMS ab.

28.3. Syntax und Optionen

28.3.1. Die Definition

Auf der Ebene von Tabellen bezeichnet man die Tabelle, auf deren Primärschlüssel verwiesen wird, auch als **Parent-Tabelle**; im ersten Beispiel der Auflistung sind dies die Tabellen *Versicherungsnehmer*, *Fahrzeug* und *Mitarbeiter*. Die Tabelle, die den bzw. die Fremdschlüssel enthält, bezeichnet man als **Child-Tabelle**. In der Literatur werden auch die Begriffe **Master-Tabelle** und **Detail-Tabelle** und manchmal die deutschen Bezeichnungen **Eltern-Tabelle** und **Kind-Tabelle** verwendet.

i **Achtung:**

Diese Begriffe sind missverständlich und werden manchmal auch im entgegengesetzten Sinn verstanden.

Im letzten Abschnitt haben wir davon gesprochen, dass in der „Haupttabelle“ nur Verweise auf die „zugeordneten“ Tabellen stehen. Dort sind dann die Einzelheiten zu finden – also wäre die Haupttabelle die Detail-Tabelle, und die Einzelheiten-Tabelle wäre die Master-Tabelle. Was denn nun?

Mit dem Beispiel im Wikipedia-Artikel „Fremdschlüssel“ wird die Bedeutung von **Master** und **Detail** verständlicher: Die Kunden sind die maßgebende Tabelle; zu jedem Kunden gehören als Details „seine“ Bestellungen.

Wir vermeiden deshalb in diesem Kapitel diese Begriffe und sprechen von Haupttabelle mit Fremdschlüsseln: Eine Spalte in der (Haupt-) **Tabelle** wird über den **Fremdschlüssel** verknüpft mit dem **Primärschlüssel** in der **zugeordneten Tabelle**.

28.3.2. Die Syntax

Die Fremdschlüssel gehören zur Definition der Tabellen, ihre Definition über FOREIGN KEY ist also Bestandteil der Data Definition Language (DDL). Wie bei anderen Erweiterungen der Tabellendefinition gibt es mehrere Wege:

- Bei der Definition einer einzelnen Spalte, die den Fremdschlüssel benutzt:

```
<Definition der Spalte>
-- ergänzt durch:
  REFERENCES <abhängige Tabelle> ( <Spaltenname> )
[ <Optionen> ]
```

- Bei der Definition der Tabelle in der Liste der Einschränkungen (CONSTRAINTS):

```
[ CONSTRAINT <Constraint-Name> ]
  FOREIGN KEY ( <Spaltenname> )
  REFERENCES <abhängige Tabelle> ( <Spaltenname> )
[ <Optionen> ]
```

- Als Änderung der Tabellendefinition mit ALTER TABLE:

```
ALTER TABLE <Haupttabelle>
  ADD [ CONSTRAINT <Constraint-Name> ]
  FOREIGN KEY ( <Spaltenname> )
  REFERENCES <abhängige Tabelle> ( <Spaltenname> )
[ <Optionen> ]
```

Zu dem Zeitpunkt, an dem ein FOREIGN KEY festgelegt wird, muss die Tabelle, auf die verwiesen wird, bereits definiert sein. Der letzte Weg ist deshalb in der

Regel am sichersten: zuerst werden alle Tabellen bestimmt, danach alle FOREIGN KEYS als Verknüpfung.

Zur eigentlichen Definition gehören die folgenden Bestandteile:

- **CONSTRAINT** <Name>
Dies legt den Namen dieser Einschränkung fest und kann auch entfallen.
- **FOREIGN KEY** <Spaltenname>
Dies ist der Hinweis auf einen Fremdschlüssel und bestimmt, zu welcher Spalte dieser gehört.
Bei der Definition einer einzelnen Spalte (erste Variante) ist von vornherein klar, um welche Spalte es sich handelt; dies muss deshalb nicht wiederholt werden und entfällt bei dieser Variante.
- **REFERENCES** <Tabellename>
Dies bestimmt die Tabelle, auf die mit dem Fremdschlüssel verwiesen wird.
- <Spaltenname> in Klammern gesetzt
Damit wird festgelegt, welche Spalten in den beiden Tabellen miteinander in Beziehung gesetzt werden.

Die Optionen werden im nächsten Abschnitt erläutert.

Der Vollständigkeit halber sei darauf hingewiesen: Eine solche Verknüpfung kann sich auch auf mehrere Spalten beziehen, nämlich sowohl als Primärschlüssel als auch als Fremdschlüssel. Da ein Primärschlüssel keine weitere Bedeutung haben soll – siehe die Erläuterungen im vorigen Kapitel zum PRIMARY KEY –, besteht er sowieso (fast) immer aus einer einzelnen Spalte; deshalb wird in der Übersicht nur ein <Spaltenname> erwähnt.

28.3.3. Optionen

Die Optionen eines FOREIGN KEY bestimmen das Verhalten der Tabelle, die die Verweise (Fremdschlüssel) benutzt – bei uns als Haupttabelle bezeichnet –, sobald in der zugeordneten Tabelle die Primärschlüssel geändert werden. Allgemein steht folgendes Verhalten zur Auswahl:

- **NO ACTION** – alle Änderungen werden verweigert
- **CASCADE** – die Weitergabe der Änderung an die Haupttabelle
- **RESTRICT** – die Verweigerung der Änderung auch in der zugeordneten Tabelle
- **SET NULL** – die Änderung des Verweises in der Haupttabelle auf NULL
- **SET DEFAULT** – die Änderung des Verweises in der Haupttabelle auf den Vorgabewert der Spalte

Die verschiedenen Update- und Löschoptionen werden nicht von allen DBMS unterstützt. Die Option ON UPDATE CASCADE z. B. wird von den meisten DBMS nicht angeboten.

Im einzelnen wirken sich diese Optionen wie folgt aus.

Bei **Neuaufnahmen** in der zugeordneten Tabelle sind die Datensätze in der übergeordneten Tabelle noch nicht vorhanden. Also kann es keine Probleme geben; deshalb muss dies bei den Optionen nicht beachtet werden.

Die folgenden Optionen wirken sich bei **Änderungen** und **Löschungen** in der zugeordneten Tabelle in gleicher Weise aus:

- **ON UPDATE NO ACTION** und **ON DELETE NO ACTION**

Die „Inaktivität“ bedeutet: Wenn ein Primärschlüssel in der zugeordneten Tabelle geändert bzw. gelöscht werden soll und abhängige Sätze in der Haupttabelle existieren, dann wird die Änderung/Löschung mit einem Fehler abgebrochen; es erfolgt ein ROLLBACK.

- **ON UPDATE RESTRICT** und **ON DELETE RESTRICT**

Die „Restriktion der Aktualisierung“ bedeutet: Wenn ein Primärschlüssel in der zugeordneten Tabelle geändert bzw. gelöscht werden soll und abhängige Sätze in der Haupttabelle existieren, dann wird die Änderung/Löschung verweigert.

- **ON UPDATE SET NULL** und **ON DELETE SET NULL**

Das „NULL-Setzen“ bedeutet: Wenn ein Primärschlüssel in der zugeordneten Tabelle geändert bzw. gelöscht wird, dann werden die Verweise in der Haupttabelle auf NULL gesetzt. Das ist nur möglich, wenn die betreffende Spalte NULL-Werte zulässt (also *nicht* mit NOT NULL definiert wurde).

- **ON UPDATE SET DEFAULT** und **ON DELETE SET DEFAULT**

Das „DEFAULT-Setzen“ bedeutet: Wenn ein Primärschlüssel in der zugeordneten Tabelle geändert bzw. gelöscht wird, dann werden die Verweise in der Haupttabelle auf den Vorgabewert der betreffenden Spalte gesetzt. Das ist nur möglich, wenn für die betreffende Spalte ein Vorgabewert festgelegt ist.

Die folgende Option wirkt sich bei Änderungen und Löschungen unterschiedlich aus:

- **ON UPDATE CASCADE** also bei **Änderungen**

Mit der „Weitergabe der Aktualisierung“ werden die Fremdschlüssel in der Haupttabelle in der gleichen Weise geändert, wie der Primärschlüssel in der zugeordneten Tabelle geändert wird.

- **ON DELETE CASCADE** also bei **Löschungen**

Die „Löschweitergabe“ bedeutet: Zusammen mit dem Datensatz in der zuge-

ordneten Tabelle werden auch alle Datensätze in der Haupttabelle gelöscht, die sich auf diesen Schlüssel beziehen.

Wenn der Primärschlüssel „richtig“ definiert ist, nämlich für alle Zeiten unveränderlich ist, dann wäre die UPDATE-Option eigentlich überflüssig. *Aber wehe, wenn man doch auf die Idee kommt, einmal einen Primary Key zu ändern; für diesen Fall sollte man vorbereitet sein.*

28.3.4. Auswirkungen

Änderungen in der **zugeordneten Tabelle** haben mit den Optionen nichts zu tun. Sie werden durch einen direkten Befehl – INSERT, UPDATE, DELETE – ausgelöst. Ein Fremdschlüssel steuert mit den Optionen nur, inwieweit eine solche Änderung Auswirkungen auf die „Haupttabelle“ hat oder nicht. Allerdings kann es passieren, dass durch die Restriktion nicht nur die Änderung in der „Haupttabelle“, sondern auch die Änderung in der „zugeordneten Tabelle“ verhindert wird.

Änderungen in der **Haupttabelle** werden wie folgt eingeschränkt, wenn ein FOREIGN KEY festgelegt ist: Bei INSERT und UPDATE dürfen in den Spalten, die für einen Fremdschlüssel benutzt werden, nur solche Werte eingefügt werden, die in der zugeordneten Tabelle als Primärschlüssel vorhanden sind. Einzige Ausnahme ist, wenn die Fremdschlüssel-Spalte als optional definiert ist. Dann kann hier auch NULL eingefügt werden, obwohl NULL niemals als Primärschlüssel in der zugeordneten Tabelle stehen wird.

28.4. Beispiele

28.4.1. Versicherungsvertrag und Kunden

Beginnen wir für die Tabelle *Versicherungsvertrag* mit dem Verweis von der Spalte *Versicherungsnehmer_ID* auf die Tabelle *Versicherungsnehmer*.

```
ALTER TABLE Versicherungsvertrag
ADD CONSTRAINT Versicherungsvertrag_VN
FOREIGN KEY (Versicherungsnehmer_ID)
REFERENCES Versicherungsnehmer (ID);
```

Wie fast immer benutzen wir eine Einschränkung mit Namen. Wie bei den anderen Constraints hängen wir an den Namen der Tabelle „etwas“ an (Suffix), das

für die Art der Einschränkung steht. Wenn eine Tabelle nur einen Fremdschlüssel bekommt, wäre FK als Suffix geeignet. Da zur Tabelle *Versicherungsvertrag* drei Fremdschlüssel gehören, verwenden wir stattdessen den jeweiligen Tabellen-Alias.

Mit diesem CONSTRAINT ist festgelegt: Ein neuer Versicherungsvertrag kann nur dann registriert werden, wenn die dort vorgesehene *Versicherungsnehmer_ID* in der Tabelle *Versicherungsnehmer* als *ID* bereits registriert ist.

Wie gesagt: Eine „richtige“ ID wird niemals mehr geändert. Vorsichtshalber legen wir aber auch die Optionen fest:

```
ALTER TABLE Versicherungsvertrag
ADD CONSTRAINT Versicherungsvertrag_VN
FOREIGN KEY (Versicherungsnehmer_ID)
REFERENCES Versicherungsnehmer (ID)
ON UPDATE RESTRICT
ON DELETE RESTRICT;
```

Die Änderung der ID oder die Löschung eines Versicherungsnehmers ist also nicht zulässig, wenn zu diesem Kunden ein Versicherungsvertrag registriert ist.

28.4.2. Mitarbeiter und Abteilung

Die Beziehung zwischen diesen Tabellen kann so festgelegt werden:

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT Mitarbeiter_FK
FOREIGN KEY (Abteilung_ID)
REFERENCES Abteilung (ID);
ON UPDATE CASCADE
ON DELETE RESTRICT;
```

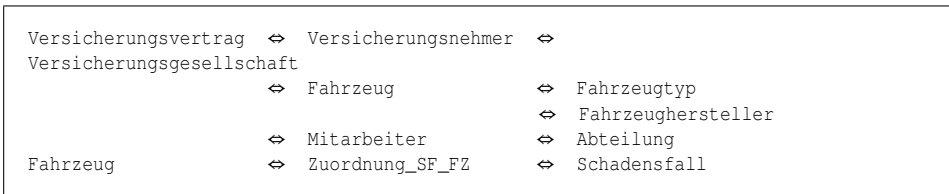
Das DBMS sorgt damit für die interne Datensicherheit:

- Ein neuer Mitarbeiter kann nicht eingetragen werden, ohne dass die *Abteilung_ID* in der Tabelle *Abteilung* als *ID* vorhanden ist.
- Wenn die Nummer einer Abteilung geändert wird, wird das automatisch bei allen ihren Mitarbeitern angepasst und ebenfalls geändert.
- Wenn eine Abteilung gelöscht (d. h. geschlossen) werden soll, tritt folgende Prüfung ein:
 - Durch ON DELETE RESTRICT kann sie nicht gelöscht werden, solange ihr Mitarbeiter zugeordnet sind.

- Würde stattdessen ON DELETE CASCADE vorgemerkt, dann würden beim Löschen einer Abteilung automatisch alle dort arbeitenden Mitarbeiter ebenfalls gestrichen.

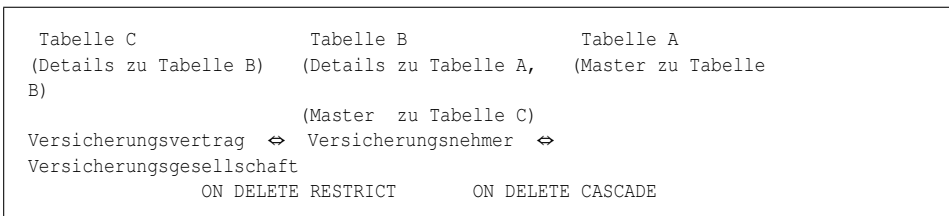
28.5. Kombination von Fremdschlüsseln

Grundsätzlich kann eine Detail-Tabelle (Haupttabelle) gleichzeitig als Master-Tabelle (zugeordnete Tabelle) für eine andere Tabelle definiert werden. In unserer Beispieldatenbank betrifft das die mehrfachen Verknüpfungen:



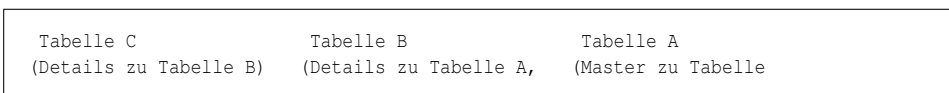
Dann sind aber nicht alle Kombinationen der Optionen CASCADE (Weitergabe) und RESTRICT (Restriktion) zulässig – weder bei DELETE noch bei UPDATE. Das DBMS prüft beim Ausführen der DDL-Befehle, ob die gewünschte Regel zulässig ist.

Nicht zulässig sind die folgende Verknüpfungen:



Erläuterung: Es ist nicht zulässig, einen Versicherungsnehmer zu löschen, wenn zu ihm noch (mindestens) ein Vertrag registriert ist. Andererseits sollen mit einer Versicherungsgesellschaft auch alle ihre Versicherungsnehmer automatisch gelöscht werden. Diese automatische Löschung stünde im Widerspruch zur Verhinderung der Löschung bei vorhandenen Verträgen.

Zulässig sind die folgende Verknüpfungen:



```
B)
                                     (Master zu Tabelle C)
Versicherungsvertrag ↔ Versicherungsnehmer ↔
Versicherungsgesellschaft
                        ON DELETE CASCADE          ON DELETE RESTRICT
```

Erläuterung: Das Löschen der Versicherungsgesellschaft ist nicht zulässig, wenn noch Versicherungsnehmer registriert sind. Damit wird die Weitergabe oder Restriktion der Löschung vom Versicherungsnehmer zum Versicherungsvertrag überhaupt nicht beeinflusst.

Möglich sind auch **Ring-Verkettungen**:

```
Tabelle A ↔ Tabelle B ↔ Tabelle C ↔ Tabelle A
```

Ob Sie diese Verknüpfungen von rechts nach links lesen oder umgekehrt, ist gleichgültig: Eine Tabelle hat Verweise auf eine andere, diese auf eine nächste und eine weitere wieder auf die erste.

28.6. Rekursive Fremdschlüssel

Fremdschlüsselbeziehungen können auch rekursiv definiert werden. Dabei verweist eine abhängige Spalte einer Tabelle auf den Primärschlüssel der eigenen Tabelle.

Als Beispiel verwenden wir eine hierarchische Gliederung der Abteilungen:

```
CREATE TABLE Abteilung
( AbtNr          INTEGER NOT NULL,
  UebergeordneteAbt INTEGER,
  AbtName        VARCHAR(100),
  PRIMARY KEY (AbtNr),
  FOREIGN KEY (UebergeordneteAbt)
    REFERENCES Abteilung (AbtNr)
    ON DELETE CASCADE
)
```

Die gesamte Firma wird mit AbtNr=1 gespeichert; zu ihr gehört keine *UebergeordneteAbt*. Bei jeder anderen Abteilung wird in dieser Spalte die AbtNr derjenigen Abteilung registriert, der sie direkt zugeordnet ist: Eine Hauptabteilung gehört direkt zur Firma, eine beliebige Abteilung zu ihrer Hauptabteilung, eine Arbeitsgruppe zu einer bestimmten Abteilung usw.

28.6.1. Praktische Probleme

Rekursive Fremdschlüsselbeziehungen sind etwas problematisch in der Handhabung.

Die Neuaufnahme von Datensätzen muss in einer bestimmten Reihenfolge geschehen: zuerst die oberste Ebene (Firma), dann die nächste Ebene (Hauptabteilungen) usw. Auf jeder Ebene ist eine Neuaufnahme nur dann möglich, wenn der übergeordnete Eintrag schon vorhanden ist.

Beim Löschen von Datensätzen kann es zu verschiedenen Problemen kommen.

Wenn der Fremdschlüssel mit Lösch-Weitergabe – ON DELETE CASCADE – definiert wurde, kann es zur Löschung von wesentlich mehr Daten kommen, als in der WHERE-Bedingung angegeben wird:

```
DELETE FROM Abteilung  
WHERE AbtNr = 33;
```

Bei diesem Beispiel werden auch alle Sätze gelöscht, die der Abteilung 33 untergeordnet sind.

Wenn der Fremdschlüssel mit Lösch-Restriktion – ON DELETE RESTRICT – definiert wurde, dann wird nach jeder Löschung eines einzelnen Satzes geprüft, ob es keine Fremdschlüsselverletzung gibt. Selbst wenn alle Sätze aus der Tabelle entfernt werden sollen, kann es passieren, dass die Ausführung fehlschlägt.

```
DELETE FROM Abteilung;
```

Das liegt daran, dass bei der Ausführung des DELETE-Befehls die Sätze in einer beliebigen Reihenfolge gelöscht werden, meistens in der Reihenfolge, in der sie „real“ in der Tabelle gespeichert sind. Nur wenn die Sätze exakt in der richtigen Reihenfolge (von der untersten Abteilung beginnend bis zur obersten Abteilung) gespeichert sind, dann kann die Ausführung des DELETE-Statements gelingen.

Dass der Erfolg eines SQL-Befehls von der physischen Speicherreihenfolge der Sätze abhängig ist, darf in einem DBMS nicht vorkommen. Daher bieten einige DBMS die Möglichkeit der **verzögerten Prüfung bei der Löschweitergabe**.

Durch ein einziges DELETE-Statement können mehrere Sätze evtl. auch alle Sätze einer Tabelle gelöscht werden. Innerhalb einer Transaktion können mehrere DELETE-Statements ausgeführt werden. Standardmäßig erfolgt die Prüfung, ob eine Löschoption ausgeführt werden darf, nach jedem einzelnen Satz, der gelöscht wurde. Das hat den Vorteil, dass bei einer unzulässigen Löschung gleich abgebrochen werden kann und der Rollback nicht unnötig viel zu tun hat.

28.6.2. Maßnahmen

Um die oben beschriebenen Löschoption-Anomalien zu vermeiden, kann bei einigen DBMS die Prüfung, ob die Löschung zulässig ist, als Gesamtprüfung stattfinden und nicht nach jedem einzelnen Satz:

- Nach der Löschung aller Sätze, die durch ein DELETE-Statement angewiesen werden, wird der Zusammenhang der Daten überprüft.
Diese Variante wird z. B. von DB2 angeboten durch die Option ON DELETE NO ACTION.
- Erst zum Abschluss der Transaktion wird die Prüfung erledigt.
Diese Variante gibt es z. B. bei Oracle durch die Option INITIALLY IMMEDIATE DEFERRABLE.
- Einige DBMS können Fremdschlüssel-Beziehungen deaktivieren und später wieder aktivieren. Bei einer Aktivierung muss der gesamte Datenbestand der betroffenen Tabelle überprüft werden, und es müssen Anweisungen erteilt werden, wie mit fehlerhaften Sätzen umgegangen werden soll.
- Mit Tools (z. B. Import, Load) kann man bei den meisten DBMS Sätze in eine Tabelle laden, ohne dabei die Fremdschlüssel-Beziehungen zu prüfen.
Bei DB2 z. B. ist die Tabelle danach gesperrt und muss durch das CHECK-Tool geprüft werden. Erst dann steht die Tabelle wieder für reguläre Zugriffe zur Verfügung.
- Wieder andere DBMS lassen rekursive Fremdschlüsselbeziehungen erst gar nicht zu.

28.7. Reihenfolge der Maßnahmen beachten

Wenn die Tabellen in einer Datenbank mit Fremdschlüsseln verbunden sind, dann muss beim Bearbeiten der Tabellen eine bestimmte Reihenfolge eingehalten werden, und zwar sowohl beim Einfügen als auch beim Löschen. Ob auch

das Ändern mit Schwierigkeiten verbunden sein kann, hängt von der Situation ab.

Alle hier genannten Probleme beziehen sich ausschließlich auf diejenigen Spalten, die mit Fremdschlüsseln an andere Tabellen gebunden sind. Änderungen in anderen Spalten können immer ohne Probleme ausgeführt werden.

28.7.1. Bei INSERT

Man muss mit den Tabellen beginnen, die keine Fremdschlüssel haben. Danach können die Tabellen befüllt werden, denen diese Tabellen zugeordnet sind, und so weiter. Bei unserer Beispieldatenbank bedeutet das für einen neuen Versicherungsvertrag:

- Registriere den Versicherungsnehmer.
 - Dazu ist *vorher* ggf. die Versicherungsgesellschaft zu speichern.
- Registriere das Fahrzeug.
 - Dazu ist *vorher* ggf. der Fahrzeugtyp zu speichern
 - und noch einen Schritt *früher* der Fahrzeughersteller.
- Registriere, soweit notwendig, den Mitarbeiter.
 - Dazu ist *vorher* ggf. die Abteilung zu speichern.

Erst jetzt sind alle Voraussetzungen vorhanden, sodass der Vertrag gespeichert werden kann.

Wenn Ring-Verkettungen vorkommen, dann müssen Tools zum initialen Befüllen verwendet werden; oder es muss mit Sätzen begonnen werden, die NULL als Fremdschlüssel enthalten.

28.7.2. Bei DELETE

Zum Löschen von Datensätzen muss – sofern nicht mit einer automatischen Lösch-Weitergabe gearbeitet wird – genau die umgekehrte Reihenfolge eingehalten werden.

28.7.3. Bei UPDATE

Die Fremdschlüssel-Beziehungen verlangen, dass die Datensätze in der zugeordneten Tabelle, auf die aus der Haupttabelle verwiesen wird, vorhanden sein müssen. Wenn man einen Fremdschlüssel in der Haupttabelle ändert, dann

muss der neue Wert ebenfalls in der zugeordneten Tabelle vorhanden sein. Wenn man einen Schlüsselwert in der zugeordneten Tabelle ändern will, dann ist das nur möglich, wenn der alte Wert von keinem Satz in der Haupttabelle verwendet wird. Sollte der Wert doch verwendet (referenziert) werden, dann ist der UPDATE nicht möglich.

Theoretisch wäre auch ein UPDATE-CASCADE vorstellbar. Das würde bedeuten, dass die Änderung eines Wertes in der zugeordneten Tabelle auch gleichzeitig alle referenzierten Werte in der Haupttabelle mitändert. Eine solche Funktion wird jedoch von den meisten Datenbanken nicht angeboten.

Wie geht man also vor, wenn man einen Schlüsselwert in der zugeordneten Tabelle ändern will, der von mehreren Sätzen in der Haupttabelle verwendet wird? Diese Aufgabe wird in drei Schritten gelöst:

- Einen Satz mit dem neuen Wert in der zugeordneten Tabelle einfügen (INSERT)
- Alle Sätze in der Haupttabelle ändern auf den neuen Wert (UPDATE)
- Den Satz mit dem alten Wert in der zugeordneten Tabelle löschen (DELETE)

28.7.4. Bestimme die „Einfüge-Reihenfolge“

Theoretisch handelt es sich hier um ein Problem der „topologischen Sortierung“.

Bei großen Datenmodellen lohnt es sich, sich alle vorhandenen Tabellen in der Reihenfolge zu notieren, in der sie befüllt werden können. Dazu kann man die Struktur der Datenbank auslesen; Voraussetzung ist also, dass die Tabellen bereits erstellt und die Fremdschlüssel definiert sind. Beispiele dazu sind im Kapitel TIPPS UND TRICKS¹ zu finden.

28.8. Zusammenfassung

In diesem Kapitel erfahren wir einiges darüber, wie mit Fremdschlüsseln (= FOREIGN KEYS) die Verbindungen zwischen Tabellen sichergestellt werden:

- Die Definition gehört zur DDL und erfolgt mit einer entsprechenden Einschränkung (CONSTRAINT), meistens über ALTER TABLE.

¹ Kapitel 33 auf Seite 433

- Sowohl beim Einfügen von Datensätzen als auch beim Ändern und Löschen sind diese Beziehungen zu beachten.
- Daraus ergibt sich eine bestimmte Reihenfolge, welche Daten zuerst gespeichert werden müssen.
- Beim Ändern und Löschen können die Anpassungen durch eine Klausel automatisiert werden.

28.9. Übungen

Bei allen Befehlen, die Sie erstellen sollen, benutzen Sie bitte CONSTRAINTS mit Namen.

Übung 1 – Definitionen

Welche der folgenden Aussagen sind wahr, welche sind falsch?

1. Ein Fremdschlüssel legt fest, dass in einer bestimmten Spalte der einen Tabelle nur solche Werte verwendet werden können, die als Primärschlüssel der anderen Tabelle vorhanden sind.
2. Die Tabelle, die den Primärschlüssel enthält, wird als Master-Tabelle bezeichnet, die Tabelle mit dem Fremdschlüssel als Detail-Tabelle.
3. In unserer Beispieldatenbank ist bei der Verknüpfung Fahrzeugtyp/-Fahrzeughersteller die Tabelle *Fahrzeugtyp* der „Master“, die Tabelle *Fahrzeughersteller* die Detail-Tabelle.
4. In diesem Kapitel wird sehr oft bei der Verknüpfung Fahrzeug/Fahrzeugtyp die Tabelle *Fahrzeug* als „Haupttabelle“, die Tabelle *Fahrzeugtyp* als „zugeordnete Tabelle“ bezeichnet.
5. Bei dieser Formulierung entspricht die „Haupttabelle“ den „Details“, die „zugeordnete Tabelle“ dem „Master“.
6. Ein INSERT in der Detail-Tabelle ist immer möglich, ohne die Werte in der Master-Tabelle zu beachten.
7. Ein UPDATE in der Master-Tabelle ist immer möglich, ohne die Werte in der Detail-Tabelle zu beachten.
8. Ein DELETE in der Detail-Tabelle ist immer möglich, ohne die Werte in der Master-Tabelle zu beachten.
9. Ein DELETE in der Master-Tabelle ist immer möglich, ohne die Werte in der Detail-Tabelle zu beachten.
10. Ein FOREIGN KEY wird so definiert: Er wird der Tabelle mit dem Fremdschlüssel und der betreffenden Spalte zugeordnet und verweist auf die Tabelle mit dem Primärschlüssel und der betreffenden Spalte.

11. Ein FOREIGN KEY kann nicht unmittelbar bei der Definition der Fremdschlüssel-Spalte angegeben werden.
12. Eine Fremdschlüssel-Beziehung kann nur zu jeweils einer Spalte der beiden Tabellen gehören.

Übung 2 – Zusammenhänge

Welche Fremdschlüssel sind bei den Tabellen *Fahrzeug* und *Mitarbeiter* der Beispieldatenbank vorzusehen? Nennen Sie jeweils die betreffenden Spalten. Sind bei den „zugeordneten Tabellen“ weitere Fremdschlüssel vorzusehen?

Übung 3 – Fremdschlüssel festlegen

Verknüpfen Sie die Tabelle *Fahrzeugtyp* mit der Tabelle *Fahrzeughersteller* als Fremdschlüssel auf die entsprechenden Spalten.

Übung 4 – Fremdschlüssel festlegen

Verknüpfen Sie die Tabelle *Dienstwagen* mit den Tabellen *Mitarbeiter* und *Fahrzeugtyp* als Fremdschlüssel auf die entsprechenden Spalten.

Übung 5 – Optionen bei Neuaufnahmen

Welche der folgenden Aussagen sind wahr, welche sind falsch?

1. Für Neuaufnahmen gibt es keine Option ON INSERT zur Automatisierung.
2. Bei einer Neuaufnahme in der „Master-Tabelle“ müssen Verknüpfungen in der „Detail-Tabelle“ nicht beachtet werden.
3. Bei einer Neuaufnahme in der „Detail-Tabelle“ müssen Verknüpfungen in der „Master-Tabelle“ nicht beachtet werden.
4. Bei einer Neuaufnahme in der „Master-Tabelle“ kann der erforderliche Datensatz in der „Detail-Tabelle“ automatisch aufgenommen werden.
5. Bei einer Neuaufnahme in der „Detail-Tabelle“ kann der erforderliche Datensatz in der „Master-Tabelle“ automatisch aufgenommen werden.
6. Bei Neuaufnahmen in beiden Tabellen ist die Reihenfolge genau zu beachten.

Übung 6 – Optionen bei Änderungen

Welche der folgenden Aussagen sind wahr, welche sind falsch?

1. Bei einer Änderung in der „Master-Tabelle“ müssen Verknüpfungen in der „Detail-Tabelle“ nicht beachtet werden.
2. Bei einer Änderung in der „Detail-Tabelle“ müssen Verknüpfungen in der „Master-Tabelle“ nicht beachtet werden.
3. Bei einer Änderung in der „Master-Tabelle“ wird der zugehörige Datensatz in der „Detail-Tabelle“ automatisch geändert, sofern ON UPDATE SET DEFAULT festgelegt ist.
4. Bei einer Änderung in der „Master-Tabelle“ wird der zugehörige Datensatz in der „Detail-Tabelle“ automatisch geändert, sofern ON UPDATE CASCADE festgelegt ist.
5. Bei einer Änderung in der „Detail-Tabelle“ wird der zugehörige Datensatz in der „Master-Tabelle“ automatisch geändert, sofern ON UPDATE CASCADE festgelegt ist.
6. Sofern ON UPDATE RESTRICT festgelegt ist, wird eine Änderung in der „Master-Tabelle“ immer ausgeführt.

Übung 8 – Die Reihenfolge

Nennen Sie die Reihenfolge der einzelnen INSERT-Befehle, wenn ein neuer Schadensfall mit drei beteiligten Fahrzeugen gespeichert werden soll. Nehmen Sie dabei an, dass ein Fahrzeug zu einem „Eigenen Kunden“ gehört und zwei Fahrzeuge zu „Fremdkunden“ gehören; die eine Versicherungsgesellschaft der Fremdkunden soll schon gespeichert sein, die andere nicht.

28.10. Lösungen

Lösung zu Übung 1 – Definitionen

Die Aussagen 1, 2, 4, 5, 8, 10 sind wahr. Die Aussagen 3, 6, 7, 9, 11, 12 sind falsch.

Lösung zu Übung 2 – Zusammenhänge

Fahrzeug: *Fahrzeugtyp_ID* verweist auf *ID* der Tabelle *Fahrzeugtyp*.

Fahrzeugtyp: *Hersteller_ID* verweist auf *ID* der Tabelle *Fahrzeughersteller*.

Mitarbeiter: *Abteilung_ID* verweist auf *ID* der Tabelle *Abteilung*.

Lösung zu Übung 3 – Fremdschlüssel festlegen

```
ALTER TABLE Fahrzeugtyp
ADD CONSTRAINT Fahrzeugtyp_FK
FOREIGN KEY (Hersteller_ID)
REFERENCES Fahrzeughersteller (ID);
```

Lösung zu Übung 4 – Fremdschlüssel festlegen

```
ALTER TABLE Dienstwagen
ADD CONSTRAINT Dienstwagen_FZ
FOREIGN KEY (Fahrzeugtyp_ID) REFERENCES Fahrzeugtyp (ID),
ADD CONSTRAINT Dienstwagen_MI
FOREIGN KEY (Mitarbeiter_ID) REFERENCES Mitarbeiter (ID);
```

Lösung zu Übung 5 – Optionen bei Neuaufnahmen

Die Aussagen 1, 2, 6 sind wahr. Die Aussagen 3, 4, 5 sind falsch.

Lösung zu Übung 6 – Optionen bei Änderungen

Die Aussagen 3, 4 sind wahr. Die Aussagen 1, 2, 5, 6 sind falsch.

Lösung zu Übung 8 – Die Reihenfolge

1. die „neue“ Versicherungsgesellschaft speichern
2. deren Kunde speichern
3. dessen Fahrzeug speichern
4. dessen Versicherungsvertrag speichern
5. den Fremdkunden der schon registrierten Versicherungsgesellschaft speichern
6. dessen Fahrzeug speichern
7. dessen Versicherungsvertrag speichern
8. den Schadensfall speichern
9. die Verknüpfungen speichern:
 - a) Schadensfall mit dem eigenen Kunden verknüpfen
 - b) Schadensfall mit dem einen Fremdkunden verknüpfen
 - c) Schadensfall mit dem anderen Fremdkunden verknüpfen

28.11. Siehe auch

Weitere Einzelheiten sind in den folgenden Kapiteln zu finden:

- TABELLENSTRUKTUR DER BEISPIELDATENBANK²
- DATA DEFINITION LANGUAGE (DDL)³
- DDL - EINZELHEITEN⁴ mit Einzelheiten zur Tabellendefinition

Wikipedia bietet verschiedene grundlegenden Informationen:

- FREMDSCHLÜSSEL⁵
- REFERENTIELLE INTEGRITÄT⁶
- ANOMALIEN⁷
- TOPOLOGISCHE SORTIERUNG⁸

2 Kapitel 36 auf Seite 469

3 Kapitel 9 auf Seite 87

4 Kapitel 27 auf Seite 321

5 [HTTP://DE.WIKIPEDIA.ORG/WIKI/FREMDSCHL%FCsSEL](http://de.wikipedia.org/wiki/Fremdschl%FCsSEL)

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/REFERENTIELLE%20INTEGRIT%E4T](http://de.wikipedia.org/wiki/Referentielle%20Integrit%E4T)

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ANOMALIE%20%28INFORMATIK%29](http://de.wikipedia.org/wiki/Anomalie%20%28Informatik%29)

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/TOPOLOGISCHE%20SORTIERUNG](http://de.wikipedia.org/wiki/Topologische%20Sortierung)

29. SQL-Programmierung

Innerhalb einer Datenbank können Arbeitsabläufe selbst gesteuert werden. Dafür gibt es Funktionen, Prozeduren und Trigger.

29.1. Allgemeine Hinweise

Bei diesen Konstruktionen gibt es relativ wenig Gemeinsamkeiten zwischen den DBMS. Für Funktionen und Prozeduren lässt bereits der SQL-Standard den DB-Anbietern „alle“ Freiheiten, wie sie diese Möglichkeiten verwirklichen wollen. Deshalb können auch wir uns nur auf einige Grundlagen beschränken und müssen erneut auf die Dokumentation des jeweiligen DBMS verweisen.

Funktionen und Prozeduren werden oft gemeinsam als **Routinen** bezeichnet.

Diese Elemente benutzen integrierte Funktionen, DML-Befehle und teilweise Datenbank-Operationen, verbunden in einer speziellen Programmiersprache, die **prozedurales SQL** o. ä. bezeichnet wird. In diesem Kapitel gibt es allgemeine Erklärungen dazu, wie solche Abläufe erstellt und programmiert werden können; in den folgenden Kapiteln werden diese Mittel konkret benutzt.

29.1.1. Funktionen

Eine (benutzerdefinierte Skalar-) Funktion liefert genau einen Wert eines bestimmten Datentyps. Es handelt sich dabei um eine Ergänzung zu den internen Skalarfunktionen des DBMS. Es gibt sie mit und ohne Argumente; sie werden gezielt vom Anwender bzw. einem Anwendungsprogramm aufgerufen.

- Einzelheiten dazu werden in EIGENE FUNKTIONEN¹ behandelt.

1 Kapitel 30 auf Seite 389

29.1.2. Prozeduren

Eine Prozedur – gespeicherte Prozedur, engl. **StoredProcedure** (SP) – ist vorgesehen für „immer wiederkehrende“ Arbeitsabläufe. Es gibt sie mit und ohne Argumente und Rückgabewerte; sie werden gezielt vom Anwender bzw. einem Anwendungsprogramm aufgerufen.

- Einzelheiten dazu werden in PROZEDUREN² behandelt.

29.1.3. Trigger

Ein Trigger ist ein Arbeitsablauf, der automatisch beim Speichern in einer Tabelle ausgeführt wird. Es gibt weder Argumente noch Rückgabewerte und keinerlei direkte Zusammenarbeit mit dem Anwender bzw. einem Anwendungsprogramm.

- Einzelheiten dazu werden in TRIGGER³ behandelt.

29.2. Routinen ohne feste Speicherung

Das, was als Prozedur gespeichert werden kann, kann in einem DBMS in der Regel auch direkt ausgeführt werden (ohne Speicherung in der Datenbank). Dazu werden die Definition von Parametern und Variablen sowie die Anweisungen mit einer EXECUTE-Anweisung aufgerufen.

Im Kapitel zu Prozeduren gibt es ein Beispiel „Testdaten in einer Tabelle erzeugen“, das auch so verwirklicht werden kann:

Firebird Quelltext

```
EXECUTE BLOCK ( Anzahl INT = ?anzahl )
    RETURNS ( Maxid INT )
AS
    DECLARE VARIABLE Temp INT = 0;      /* usw. identisch wie bei der
    Prozedur */
BEGIN
    Maxid = 0;
    WHILE (Temp < Anzahl) DO
    BEGIN
        /* identischer Arbeitsablauf wie bei der Prozedur */
```

2 Kapitel 31 auf Seite 399

3 Kapitel 32 auf Seite 421


```

Temp = Temp + 1;
END
select MAX(ID) from Fahrzeug into :Maxid;
SUSPEND;
END

```

Der Aufbau entspricht dem einer Prozedur (siehe unten). Der Unterschied besteht in der direkten Ausführung durch EXECUTE BLOCK.

29.3. Programmieren innerhalb von Routinen

Bitte haben Sie Nachsicht: Wegen der vielen Varianten bei den DBMS wurde ein Teil der folgenden Hinweise und der Beispiele in den nächsten Kapiteln nur nach der Dokumentation verfasst und nicht in der Praxis umgesetzt.

29.3.1. Allgemeines

Routinen – also Funktionen und Prozeduren – werden grundsätzlich mit einer **Syntax** ähnlich der folgenden definiert:

```

CREATE OR ALTER { FUNCTION | PROCEDURE } <routine-name>
    ( [ <parameterliste> ] )
    RETURNS <parameterliste>
AS
BEGIN
    <variablenliste>
    <anweisungen>
END

```

Die Definition von Triggern verläuft so ähnlich: Parameter entfallen, aber die Art der Auslösung kommt hinzu. Die Hinweise zu Variablen und Anweisungen in den folgenden Abschnitten gelten für Trigger in gleicher Weise wie für Routinen.

Der Teil zwischen BEGIN und END (jeweils einschließlich) wird als *Rumpf* – engl. *body* – bezeichnet, alles davor heißt *Kopf* – engl. *header* – der Routine.

Bitte beachten Sie, dass jedes DBMS seine eigenen Besonderheiten hat. Die wichtigsten Unterschiede sind:

- Bei MySQL müssen CREATE und ALTER getrennt werden, bei Oracle heißt es CREATE OR REPLACE.
- RETURNS gehört zu Funktionen (bei Oracle: RETURN). Nur Firebird benutzt dies zur Trennung der Ausgabe-Parameter auch bei Prozeduren.

- Ob die Parameter in Klammern stehen müssen oder nicht, ist unterschiedlich geregelt.
- AS kann teilweise auch entfallen, bei Oracle wird auch IS verwendet.
- Die <variablenliste> ist Bestandteil der <anweisungen>; bei Firebird und Oracle steht sie zwischen AS und BEGIN.

Wenn es insgesamt (einschließlich Variablen) nur eine einzige Anweisung gibt, kann auf BEGIN und END verzichtet werden; der Übersichtlichkeit halber ist ihre Verwendung aber fast immer zu empfehlen.

Gleiches gilt innerhalb einzelner Abschnitte (wie Verzweigungen oder Schleifen): Eine einzelne Anweisung kann ohne BEGIN...END angegeben werden; wenn es die Übersichtlichkeit oder Verschachtelung erfordern, ist die Verwendung dieser Schlüsselwörter vorzuziehen.

Eine Funktion benötigt als (letzte) Anweisung RETURN, mit der ein bestimmter Wert zurückgegeben wird.

Bei Verzweigungen und Schleifen kann durch LABELs der Zusammenhang deutlich gemacht werden. Meistens gibt es keine Notwendigkeit dazu, sodass wir darauf verzichten.

Hinweis: Dieser Abschnitt beschränkt sich auf die wichtigsten Erläuterungen. Die konkreten SQL-Anweisungen sind in den folgenden Kapiteln zu finden. Außerdem gibt es zu fast allen genannten Themen weitere Möglichkeiten.

29.3.2. Spalten, Variable und Parameter

In diesem Buch werden nur einfache lokale Variable benutzt; deren Gültigkeitsbereich beschränkt sich auf die aktuelle Routine. Je nach DBMS stehen auch globale Variable zur Verfügung. Außerdem kann man über das Schlüsselwort CURSOR eine ganze Zeile von Tabellen oder Ergebnismengen mit einer Variablen benutzen.

In allen Fällen, in denen die Namen von Variablen oder Parametern auf die Namen von Tabellenspalten treffen, muss dem DBMS klar sein, um welche Art von Namen es sich handelt:

- MS-SQL regelt das mit '@' am Anfang des Namens von Variablen oder Parametern.
- MySQL und Oracle unterscheiden nicht. Sie müssen selbst für unterschiedliche Bezeichner sorgen.

- Firebird verlangt in diesen Situationen einen Doppelpunkt vor dem Namen von Variablen und Parametern.

Wegen der vielfältigen Unterschiede werden die wichtigsten Möglichkeiten getrennt behandelt.

29.4. SQL-Programmierung mit Firebird

Einzelheiten

Parameter deklarieren

Jeder Parameter, der innerhalb der Anweisungen benutzt wird und dessen Wert an die Routine übergeben oder durch die Bearbeitung zurückgegeben wird, muss im Kopf der Routine festgelegt werden: Name, Datentyp, Vorgabe- oder Anfangswert. Mehrere Parameter werden mit Komma verbunden; nach dem letzten Parameter fehlt es.

Bei Funktionen kann es nur Eingabe-Parameter geben; der Ausgabe-Parameter wird durch RETURNS immer getrennt angegeben.

Eingabe-Parameter stehen nach dem Namen der Routine vor der RETURNS-Klausel, Ausgabe-Parameter sind Teil der RETURNS-Klausel. Als Datentypen sind ab Version 2.1 auch DOMAINS zulässig. Ein einzelner Parameter wird so **deklariert**: `<name> <typ> [{= | DEFAULT} <wert>]`

Bei Eingabe-Parametern sind auch Vorgabewerte möglich, die durch '=' oder DEFAULT gekennzeichnet werden. Wichtig ist: Wenn ein Parameter einen Vorgabewert erhält und deshalb beim Aufruf in der Liste nicht benutzt wird, müssen alle nachfolgenden Parameter ebenfalls mit Vorgabewert arbeiten.

Variable deklarieren

Jede Variable, die innerhalb der Anweisungen benutzt wird, muss im Kopf der Routine festgelegt werden, nämlich zwischen AS und BEGIN: Name, Datentyp, Vorgabe- oder Anfangswert. Jede Deklaration gilt als eine einzelne Anweisung und ist mit Semikolon abzuschließen.

```
DECLARE [VARIABLE] <name> <typ> [ {=|DEFAULT} <wert> ];
```

Als Vorgabewert ist auch ein SQL-Ausdruck möglich.

Zuweisungen von Werten zu Variablen und Parametern

Der einfachste Weg ist die **direkte Zuweisung** eines Wertes oder eines Ausdrucks (einer internen oder einer eigenen Funktion) zu einer Variablen oder einem Parameter:

```
<name> = <ausdruck> ;      /* Standard: nur das Gleichheitszeichen
*/
```

Sehr oft werden die Werte aus einem **SELECT-Befehl** mit Variablen weiterverarbeitet. Dazu gibt es die INTO-Klausel:

```
SELECT <spaltenliste>
FROM <usw. alles andere>
INTO <variablenliste> ;
```

Die Liste der Variablen muss von Anzahl und Typ her der Liste der Spalten entsprechen. Bitte beachten Sie, dass bei Firebird die INTO-Klausel erst am Ende des Befehls stehen darf.

In ähnlicher Weise kann auch das **Ergebnis einer Prozedur** übernommen und in der aktuellen Routine verarbeitet werden:

```
EXECUTE PROCEDURE <routine-name> [ <eingabe-parameter> ]
RETURNING_VALUES <variablenliste> ;      /* Variablen mit
Doppelpunkt */
```

Jede hier genannte Variable muss (in Reihenfolge und Typ) einem der <ausgabe-parameter> der Prozedur entsprechen.

Achtung: In allen Fällen, in denen die Namen von Variablen oder Parametern auf die Namen von Tabellenspalten treffen, muss dem DBMS klar sein, um welche Art von Namen es sich handelt. Firebird verlangt in diesen Situationen einen **Doppelpunkt** vor dem Namen von Variablen und Parametern.

Erst die **SUSPEND**-Anweisung sorgt dafür, dass ein Ausgabe-Parameter vom „rufenden“ Programm entgegengenommen werden kann. Bei einer Rückgabe einfacher Werte steht diese Anweisung am Ende einer Prozedur; bei einer Funktion übernimmt RETURN diese Aufgabe. Wenn aber (wie durch einen SELECT-Befehl) mehrere Zeilen zu übergeben sind, muss SUSPEND bei jeder dieser

Zeilen stehen. Ein Beispiel steht im Kapitel zu Prozeduren unter *Ersatz für eine View mit Parametern*.

Verzweigungen

Die **IF**-Abfrage steuert den Ablauf nach Bedingungen:

```
IF ( <bedingung> ) THEN
BEGIN
    <anweisungen>
END
[ ELSE
BEGIN
    <anweisungen>
END
]
```

Diese Abfrage sieht so aus:

- Die <bedingung> muss in Klammern stehen; sie kann auch mit AND und OR sowie weiteren Klammern verschachtelt werden.
- Der ELSE-Zweig ist optional; die IF-Abfrage kann auch auf den IF-THEN-Abschnitt beschränkt werden.
- Der ELSE-Zweig kann durch weitere IF-Abfragen verschachtelt werden.
- Vor allem bei verschachtelten Abfragen sollten BEGIN...END immer benutzt und durch Einrückungen der Zusammenhang deutlich werden.

Schleifen

Es gibt zwei Arten von Schleifen: eine Schleife mit einer Bedingung und eine Schleife mit einer Ergebnismenge für eine SELECT-Abfrage.

Die **WHILE**-Schleife prüft eine Bedingung und wird so lange durchlaufen, wie diese Bedingung wahr ist:

```
WHILE ( <bedingung> ) DO
BEGIN
    <anweisungen>
END
```

Diese Schleife sieht so aus:

- Die <bedingung> muss in Klammern stehen; sie kann auch mit AND und OR sowie weiteren Klammern verschachtelt werden.
- Die <bedingung> wird jeweils am Anfang eines Durchgangs geprüft. Wenn ihr Wert von Anfang an FALSE ist, wird die Schleife überhaupt nicht durchlaufen.

Die **FOR SELECT**-Schleife erstellt durch einen SELECT-Befehl eine Ergebnismenge und führt für jede Zeile des Ergebnisses etwas aus:

```
FOR SELECT <abfrage-einzelheiten>
        INTO <variablenliste>
DO BEGIN
    <anweisungen>
END
```

Diese Schleife sieht so aus:

- Beim SELECT-Befehl handelt es sich um eine beliebige Abfrage.
- Für jede Zeile des Ergebnisses wird der DO-Block einmal durchlaufen.
- Die Ergebnisspalten und ihre Werte sind nur innerhalb des SELECT-Befehls bekannt, nicht innerhalb der DO-Anweisungen. Die Werte müssen deshalb zunächst in Variablen übergeben (für jede Spalte eine Variable oder Ausgabe-Parameter, mit Doppelpunkt gekennzeichnet), bevor sie in den <anweisungen> benutzt werden können.
- Wenn diese Werte für jede Zeile einzeln zurückgegeben werden sollen, wird SUSPEND als eine Anweisung benötigt.
- Wenn SUSPEND die einzige Anweisung ist und kein Einzelwert später (außerhalb der Schleife) noch benötigt wird, kann auf die INTO-Klausel verzichtet werden.

Zulässige Befehle

Innerhalb von Routinen sind ausschließlich DML-Befehle zulässig. Keinerlei anderer Befehl wird akzeptiert, also vor allem kein DDL-Befehl, aber auch nicht GRANT/REVOKE (DCL) oder COMMIT/ROLLBACK (TCL).

Sämtliche DML-Befehle, die innerhalb einer Routine ausgeführt werden, gehören zur selben Transaktion wie die Befehle, durch die sie aufgerufen bzw. ausgelöst werden.

Anweisungen begrenzen

Jede einzelne Anweisung innerhalb einer Routine und jeder SQL-Befehl müssen mit einem Semikolon abgeschlossen werden. Das ist kein Problem, wenn nur ein einzelner CREATE PROCEDURE o. ä. ausgeführt werden soll; dann wird das abschließende Semikolon weggelassen, und es gibt keine Unklarheiten (siehe ein Firebird-Beispiel im Trigger-Kapitel).

Wenn – wie im Skript zur Beispieldatenbank – mehrere Routinen nacheinander erzeugt werden, muss das DBMS zwischen den verschiedenen Arten von Abschlusszeichen unterscheiden. Dazu dient der **SET TERM**-Befehl (TERM steht für *Terminator*, also *Begrenzer*):

```
Firebird Quelltext
```

```
SET TERM ^ ;

CREATE OR ALTER TRIGGER Abteilung_BIO FOR Abteilung
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF ((new.ID IS NULL) OR (new.ID = 0))
  THEN new.ID = NEXT VALUE FOR Abteilung_ID;
END
^

SET TERM ; ^
```

Zuerst wird der Begrenzer für SQL-Befehle auf '^' geändert; das Abschlusszeichen für einzelne Anweisungen bleibt das Semikolon. Dann folgen alle Befehle zur Trigger-Definition; jeder einzelne Trigger wird mit dem neuen Begrenzer beendet. Abschließend wird der Begrenzer wieder auf das Semikolon zurückgesetzt.

29.5. SQL-Programmierung mit MS-SQL

Einzelheiten

Als erste Anweisung einer Routine sollte immer `SET NOCOUNT ON;` verwendet werden; dies beschleunigt die Ausführung.

Parameter deklarieren

Jeder Parameter, der innerhalb der Anweisungen benutzt wird und dessen Wert an die Routine übergeben oder durch die Bearbeitung zurückgegeben wird, muss im Kopf der Routine festgelegt werden: Name, Datentyp, Verwendung für Eingabe und/oder Ausgabe, Vorgabe- oder Anfangswert. Mehrere Parameter werden mit Komma verbunden; nach dem letzten Parameter fehlt dies.

Bei Funktionen kann es nur Eingabe-Parameter geben; der Ausgabe-Parameter wird durch RETURNS immer getrennt angegeben.

Ein einzelner Parameter wird so deklariert:

```
<name> <typ> [ = <wert> ] [ OUT {{{}} OUTPUT ]
```

Parameternamen müssen immer mit '@' beginnen. Ausgabe-Parameter werden mit OUT bzw. OUTPUT markiert; alle anderen sind Eingabe-Parameter. Durch das Gleichheitszeichen kann ein Vorgabewert zugewiesen werden.

Variablen deklarieren

Jede Variable, die innerhalb der Anweisungen benutzt wird, muss im Rumpf der Routine festgelegt werden: Name, Datentyp, Vorgabe- oder Anfangswert. Jede Deklaration gilt als eine einzelne Anweisung und ist mit Semikolon abzuschließen.

```
DECLARE <name> [AS] <typ> [ = <wert> ];
```

Bei MS-SQL muss der Name immer mit '@' beginnen. Als Vorgabewert ist auch ein SQL-Ausdruck möglich.

Zuweisungen von Werten zu Variablen und Parametern

Der einfachste Weg ist die **direkte Zuweisung** eines Wertes oder eines Ausdrucks (einer internen oder einer eigenen Funktion) zu einer Variablen oder einem Parameter mit dem SET-Befehl:


```
SET <name> = <ausdruck> ;
```

Sehr oft werden die Werte aus einem **SELECT-Befehl** mit Variablen weiterverarbeitet. Ein einzelner Wert wie das Ergebnis von SELECT COUNT(*) wird ebenfalls durch SET der Variablen zugewiesen. Mehrere Werte können innerhalb des SELECT direkt zugewiesen werden:

```
SELECT @Variable1 = <spalte1>, @Variable2 = <spalte2>
FROM <tabellenliste> /* usw. weitere Bedingungen */
```

Um eine Ergebnismenge zu nutzen, wird ein **CURSOR** benötigt, bei dem mit FETCH ein Datensatz geholt und der Wert einer Spalte mit INTO einer Variablen übergeben wird.

Eine Routine wird meistens mit EXECUTE ausgeführt. **Rückgabewerte**, die über OUTPUT-Parameter bestimmt werden, werden vorher deklariert und mit dem <ausgabe-parameter> der Prozedur verbunden.

```
declare @inputvariable varchar(25);
set @inputvariable = 'Meier';
declare @outputvariable MONEY;
execute myprocedure @inputparameter = @inputvariable, @outputparameter
= @outputvariable OUTPUT;
select @outputvariable;
```

Jede hier genannte Variable muss (in Reihenfolge und Typ) allen Parametern der Prozedur entsprechen.

Für EXECUTE (auch die Abkürzung EXEC ist möglich) gibt es viele Varianten für Aufruf und Zuweisung der Parameter.

Verzweigungen

Die IF-Abfrage steuert den Ablauf nach Bedingungen:

```
IF <bedingung>
BEGIN
  <anweisungen>
END
[ ELSE IF <bedingung>
  BEGIN
    <anweisungen>
  END ]
[ ELSE
  BEGIN
    <anweisungen>
  END ]
END
```

Diese Abfrage sieht so aus:

- Bei <bedingung> handelt es sich um eine einfache Prüfung, die nicht mit AND oder OR erweitert werden kann.
- Wenn ein SELECT-Befehl Teil der <bedingung> ist, muss er in Klammern stehen.
- Der ELSE-Zweig ist optional. Es ist auch möglich, dass nur der IF-THEN-Abschnitt ausgeführt werden muss.
- Durch ELSE IF-Zweige sind Verschachtelungen, also auch weitere Prüfungen möglich.
- Vor allem bei verschachtelten Abfragen sollten BEGIN...END immer benutzt und durch Einrückungen der Zusammenhang deutlich gemacht werden.

Schleifen

Die **WHILE**-Schleife prüft eine Bedingung und wird so lange durchlaufen, wie diese Bedingung wahr ist:

```
WHILE <bedingung>
BEGIN <anweisungen> END
```

Dabei sind folgende Punkte wichtig:

- Bei <bedingung> handelt es sich um eine einfache Prüfung, die nicht mit AND oder OR erweitert werden kann.
- Wenn ein SELECT-Befehl Teil der <bedingung> ist, muss er in Klammern stehen.

- Die <bedingung> wird jeweils am Anfang eines Durchgangs geprüft. Wenn ihr Wert von Anfang an FALSE ist, wird die Schleife überhaupt nicht durchlaufen.
- Schleifen können verschachtelt werden. Vor allem dann sollten BEGIN...END immer benutzt und durch Einrückungen der Zusammenhang deutlich gemacht werden.

Eine Schleife oder ein IF-Zweig kann mit BREAK vorzeitig abgebrochen; mit CONTINUE kann direkt der nächste Durchlauf begonnen werden. Bei Verschachtelung wird mit BREAK zur nächsthöheren Ebene gesprungen.

Um alle Datensätze einer Ergebnismenge, also eines SELECT-Befehls zu durchlaufen, wird ein **CURSOR** benötigt. Dies wird in dieser *Einführung* nicht behandelt.

Zulässige Befehle

Einige Anweisungen können nicht in einer gespeicherten Prozedur verwendet werden. Dazu gehören vor allem der Wechsel der aktuellen Datenbank durch USE sowie CREATE/ALTER für Views, Funktionen, Prozeduren und Trigger. Im Gegensatz zu anderen DBMS ist aber z. B. CREATE TABLE möglich; auch COMMIT bzw. ROLLBACK sind unter Umständen möglich.

Anweisungen begrenzen

Bei MS-SQL gibt es keine Notwendigkeit, zwischen Anweisungen innerhalb einer Routine und getrennten SQL-Befehlen zu unterscheiden: Jede Anweisung wird mit Semikolon abgeschlossen; ein „selbständiger“ SQL-Befehl wird durch GO abgeschlossen und ausgeführt.

29.6. SQL-Programmierung mit MySQL

Einzelheiten

MySQL hat gespeicherte Prozeduren und Funktionen erst mit Version 5 eingeführt, sodass noch nicht alle Wünsche erfüllt werden. Aber das wird natürlich von Version zu Version besser.

Bitte achten Sie unbedingt darauf, dass sich die Namen von Parametern und Variablen von Spaltennamen unterscheiden, die in derselben Anweisung vorkommen. Wenn Sie gegen diese Bedingung verstoßen, gibt es unvorhersehbare Ergebnisse, weil die Namen der Variablen Vorrang haben gegenüber gleichnamigen Spalten. Ein gängiges Verfahren ist es (wie bei MS-SQL), dass diese Namen mit '@' beginnen.

Parameter deklarieren

Jeder Parameter, der innerhalb der Anweisungen benutzt wird und dessen Wert an die Routine übergeben oder durch die Bearbeitung zurückgegeben wird, muss im Kopf der Routine festgelegt werden: Name, Datentyp, Verwendung für Eingabe und/oder Ausgabe, Vorgabe- oder Anfangswert. Mehrere Parameter werden mit Komma verbunden; nach dem letzten Parameter fehlt dies.

Bei Funktionen kann es nur Eingabe-Parameter geben; der Ausgabe-Parameter wird durch RETURNS immer getrennt angegeben.

Ein einzelner Parameter wird so deklariert:

```
[ IN {} OUT {} INOUT ] <name> <typ>
```

Die Festlegung als Eingabe-Parameter kann entfallen; IN ist der Standardwert.

Variablen deklarieren

Jede Variable, die innerhalb der Anweisungen benutzt wird, muss im Rumpf der Routine festgelegt werden: Name, Datentyp, Vorgabe- oder Anfangswert. Jede Deklaration gilt als eine einzelne Anweisung und ist mit Semikolon abzuschließen.

```
DECLARE <name> <typ> [ DEFAULT <wert> ];
```

Als Vorgabewert ist auch ein SQL-Ausdruck möglich.

Zuweisungen von Werten zu Variablen und Parameter

Der einfachste Weg ist die **direkte Zuweisung** eines Wertes oder eines Ausdrucks (einer internen oder einer eigenen Funktion) zu einer Variablen oder einem Parameter mit dem SET-Befehl:

```
SET <name> = <ausdruck> ;
```

Sehr oft werden die Werte aus einem **SELECT-Befehl** mit Variablen weiterverarbeitet. Dazu gibt es die INTO-Klausel, die direkt nach der <spaltenliste> kommt. Die Liste der Variablen muss von Anzahl und Typ her der Liste der Spalten entsprechen.

```
SELECT <spaltenliste> INTO <variablenliste>  
FROM <usw. alles andere> ;
```

Eine Prozedur wird mit CALL ausgeführt. **Rückgabewerte**, die über OUT-Parameter bestimmt werden, werden vorher deklariert.

```
declare @inputvariable varchar(25);  
SET @inputvariable = 'Meier';  
declare @outputvariable MONEY = 0;  
CALL myprocedure (@inputvariable, @outputvariable);  
SELECT @outputvariable;
```

Jede hier genannte Variable muss (in Reihenfolge und Typ) einem der <ausgabeparameter> der Prozedur entsprechen.

Verzweigungen

MySQL kennt zwei Arten, um auf unterschiedliche Bedingungen zu reagieren:

Die IF-Abfrage steuert den Ablauf nach Bedingungen:

```

IF <bedingung> THEN
BEGIN
  <anweisungen>
END
[ ELSEIF <bedingung> THEN
  BEGIN
    <anweisungen>
  END ]
[ ELSE
  BEGIN
    <anweisungen>
  END ]
END IF

```

Diese Abfrage sieht so aus:

- Bei <bedingung> handelt es sich um eine einfache Prüfung, die nicht mit AND oder OR erweitert werden kann.
- Der ELSE-Zweig ist optional. Es ist auch möglich, dass nur der IF-THEN-Abschnitt ausgeführt werden muss.
- Durch ELSEIF-Zweige sind Verschachtelungen, also auch weitere Prüfungen möglich.
- Vor allem bei verschachtelten Abfragen sollten BEGIN...END immer benutzt und durch Einrückungen der Zusammenhang deutlich gemacht werden.

Die **CASE**-Prüfung entspricht der Fallunterscheidung aus dem Kapitel *Nützliche Erweiterungen*:

```

-- Variante 1
CASE <ausdruck>
  WHEN <ausdruck 1> THEN BEGIN <anweisungen 1> END
  [ WHEN <ausdruck 1> THEN BEGIN <anweisungen 2> END ] /* usw. */
  [ ELSE
    BEGIN <anweisungen n> END ]
END CASE
-- Variante 2
CASE
  WHEN <bedingung 1> THEN BEGIN <anweisungen 1> END
  [ WHEN <bedingung 2> THEN BEGIN <anweisungen 2> END ] /* usw. */
  [ ELSE
    BEGIN <anweisungen n> END ]
END CASE

```

Bei dieser Prüfung gelten die gleichen Prüfungen wie bei der Fallunterscheidung mit folgenden Abweichungen:

- Im ELSE-Zweig darf es keine NULL-Anweisung geben.
- Die Verzweigung muss mit END CASE abgeschlossen werden.

Schleifen

MySQL kennt mehrere Arten von Schleifen, aber **keine FOR**-Schleife. Wenn ITERATE oder LEAVE verwendet werden, müssen LABELs gesetzt werden.

Die **LOOP**-Schleife arbeitet ohne Bedingung am Anfang oder Ende und wird solange durchlaufen, bis – vor allem aufgrund einer „inneren“ Bedingung – die LEAVE-Anweisung getroffen wird. Mit ITERATE wird direkt der nächste Durchgang gestartet.

```
LOOP
BEGIN <anweisungen> END
END LOOP ;
```

Die **REPEAT**-Schleife wird mindestens einmal durchlaufen, und zwar solange, bis die Ende-Bedingung FALSE ergibt oder bis – aufgrund einer „inneren“ Bedingung – die LEAVE-Anweisung getroffen wird.

```
REPEAT
BEGIN <anweisungen> END
UNTIL <bedingung>
END REPEAT
```

Die **WHILE**-Schleife prüft eine Bedingung und wird so lange durchlaufen, wie diese Bedingung wahr ist:

```
WHILE <bedingung> DO
BEGIN <anweisungen> END
END WHILE
```

Die <bedingung> wird jeweils am Anfang eines Durchgangs geprüft. Wenn ihr Wert von Anfang an FALSE ist, wird die Schleife überhaupt nicht durchlaufen.

Um alle Datensätze einer Ergebnismenge, also eines SELECT-Befehls zu durchlaufen, wird ein **CURSOR** benötigt. Dies wird in dieser „Einführung“ nicht behandelt.

Zulässige Befehle

Innerhalb von eigenen Funktionen, Prozeduren und Triggern sind nicht alle SQL-Befehle zulässig. Bitte lesen Sie in Ihrer Dokumentation Einzelheiten nach.

Anweisungen begrenzen

Jede einzelne Anweisung innerhalb einer Routine und jeder SQL-Befehl müssen mit einem Semikolon abgeschlossen werden. Das ist kein Problem, wenn nur ein einzelner CREATE PROCEDURE o. ä. ausgeführt werden soll; dann wird das abschließende Semikolon weggelassen, und es gibt keine Unklarheiten (siehe ein MySQL-Beispiel zu *Trigger*).

Wenn mehrere Routinen nacheinander erzeugt werden, muss das DBMS zwischen den verschiedenen Arten von Abschlusszeichen unterscheiden. Dazu dient der **delimiter**-Befehl:

```
MySQL Quelltext
```

```
delimiter //

CREATE OR ALTER TRIGGER Mitarbeiter_BD
  ACTIVE BEFORE DELETE ON Mitarbeiter
FOR EACH ROW
BEGIN
  UPDATE Dienstwagen
    SET Mitarbeiter_ID = NULL
    WHERE Mitarbeiter_ID = old.ID;
END
//

delimiter ;
```

Zuerst wird der Begrenzer für SQL-Befehle auf `'''` geändert; das Abschlusszeichen für einzelne Anweisungen bleibt das Semikolon. Dann folgen alle Befehle zur Trigger-Definition; jeder einzelne wird mit dem neuen Begrenzer beendet. Abschließend wird der Begrenzer wieder auf das Semikolon zurückgesetzt.

29.7. SQL-Programmierung mit Oracle

Einzelheiten

Bei Oracle gilt das prozedurale SQL nicht als Teil des DBMS, sondern als spezielle Programmiersprache **PL/SQL**. In der Praxis werden beide Teile gemischt verwendet.

Bitte achten Sie unbedingt darauf, dass sich die Namen von Parametern und Variablen von Spaltennamen unterscheiden, die in derselben Anweisung vorkommen.

Parameter deklarieren

Jeder Parameter, der innerhalb der Anweisungen benutzt wird und dessen Wert an die Routine übergeben oder durch die Bearbeitung zurückgegeben wird, muss im Kopf der Routine festgelegt werden: Name, Datentyp, Verwendung für Eingabe und/oder Ausgabe, Vorgabe- oder Anfangswert. Mehrere Parameter werden mit Komma verbunden; nach dem letzten Parameter fehlt dies.

Bei Funktionen kann es nur Eingabe-Parameter geben; der Ausgabe-Parameter wird durch RETURN (anders als bei anderen DBMS nicht durch RETURNS) immer getrennt angegeben.

Ein einzelner Parameter wird so deklariert:

```
<name> [ IN {{!}} OUT {{!}} IN OUT ] <typ>
```

Die Festlegung als Eingabe-Parameter kann entfallen; IN ist der Standardwert. Beim Datentyp darf die Größe nicht angegeben werden: VARCHAR2 ist zulässig, aber VARCHAR2(20) nicht.

Variablen deklarieren

Jede Variable, die innerhalb der Anweisungen benutzt wird, muss im Kopf der Routine festgelegt werden, nämlich zwischen AS/IS und BEGIN: Name, Datentyp und ggf. Vorgabewert; das Schlüsselwort DECLARE entfällt. Jede Deklaration gilt als eine einzelne Anweisung und ist mit Semikolon abzuschließen.

```
<name> <typ> [ := <ausdruck> ] ;
```

Als Vorgabewert ist auch ein SQL-Ausdruck möglich.

Zuweisungen von Werten zu Variablen und Parameter

Der einfachste Weg ist die **direkte Zuweisung** eines Wertes oder eines Ausdrucks (einer internen oder einer eigenen Funktion) zu einer Variablen oder einem Parameter:

```
<name> := <ausdruck> ;
```

Bitte beachten Sie, dass (wie bei Pascal) Doppelpunkt und Gleichheitszeichen zu verwenden sind.

Sehr oft werden die Werte aus einem **SELECT-Befehl** mit Variablen weiterverarbeitet. Dazu gibt es die INTO-Klausel, die direkt nach der <spaltenliste> kommt:

```
SELECT <spaltenliste> INTO <variablenliste>  
FROM <usw. alles andere> ;
```

Die Liste der Variablen muss von Anzahl und Typ her der Liste der Spalten entsprechen.

In ähnlicher Weise kann auch das **Ergebnis einer Prozedur** übernommen und in der aktuellen Routine verarbeitet werden:

1. Deklaration der Übergabevariablen: <variablenname> <typ>;
oder unter Bezug auf den Typ einer definierten Variablen:
<variablenname> <package_name.typ_name>;
2. Aufruf der Prozedur: <prozedur_name> (<Eingabe_Parameter>,
<Ausgabe_Parameter>);
3. Weiterarbeiten mit den Ausgabeparametern aus der Prozedur

Jede hier genannte Variable muss (in Reihenfolge und Typ) einem der <parameter> der Prozedur entsprechen.

Verzweigungen

Oracle kennt zwei Arten, um auf unterschiedliche Bedingungen zu reagieren:

Die **IF-Abfrage** steuert den Ablauf nach Bedingungen:

```
IF <bedingung> THEN
BEGIN <anweisungen> END
[ ELSE
  BEGIN <anweisungen> END
]
END IF
```

Diese Abfrage sieht so aus:

- Bei <bedingung> handelt es sich um eine einfache Prüfung, die nicht mit AND oder OR erweitert werden kann.
- Der ELSE-Zweig ist optional. Es ist auch möglich, dass nur der IF-THEN-Abschnitt ausgeführt werden muss.
- Durch ELSIF-Zweige sind auch Verschachtelungen möglich.
- Vor allem bei verschachtelten Abfragen sollten BEGIN...END immer benutzt und durch Einrückungen der Zusammenhang deutlich gemacht werden.

Die **CASE-Prüfung** entspricht der Fallunterscheidung aus dem Kapitel *Nützlichen Erweiterungen*:

```
-- Variante 1
CASE <ausdruck>
  WHEN <ausdruck 1> THEN BEGIN <anweisungen 1> END
  [ WHEN <ausdruck 1> THEN BEGIN <anweisungen 2> END ] /* usw. */
  [ ELSE
    BEGIN <anweisungen n> END ]
END CASE
-- Variante 2
CASE
  WHEN <bedingung 1> THEN BEGIN <anweisungen 1> END
  [ WHEN <bedingung 2> THEN BEGIN <anweisungen 2> END ] /* usw. */
  [ ELSE
    BEGIN <anweisungen n> END ]
END CASE
```

Bei dieser Prüfung gelten die gleichen Prüfungen wie bei der Fallunterscheidung mit folgenden Abweichungen:

- Im ELSE-Zweig darf es keine NULL-Anweisung geben.
- Die Verzweigung muss mit END CASE abgeschlossen werden.

Schleifen

Oracle kennt mehrere Arten von Schleifen.

Die **LOOP**-Schleife arbeitet ohne Bedingung am Anfang oder Ende und wird solange durchlaufen, bis – vor allem aufgrund einer „inneren“ Bedingung – die EXIT-Anweisung getroffen wird.

```
LOOP
BEGIN <anweisungen> END
END LOOP ;
```

Oracle Quelltext

```
DECLARE x NUMBER := 1;
BEGIN
  LOOP
    X := X + 1;
    EXIT WHEN x > 10;
  END LOOP;
END;
```

Die **WHILE**-Schleife prüft eine Bedingung und wird so lange durchlaufen, wie diese Bedingung wahr ist:

```
WHILE <bedingung> LOOP
BEGIN <anweisungen> END
END LOOP;
```

Die <bedingung> wird jeweils am Anfang eines Durchgangs geprüft. Wenn ihr Wert von Anfang an FALSE ist, wird die Schleife überhaupt nicht durchlaufen.

Die **FOR**-Schleife durchläuft eine vorgegebene Liste von Werten von Anfang bis Ende:

```
FOR <variable> IN <werteliste> LOOP
BEGIN <anweisungen> END
END LOOP;
```

Bei der FOR-Schleife wird der Wert der „Laufvariablen“ am Beginn jedes weiteren Durchlaufs automatisch erhöht; bei LOOP und WHILE müssen Sie sich selbst um die Änderung einer solchen Variablen kümmern.

Um alle Datensätze einer Ergebnismenge, also eines SELECT-Befehls zu durchlaufen, wird ein **CURSOR** benötigt. Dies wird in dieser „Einführung“ nicht behandelt.

Zulässige Befehle

Neben Anweisungen und den DML-Befehlen sind auch TCL-Befehle (Steuerung von Transaktionen) sowie die Sperre von Tabellen durch LOCK TABLE möglich.

Anweisungen begrenzen

Jede einzelne Anweisung innerhalb einer Routine und jeder SQL-Befehl müssen mit einem Semikolon abgeschlossen werden, auch das abschließende END. Das ist kein Problem, wenn nur ein einzelner CREATE PROCEDURE o. ä. ausgeführt werden soll. Andernfalls folgt in einer eigenen Zeile ein einzelner Schrägstrich:

```
BEGIN
    <anweisungen>
END;
/
```

29.8. Zusammenfassung

In diesem Kapitel wurden die wichtigsten Bestandteile besprochen, mit denen SQL-Befehle in eigenen Funktionen, in gespeicherten Prozeduren oder in Triggern verarbeitet werden:

- Deklaration von Parametern und Variablen
- Verwendung von Parametern und Variablen
- Verzweigungen mit IF u. a. sowie Schleifen
- Zulässigkeit von DML- und DDL-Befehlen
- Trennung zwischen einer Routine insgesamt und einer einzelnen Anweisung

Bei allen Einzelheiten müssen die Besonderheiten eines jeden DBMS beachtet werden.

29.9. Übungen

Übung 1 – Erklärungen

1. Erläutern Sie den Zweck einer (benutzerdefinierten) Funktion.
2. Erläutern Sie den Zweck einer (gespeicherten) Prozedur.
3. Erläutern Sie den Zweck eines Triggers.
4. Worin unterscheiden sich „Eigene Funktionen“ und Prozeduren?
5. Worin unterscheiden sich Prozeduren und Trigger?

Übung 2 – Der Kopf einer Routine

Erläutern Sie die folgenden Bestandteile aus dem „Kopf“ (header) einer Routine, d. h. einer Funktion oder einer Prozedur. Erwähnen Sie auch, was zu diesen Bestandteilen gehört.

1. CREATE OR ALTER bzw. CREATE OR REPLACE
2. <Parameterliste>
3. Eingabe-Parameter
4. Ausgabe-Parameter
5. RETURNS bzw. RETURN

Übung 3 – Der Rumpf einer Routine

Erläutern Sie die folgenden Bestandteile aus dem „Rumpf“ (body) einer Routine, d. h. einer Funktion oder einer Prozedur. Erwähnen Sie auch, was zu diesen Bestandteilen gehört; die Antworten können sehr knapp ausfallen, weil sie sich in Abhängigkeit vom DBMS sehr unterscheiden müssten.

1. BEGIN und END am Anfang und Ende des Rumpfes
2. BEGIN und END innerhalb des Rumpfes
3. Variablen
4. Verzweigungen
5. Schleifen
6. Zuweisung von Werten

29.10. Lösungen

Lösung zu Übung 1 – Erklärungen

1. Eine benutzerdefinierte Funktion ist eine Ergänzung zu den internen Funktionen des DBMS und liefert immer einen bestimmten Wert zurück.
2. Eine gespeicherte Prozedur (StoredProcedure, SP) ist ein Arbeitsablauf, der fest innerhalb der Datenbank gespeichert ist und Aufgaben ausführt, die innerhalb der Datenbank erledigt werden sollen – im wesentlichen ohne „Kommunikation“ mit dem Benutzer bzw. dem auslösenden Programm.
3. Ein Trigger ist ein Arbeitsablauf, der fest innerhalb der Datenbank gespeichert ist und Aufgaben ausführt, die automatisch beim Speichern innerhalb der Datenbank erledigt werden sollen.
4. Eine Funktion liefert genau einen Wert zurück, der durch RETURNS festgelegt wird. Prozeduren gibt es mit und ohne Rückgabewert.
5. Ein Trigger wird automatisch bei einem Speichern-Befehl ausgeführt, und es gibt keinen Rückgabewert. Eine Prozedur wird bewusst aufgerufen, und sie kann Rückgabewerte liefern.

Lösung zu Übung 2 – Der Kopf einer Routine

1. Dies definiert eine Routine. Dazu gehören einer der Begriffe FUNCTION und PROCEDURE sowie der Name der Routine.
2. Die Parameterliste enthält diejenigen Parameter, die beim Aufruf an die Routine übergeben werden (Eingabe-Parameter), sowie diejenigen, die nach Erledigung an den Aufrufer zurückgegeben werden (Ausgabe-Parameter). Wie Eingabe- und Ausgabe-Parameter gekennzeichnet bzw. unterschieden werden, hängt vom DBMS ab.
3. Alle Eingabe-Parameter sind (durch Komma getrennt) aufzuführen; die Reihenfolge muss beim Aufruf beachtet werden. Zu jedem Parameter gehören der Name und der Datentyp; unter Umständen ist ein Standardwert möglich.
4. Alle Ausgabe-Parameter sind (durch Komma getrennt) aufzuführen; die Reihenfolge muss bei der Auswertung im aufrufenden Programm o. ä. beachtet werden. Zu jedem Parameter gehören der Name und der Datentyp; unter Umständen ist ein Standardwert möglich.

5. RETURNS bzw. RETURN gibt bei Funktionen den Ausgabe-Parameter, d. h. das Ergebnis der Funktion an. Nur Firebird benutzt dies, um auch die Ausgabe-Parameter einer Prozedur anzugeben.

Lösung zu Übung 3 – Der Rumpf einer Routine

1. Dies beschreibt Anfang und Ende des gesamten Rumpfes. Wenn der Rumpf nur einen einzelnen Befehl enthält, kann es entfallen.
2. Dies begrenzt einzelne Abschnitte innerhalb des Rumpfes, vor allem bei Schleifen und IF-Verzweigungen. Wenn der Abschnitt nur einen einzelnen Befehl enthält, kann BEGIN/END entfallen; die Verwendung wird aber stets empfohlen.
3. Dies definiert Werte, die (nur) innerhalb der Routine benutzt werden. Sie müssen mit Namen und Datentyp ausdrücklich deklariert werden.
4. Mit IF-ELSE o. ä. können – abhängig von Bedingungen – unterschiedliche Befehle ausgeführt werden.
5. Mit WHILE, FOR o. a. können einzelne Befehle oder Abschnitte von Befehlen wiederholt ausgeführt werden. Wie oft bzw. wie lange die Schleife durchlaufen wird, hängt von der Art der Schleife (unterschiedlich nach DBMS) und Bedingungen ab.
6. Den Variablen und Parametern können Werte zugewiesen werden, und zwar durch das Gleichheitszeichen mit konkreten Angaben oder als Ergebnis von Funktionen, Prozeduren oder SELECT-Befehlen.

29.11. Siehe auch

Verschiedene Einzelheiten stehen in den folgenden Kapiteln:

- SQL-AUSDRUCK⁴
- FUNKTIONEN⁵ und FUNKTIONEN (2)⁶
- NÜTZLICHE ERWEITERUNGEN⁷

Zur SQL-Programmierung mit Oracle gibt es das (unvollständige) Wikibook PL/SQL⁸.

4 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%C3%9Cfung%20in%20SQL%3A%20SQL-%7B%7D_Befehle%23Anker%3ASQLAusdruck](http://de.wikibooks.org/wiki/Einf%C3%9Cfung%20in%20SQL%3A%20SQL-%7B%7D_Befehle%23Anker%3ASQLAusdruck)

5 Kapitel 13 auf Seite 121

6 Kapitel 15 auf Seite 157

7 Kapitel 22 auf Seite 237

8 [HTTP://DE.WIKIBOOKS.ORG/WIKI/PL%2FSQL](http://de.wikibooks.org/wiki/PL%2FSQL)

30. Eigene Funktionen

Auch wenn ein DBMS viele Funktionen zur Verfügung stellt, kommt man in der Praxis immer wieder einmal zu weiteren Wünschen und Anforderungen. Dafür kann ein Benutzer eigene Funktionen definieren und dem DBMS bekannt geben oder in einer Datenbank speichern. Einmal definiert, erspart dies künftig, die gleiche Routine immer neu zu erstellen; stattdessen wird die Funktion aufgerufen und liefert den Rückgabewert.

Firebird hat solche Funktionen erst für Version 3 angekündigt. Zurzeit kann eine Funktion nur als UDF (user-defined function) aus einer externen DLL, die mit einer Programmiersprache erstellt wurde, eingebunden werden.

Bitte haben Sie Nachsicht: Wegen der vielen Varianten bei den DBMS wurde ein Teil der Hinweise und Beispiele auch in diesem Kapitel nur nach der Dokumentation verfasst und nicht in der Praxis umgesetzt.

30.1. Funktion definieren

Benutzerdefinierte Funktionen geben (ebenso wie die eigenen Funktionen des DBMS) genau einen Wert zurück, sind also Skalarfunktionen.

30.1.1. Funktion erstellen

Mit **CREATE FUNCTION** in der folgenden Syntax (vereinfachte Version des SQL-Standards) wird eine Funktion definiert:

```
CREATE FUNCTION <routine-name>
  ( [ <parameterliste> ] )
  RETURNS <datentyp>
  [ <characteristics> ]
  <routine body>
```

Der <routine-name> der Funktion muss innerhalb eines gewissen Bereichs („Schema“ genannt) eindeutig sein und darf auch nicht als Name einer Prozedur verwendet werden. Teilweise wird verlangt, dass der Name der Datenbank ausdrücklich angegeben wird. Es empfiehlt sich, keinen Namen einer eingebauten Funktion zu verwenden.

Die Klammern sind erforderlich und kennzeichnen eine Routine. Eingabeparameter können vorhanden sein, müssen es aber nicht. Mehrere Parameter werden durch Kommata getrennt, der einzelne <parameter> wird wie folgt definiert:

```
[ IN ] <parameter-name> <datatype>
```

Der <parameter-name> muss innerhalb der Funktion eindeutig sein. Der Zusatz IN kann entfallen, weil es bei Funktionen nur Eingabe-Parameter (keine Ausgabe-Parameter) gibt.

Der RETURNS-Parameter ist wesentlich und kennzeichnet eine Funktion.

Der <datatype> sowohl für den RETURNS-Parameter als auch für die Eingabe-Parameter muss einer der SQL-Datentypen des DBMS sein.

Für <characteristics> gibt es diverse Festlegungen, wie die Funktion arbeiten soll, z. B. durch LANGUAGE mit der benutzten Programmiersprache.

<routine body> kennzeichnet den eigentlichen Arbeitsablauf, also die Schritte, die innerhalb der Routine ausgeführt werden sollen. Bei diesen Befehlen handelt es sich um „normale“ SQL-Befehle, die mit Bestandteilen der SQL-Programmiersprache verbunden werden. Zum Semikolon, das den Abschluss des CREATE-Befehls darstellen sollte, aber innerhalb des Inhalts für jede einzelne Anweisung benutzt wird, beachten Sie bitte auch die Hinweise zur SQL-Programmierung unter *Anweisungen begrenzen*.

Bei einer Funktion muss – beispielsweise durch eine RETURN-Anweisung – der gesuchte Rückgabewert ausdrücklich festgelegt werden. In der Regel werden die Befehle durch BEGIN...END zusammengefasst; bei einem einzelnen Befehl ist dies nicht erforderlich.

30.1.2. Funktion ausführen

Eine benutzerdefinierte Funktion kann wie jede interne Funktion des DBMS benutzt werden. Sie kann an jeder Stelle benutzt werden, an der ein einzelner Wert erwartet wird, wie in den Beispielen zu sehen ist.

30.1.3. Funktion ändern oder löschen

Mit **ALTER FUNCTION** wird die Definition einer Funktion geändert. Dafür gilt die gleiche Syntax:

```
ALTER FUNCTION <routine-name>
  ( [ <parameterliste> ] )
  RETURNS <datentyp>
  [ <characteristics> ]
  <routine body>
```

Mit **DROP FUNCTION** wird die Definition einer Funktion gelöscht.

```
DROP FUNCTION <routine-name>;
```

30.2. Beispiele

30.2.1. Einfache Bestimmung eines Wertes

Das folgende Beispiel erstellt den Rückgabewert direkt aus der Eingabe.

► **Aufgabe:** Zur Begrüßung wird ein Name mit einem Standardtext, der von der Tageszeit abhängt, verbunden.

MySQL Quelltext

```
CREATE FUNCTION Hello (s CHAR(20))
  RETURNS CHAR(50)
  RETURN WHEN
    CASE CURRENT_TIME < '12:00' THEN 'Guten Morgen, ' + s + '!'
    CASE CURRENT_TIME < '18:00' THEN 'Guten Tag, ' + s + '!'
    ELSE 'Guten Abend, ' + s + '!'
  END;
```

Bei diesem Beispiel kann auf BEGIN...END verzichtet werden, weil der „Rumpf“ der Funktion nur eine einzige Anweisung enthält, nämlich RETURN unter Benutzung des WHEN-Ausdrucks.

Eine solche Funktion wird wie jede eingebaute Funktion benutzt.

```
SELECT Hello('Juetho') [from fiktiv];
```

```
'Guten Abend, Juetho!'
```

30.2.2. Text in Anführungszeichen einschließen

Im folgenden Beispiel werden mehr Maßnahmen benötigt, bis der RETURN-Wert feststeht. Der praktische Nutzen liegt im Export von Daten aus einem System zu einem anderen.

► **Aufgabe:** Ein gegebener Text soll in Anführungszeichen eingeschlossen werden; Gänsefüßchen innerhalb des Textes müssen verdoppelt werden.

```
MySQL Quelltext
```

```
CREATE FUNCTION Quoting
  ( instring VARCHAR(80) )
  RETURNS ( VARCHAR(100) )
AS
BEGIN
  if (instring CONTAINING('')) THEN BEGIN
    instring = REPLACE( instring, '', '""' );
  END
  RETURN CONCAT( '', instring, '' );
END
```

Diese Funktion kann direkt aufgerufen werden:

```
select Quoting('Schulze'), Quoting('Restaurant "Abendrot"') [from
fiktiv];
```

```
"Schulze" "Restaurant ""Abendrot"""
```

30.2.3. Anzahl der Mitarbeiter einer Abteilung

Bei der folgenden Funktion werden zunächst weitere Informationen benötigt.

► **Aufgabe:** Suche zu einer Abteilung (angegeben durch das *Kuerzel*) die Anzahl der Mitarbeiter.

Oracle Quelltext

```
CREATE OR REPLACE FUNCTION AnzahlMitarbeiter
  ( abt CHAR(4) )
  RETURN ( INTEGER )
AS
  anzahl INTEGER;
BEGIN
  select COUNT(*) into anzahl
    from Mitarbeiter mi
      join Abteilung ab on ab.ID = mi.Abtteilung_ID
     where ab.Kuerzel = abt;
  RETURN anzahl;
END
```

Damit erhalten wir die Anzahl der Mitarbeiter einer bestimmten Abteilung:

```
select AnzahlMitarbeiter('Vert')   [from fiktiv] ;
```

```
AnzahlMitarbeiter : 4
```

30.3. Zusammenfassung

In diesem Kapitel lernten wir die Grundregeln für die Erstellung eigener Funktionen kennen:

- Benutzerdefinierte Funktionen sind immer Skalarfunktionen, die genau einen Wert zurückgeben.
- Der Datentyp des Rückgabewerts ist in der RETURNS-Klausel anzugeben, der Wert selbst durch eine RETURN-Anweisung.
- Komplexe Maßnahmen müssen in BEGIN...END eingeschlossen werden; eine Funktion kann aber auch nur aus der RETURN-Anweisung bestehen.

30.4. Übungen

Wenn bei den folgenden Übungen der Begriff „Funktion“ ohne nähere Erklärung verwendet wird, ist immer eine „Eigene Funktion“, d.h. eine benutzerdefinierte Funktion gemeint.

Unter „Skizzieren“ (Übung 3, 5) ist gemeint: Angeben aller Eingabe- und Ausgabeparameter sowie Variablen mit sinnvollen Namen und Datentypen, möglichst genaue Beschreibung des Arbeitsablaufs mit Pseudo-Code oder normaler Sprache. *Tipp: Die Parameter ergeben sich in der Regel aus der Aufgabenstellung. Aus der Überlegung des Arbeitsablaufs folgen die Variablen.*

Übung 1 – Definition einer Funktion

Welche der folgenden Aussagen sind richtig, welche sind falsch?

1. Eine Funktion dient zur Bestimmung genau eines Wertes.
2. Eine Funktion kann höchstens einen Eingabe-Parameter enthalten.
3. Eine Funktion kann mehrere Ausgabe-Parameter enthalten.
4. Eine Funktion kann einen oder mehrere SQL-Befehle verwenden.
5. Der Rückgabewert wird mit einer RETURN-Anweisung angegeben.
6. Der Datentyp des Rückgabewerts ergibt sich automatisch aus der RETURN-Anweisung.
7. Die Definition einer Funktion kann nicht geändert werden; sie kann nur gelöscht und mit anderem Inhalt neu aufgenommen werden.

Übung 2 – Definition einer Funktion prüfen

Nennen Sie in der folgenden Definition Punkte, die unabhängig vom SQL-Dialekt falsch sind. (Je nach DBMS können noch andere Punkte falsch sein, danach wird aber nicht gefragt.) Die Funktion soll folgende Aufgabe erledigen:

- Es wird eine Telefonnummer in beliebiger Formatierung als Eingabe-Parameter übergeben, z. B. als '(0049 / 030) 99 68-32 53'.
- Das Ergebnis soll diese Nummer ganz ohne Trennzeichen zurückgeben.
- Sofern die Ländervorwahl enthalten ist, soll das führende '00' durch '+' ersetzt werden.

```
CREATE Telefon_Standard AS Function
  ( IN Eingabe VARCHAR(20),
    OUT Ausgabe VARCHAR(20) )
AS
```

```

DECLARE INTEGER x1, i1;
DECLARE CHAR c1;
BEGIN
  -- Rückgabewert vorbelegen
  Ausgabe = "";
  -- Länge der Schleife bestimmen
  i1 = CHAR_LENGTH(Eingabe);
  -- die Schleife für jedes vorhandene Zeichen verarbeiten
  WHILE (i1 < x1)
  DO
    -- das nächste Zeichen auslesen
    x1 = x1 + 1
    c1 = SUBSTRING(Eingabe from x1 for 1);
    -- dieses Zeichen prüfen: ist es eine Ziffer
    if (c1 >= '0' and (c1 <= '9'))
    THEN
      -- ja: an den bisherigen Rückgabewert anhängen
      Ausgabe = Ausgabe || c1;
    END
  END
  -- Zusatzprüfung: '00' durch '+' ersetzen
  IF (Ausgabe STARTS WITH '00')
  THEN
    Ausgabe = '+' || SUBSTRING(Ausgabe from 3 for 20)
  END
END

```

Übung 3 – Funktion *DateToString* erstellen

Skizzieren Sie eine Funktion **DateToString**: Aus einem Date- oder DateTime-Wert als Eingabe-Parameter soll eine Zeichenkette mit genau 8 Zeichen der Form 'JJJJMMTT' gemacht werden. Benutzen Sie dafür nur die EXTRACT- und LPAD-Funktionen sowie die „Addition“ von Zeichenketten; Sie können davon ausgehen, dass eine Zahl bei LPAD korrekt als Zeichenkette verstanden wird.

Übung 4 – Funktion *DateToString* erstellen

Erstellen Sie die Funktion aus der vorigen Übung.

Übung 5 – Funktion *String_Insert* erstellen

Skizzieren Sie eine Funktion **String_Insert**: In eine Zeichenkette soll an einer bestimmten Position eine zweite Zeichenkette eingefügt werden; alle Zeichenketten sollen max. 80 Zeichen lang sein. Benutzen Sie dafür nur SUBSTRING und CAST sowie die „Addition“ von Zeichenketten; stellen Sie außerdem sicher, dass das Ergebnis die mögliche Maximallänge nicht überschreitet und schneiden Sie ggf. ab.

Übung 6 – Funktion *String_Insert* erstellen

Erstellen Sie die Funktion aus der vorigen Übung.

30.5. Lösungen

Lösung zu Übung 1 – Definition einer Funktion

Die Aussagen 1, 4, 5 sind richtig. Die Aussagen 2, 3, 6, 7 sind falsch.

Lösung zu Übung 2 – Definition einer Funktion prüfen

- Zeile 1: Die Definition erfolgt durch CREATE FUNCTION <Name>.
- Zeile 2: Bei einer Funktion gehören in die Klammern nur die Eingabe-Parameter.
- Zeile 3: Der Ausgabe-Parameter muss mit RETURNS festgelegt werden.
- Zeile 4: Bei jeder Variablen ist zuerst der Name, dann der Typ anzugeben.
- Zeile 12: Der Startwert für die Schleifenvariable $x1 = 0$ fehlt.
- Zeile 14: Die Schleife verarbeitet mehrere Befehle, muss also hinter DO auch ein BEGIN erhalten.
- Zeile 16 und 29: Jeder einzelne Befehl muss mit einem Semikolon abgeschlossen werden.
- Zeile 19: Die Klammern sind falsch gesetzt; am einfachsten wird die zweite öffnende Klammer entfernt.
- Zeile 20/23: Entweder es wird THEN BEGIN...END verwendet, oder das END wird gestrichen.
- Zeile 28/30 enthält den gleichen Fehler.
- Zeile 31: Es fehlt die RETURN-Anweisung, mit der der erzeugte Ausgabe-String zurückgegeben wird.

Lösung zu Übung 3 – Funktion *DateToString* erstellen

- Eingabe-Parameter: ein DateTime-Wert *Eingabe*
- Ausgabe-Parameter: ein CHAR(8) *Ausgabe*
- drei INTEGER-Variable: jjjj, mm, tt
- mit 3x EXTRACT werden die Bestandteile jjjj, mm, tt herausgeholt
- diese werden mit LPAD auf CHAR(2) für Monat und Tag gesetzt
- schließlich werden diese Werte per „Addition“ verknüpft und

- als Ergebnis zurückgegeben

Lösung zu Übung 4 – Funktion *DateToString* erstellen

```
CREATE FUNCTION DateToString
  ( Eingabe DATE )
RETURNS ( Ausgabe CHAR(8) )
AS
  DECLARE VARIABLE jjjj int;
  DECLARE VARIABLE mm int;
  DECLARE VARIABLE tt int;
BEGIN
  jjjj = EXTRACT(YEAR from Eingabe);
  mm = EXTRACT(MONTH from Eingabe);
  tt = EXTRACT(DAY from Eingabe);
  Ausgabe = jjjj || LPAD(mm, 2, '0') || LPAD(tt, 2, '0');
  RETURN Ausgabe;
END
```

Lösung zu Übung 5 – Funktion *String_Insert* erstellen

- Eingabe-Parameter: Original-String *Eingabe*, *pos* als Position (der Name *Position* ist als interne Funktion verboten), der Zusatz-String *part*
- Ausgabe-Parameter: die neue Zeichenkette *Ausgabe*
- Variable: *temp* mit Platz für 160 Zeichen
- es wird einfach eine neue Zeichenkette *temp* zusammengesetzt aus drei Teilen:
 - der Anfang von *Eingabe* mit *pos* Zeichen
 - dann der Zusatz-String *part*
 - dann der Rest von *Eingabe* ab Position *pos + 1*
- falls die Zeichenkette *temp* zu lang ist, werden mit *SUBSTRING* nur noch die ersten 80 Zeichen verwendet
- das Ergebnis muss wegen der seltsamen Definition von *SUBSTRING* mit *CAST* auf *VARCHAR(80)* gebracht werden und
- kann dann als Rückgabewert verwendet werden

Lösung zu Übung 6 – Funktion *String_Insert* erstellen

```
CREATE FUNCTION String_Insert
  ( Eingabe VARCHAR(80), pos INTEGER, part VARCHAR(80) )
RETURNS ( Ausgabe VARCHAR(80) )
AS
  -- genügend Zwischenspeicher bereitstellen
```

```
DECLARE VARIABLE temp VARCHAR(160);
BEGIN
  -- Teil 1, dazu Zwischenteil, dann Teil 2
  temp = SUBSTRING( Eingabe from 1 for pos )
    || part
    || SUBSTRING( Eingabe from pos + 1 );
  -- auf jeden Fall auf die Maximallänge von 80 Zeichen bringen
  if (CHAR_LENGTH(temp) > 80) THEN
    Ausgabe = cast( substring(temp from 1 for 80) as varchar(80));
  ELSE
    Ausgabe = cast( temp as varchar(80));
  RETURN Ausgabe;
END
```

30.6. Siehe auch

Verschiedene Einzelheiten werden in den folgenden Kapiteln behandelt:

- [SQL-PROGRAMMIERUNG¹](#)
- [DATENTYPEN²](#)
- [FUNKTIONEN³ und FUNKTIONEN \(2\)⁴](#)

1 Kapitel 29 auf Seite 363

2 Kapitel 12 auf Seite 107

3 Kapitel 13 auf Seite 121

4 Kapitel 15 auf Seite 157

31. Prozeduren

Eine Prozedur – gespeicherte Prozedur, engl. StoredProcedure (SP) – ist vorgesehen für Arbeitsabläufe, die „immer wiederkehrende“ Arbeiten direkt innerhalb der Datenbank ausführen sollen.

Bitte haben Sie Nachsicht: Wegen der vielen Varianten bei den DBMS beschränkt sich dieses Kapitel bei den Hinweisen und Beispielen auf Firebird. Zusammen mit den speziellen Hinweisen zur SQL-Programmierung sollten sie problemlos an andere DBMS-Regeln angepasst werden können.

31.1. Ein Überblick

Zur **Verwendung von Prozeduren** gibt es zwei gegensätzliche Standpunkte, die mit den Begriffen **Fat Server** und **Fat Client** zusammengefasst werden können:

- Beim *Fat Server* wird so viel Funktionalität wie möglich in die (zentral gespeicherte) Datenbank gelegt.
- Beim *Fat Client* ist die Datenbank nur für die Speicherung der Daten vorgesehen; sämtliche Verarbeitung erfolgt auf dem Arbeitsplatzrechner.

Der große Vorteil eines *Fat Server* liegt darin, dass Daten direkt in der Datenbank verarbeitet werden können, sofern keine Ansicht der Daten auf dem Arbeitsplatz benötigt wird. Es ist natürlich überflüssig, die Daten zuerst von der Datenbank zum Arbeitsplatz zu kopieren, dann automatisch zu verarbeiten und schließlich das Arbeitsergebnis (ohne manuelle Überprüfung) auf die Datenbank zurückzukopieren. Einfacher ist es, alles in der Datenbank ausführen zu lassen.

Der große Vorteil eines *Fat Client* liegt darin, dass das Client-Programm meistens schneller und einfacher geändert werden kann als die Arbeitsabläufe innerhalb der Datenbank.

In der Praxis ist eine Mischung beider Verfahren am sinnvollsten. Zur Steuerung der Arbeitsabläufe in der Datenbank werden die gespeicherten Prozeduren verwendet; Beispiele dafür werden in diesem Abschnitt behandelt.

31.2. Prozedur definieren

31.2.1. Prozedur erstellen

Die **Syntax** für die Definition einer **Prozedur** sieht so aus:

```
CREATE OR ALTER PROCEDURE <routine-name>
    ( [ <parameterliste> ] )
AS
BEGIN
    [ <variablenliste> ]
    <routine body>
END
```

Notwendig sind folgende Angaben:

- neben dem Befehlsnamen der Name der Prozedur
- 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 Parametern, sofern Werte übergeben oder abgefragt werden sollen
 - Bei den meisten DBMS werden Eingabe- und Ausgabe-Parameter durch Schlüsselwörter wie IN und OUT unterschieden, sie stehen dabei innerhalb derselben Liste.
 - Bei Firebird enthält die Parameterliste nur die Eingabe-Parameter; die Ausgabe-Parameter stehen hinter RETURNS in einer eigenen Liste mit Klammern.
- eine Liste von Variablen, die innerhalb der Prozedur 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

Bei den Befehlen innerhalb der Prozedur handelt es sich um „normale“ SQL-Befehle sowie um Bestandteile der SQL-Programmiersprache. Zum Semikolon, das den Abschluss des CREATE-Befehls darstellen sollte, aber innerhalb des Inhalts bereits für jede einzelne Anweisung benutzt wird, beachten Sie bitte auch die Hinweise zur SQL-Programmierung unter *Anweisungen begrenzen*.

31.2.2. Prozedur ausführen

Es gibt mehrere Verfahren, wie eine Prozedur vom Nutzer, aus einem Anwendungsprogramm oder aus einer anderen Prozedur oder einem Trigger heraus benutzt werden kann.

Der direkte Weg geht bei Firebird über **Execute Procedure** und führt eine Prozedur aus:

```
EXECUTE PROCEDURE <routine-name> [ <eingabe-parameter> ]  
  [ RETURNING_VALUES <ausgabe-parameter> ] ;
```

Dem Namen der Prozedur folgen (soweit erforderlich) die Eingabe-Parameter; mehrere werden mit Komma getrennt. Sofern die Prozedur Werte zurückliefert, werden sie in eine oder mehrere Variablen eingetragen, die dem Begriff RETURNING_VALUES folgen.

Bei den anderen DBMS gehören zu EXECUTE (MySQL: CALL) nicht nur die Eingabe-, sondern auch die Ausgabe-Parameter. Beispiele stehen im Kapitel zur SQL-Programmierung bei den Erläuterungen des jeweiligen DBMS.

Eine Prozedur kann auch mit **Select Procedure** wie eine Abfrage verwendet werden. Dies ist vor allem dann erforderlich, wenn eine Reihe von Werten oder Datensätzen zurückgeliefert werden.

```
SELECT * FROM <routine-name> [ <eingabe-parameter> ] ;
```

Die Ausgabe-Parameter stehen dann in der Liste der Spalten, also in der Ergebnismenge des SELECT-Befehls.

Man könnte auch an **Execute Block** denken, aber dieser Hinweis passt hier überhaupt nicht. Ein Block ist eine Menge von Anweisungen, die einmalig benutzt werden, aber eben nicht in der Datenbank gespeichert werden. Ein solcher Block *ersetzt* eine gespeicherte Prozedur, wenn das Speichern überflüssig ist, und wird bei der SQL-Programmierung unter *Routinen ohne feste Speicherung* beschrieben.

31.2.3. Prozedur ändern

Eine Änderung einer Prozedur ist nur möglich durch eine vollständige Neudefinition mit allen Bestandteilen. Deshalb wird es heutzutage fast immer durch

CREATE OR ALTER zusammengefasst. Wenn dies nicht möglich ist, dann muss ein einzelner ALTER-Befehl benutzt werden, der alle Bestandteile der obigen Erläuterungen enthält.

31.2.4. Prozedur löschen

Das geht wie üblich mit dem DROP-Befehl:

```
DROP PROCEDURE <routine-name>;
```

31.3. Beispiele

Um die Beispiele für ein anderes DBMS zu übertragen, sind vor allem folgende Punkte zu ändern:

- Die Ausgabe-Parameter folgen nicht nach RETURNS, sondern gehören in die Parameterliste.
- Die Deklaration der Variablen ist anzupassen, ebenso die Kennzeichnung mit Doppelpunkt.
- Schleifen und ähnliche Maßnahmen sind umzuschreiben.
- Dies gilt auch dafür, wie Werte – z. B. bei einem SELECT-Befehl – übernommen werden.

31.3.1. Ersatz für eine View mit Parametern

Im Kapitel zu VIEWS wollten wir eine VIEW mit variabler Selektion erstellen, konnten aber keine variablen Bedingungen einbinden. Mit einer Prozedur geht es.

► **Aufgabe:** Mache aus der View *Mitarbeiter_in_Abteilung* eine Prozedur *Mitarbeiter_aus_Abteilung*, die die Abteilungsnummer als Parameter entgegennimmt.

Firebird Quelltext

```
CREATE OR ALTER PROCEDURE Mitarbeiter_aus_Abteilung
  ( Abt INTEGER)
RETURNS ( Personalnummer VARCHAR(10),
          Name              VARCHAR(30),
```

```

        Vorname      VARCHAR(30),
        Geburtsdatum DATE )
AS
BEGIN
    FOR SELECT Personalnummer, Name, Vorname, Geburtsdatum
        FROM Mitarbeiter
        WHERE Abteilung_ID = :Abt
        ORDER BY Ist_Leiter, Name, Vorname
        INTO :Personalnummer, :Name, :Vorname, :Geburtsdatum
    DO SUSPEND;
END

```

Als Prozedur enthält die Abfrage folgende Bestandteile:

- Als **Eingabe-Parameter** wird die gewünschte Abteilungsnummer erwartet und in der Variablen *Abt* gespeichert.
- Als **Ausgabe-Parameter** werden die gewünschten Spalten der Tabelle *Mitarbeiter* aufgeführt.
- Weitere **Variable** werden nicht benötigt.

Zwischen BEGIN und END steht der eigentliche Arbeitsablauf mit **Anweisungen** gemäß SQL-Programmierung.

- Es handelt sich dabei vor allem um einen SELECT-Befehl mit der gewünschten WHERE-Klausel, die bei der VIEW nicht möglich war.
- Diese Abfrage wird in eine FOR-DO-Schleife eingebunden.
- Dabei werden die Spalten eines jeden ausgewählten Datensatzes mit INTO in die Ausgabe-Variablen übertragen und per SUSPEND zurückgegeben.

Damit haben wir die gewünschte variable Abfrage. Bei der Verwendung ist es für den Anwender gleichgültig, ob er sie als View oder als Prozedur aufzurufen hat:

► **Aufgabe:** Hole alle Mitarbeiter einer bestimmten Abteilung.

Aufruf als View

```

SELECT * FROM Mitarbeiter_in_Abteilung
WHERE Abt = 5;

```

Aufruf als Prozedur

```

SELECT * FROM Mitarbeiter_aus_Abteilung (5);

```

31.3.2. INSERT in mehrere Tabellen

Wenn in der Beispieldatenbank ein neuer Versicherungsvertrag eingegeben werden soll, benötigt man in der Regel drei INSERT-Befehle:

- für einen neuen Eintrag in der Tabelle *Fahrzeug*
- für einen neuen Eintrag in der Tabelle *Versicherungsnehmer*
- für einen neuen Eintrag in der Tabelle *Versicherungsvertrag* mit Verweisen auf die beiden anderen Einträge

Bei den ersten beiden Einträgen ist durch SELECT die neue ID abzufragen und beim dritten INSERT zu verwenden. Warum sollte man sich die Arbeit in dieser Weise erschweren? Soll doch die Datenbank einen einzigen INSERT-Befehl entgegennehmen und die Parameter selbständig auf die beteiligten Tabellen verteilen. ► **Aufgabe:** Speichere einen neuen Versicherungsvertrag mit allen Einzelheiten.

Firebird Quelltext

```
create or alter PROCEDURE Insert_Versicherungsvertrag
( /* Bestandteile des Vertrags */
  Vertragsnummer VARCHAR(20),
  Abschlussdatum DATE,
  Art CHAR(2),
  Praemiensatz INTEGER,
  Basispraemie DECIMAL(9,0),
  Mitarbeiter_ID INTEGER,
  /* Angaben zum Versicherungsnehmer */
  Name VARCHAR(30),
  Vorname VARCHAR(30),
  Geschlecht CHAR(1),
  Geburtsdatum DATE,
  Fuehrerschein DATE,
  Ort VARCHAR(30),
  PLZ CHAR(5),
  Strasse VARCHAR(30),
  Hausnummer VARCHAR(10),
  Eigener_Kunde CHAR(1),
  Versicherungsgesellschaft_ID INTEGER,
  /* Angaben zum Fahrzeug */
  Kennzeichen VARCHAR(10),
  Farbe VARCHAR(30),
  Fahrzeugtyp_ID INTEGER
)
RETURNS ( NewID INTEGER )
AS
  DECLARE VARIABLE NewFahrzeugID integer;
  DECLARE VARIABLE NewVersnehmerID integer;
BEGIN
  NewFahrzeugID = NEXT VALUE FOR Fahrzeug_ID;
  INSERT into Fahrzeug
    values ( :NewFahrzeugID, :Kennzeichen, :Farbe, :Fahrzeugtyp_ID );
  NewVersnehmerID = NEXT VALUE FOR Versicherungsnehmer_ID;
  insert into Versicherungsnehmer
    values ( :NewVersnehmerID, :Name, :Vorname, :Geburtsdatum,
  :Fuehrerschein,
    :Ort, :PLZ, :Strasse, :Hausnummer,
    :Eigener_Kunde, :Versicherungsgesellschaft_ID,
```



```

:Geschlecht );
NewID = NEXT VALUE FOR Versicherungsvertrag_ID;
insert into Versicherungsvertrag
    values ( :NewID, :Vertragsnummer, :Abschlussdatum, :Art,
:Mitarbeiter_ID,
           :NewFahrzeugID, :NewVersnehmerID, :Praemiensatz,
:Abschlussdatum, :Basispraemie );
SUSPEND;
END

```

Als **Eingabe-Parameter** werden alle Werte benötigt, die der Sachbearbeiter für die einzelnen Tabellen eingeben muss (siehe die durch Kommentare getrennten Abschnitte).

Als **Ausgabe-Parameter** wollen wir die ID für den neuen Vertrag erhalten.

Als **Variable** werden die Verweise auf die zugeordneten Tabellen *Fahrzeug* und *Versicherungsnehmer* vorgesehen.

Der **Arbeitsablauf** ist ganz einfach strukturiert:

- Hole die nächste ID für die Tabelle *Fahrzeug* und speichere den nächsten Eintrag in dieser Tabelle.
- Hole die nächste ID für die Tabelle *Versicherungsnehmer* und speichere den nächsten Eintrag in dieser Tabelle.
- Hole die nächste ID für die Tabelle *Versicherungsvertrag* und speichere den nächsten Eintrag in dieser Tabelle. Benutze dafür die IDs der beiden anderen Tabellen.

Bei einem DBMS mit automatischem Zähler wird statt der „nächsten ID“ die gerade neu vergebene ID abgefragt. Möglichkeiten dafür enthält der Abschnitt *Die letzte ID abfragen* des Kapitels „Tipps und Tricks“.

Damit wird ein neuer Vertrag mit einem einzigen Befehl erstellt:

► **Aufgabe:** Speichere einen neuen Vertrag mit allen Angaben.

Firebird Quelltext

```

EXECUTE PROCEDURE Insert_Versicherungsvertrag
( 'HS-38', '03.11.2009', 'VK', 125, 870, 11,
  'Graefing', 'Charlotte', 'W', '09.11.1989', '26.02.2008',
  'Hattingen', '45529', 'Laakerweg', '17 b', 'J', NULL,
  'BO-MC 413', 'gelb', 8 );

```

```

----- Procedure executing results: -----
NEWID = 29      /* die ID des neuen Vertrags */

```

31.3.3. Automatisches UPDATE gemäß Bedingungen

Bei einer Versicherungsgesellschaft muss regelmäßig der Prämiensatz eines Vertrags neu berechnet werden: Bei verschuldeten Schadensfällen wird er (abhängig von der Höhe des Schadens) erhöht, bei Schadensfreiheit verringert. Dies ist eine Aufgabe, die die Datenbank selbständig erledigen kann und soll. Das ist ein komplexer Arbeitsablauf mit mehreren Prüfungen; vor allem die unterschiedliche Zuordnung neuer Werte wird hier stark vereinfacht.

► **Aufgabe:** Berechne für alle (eigenen) Verträge, ob eine neue Prämienrechnung ansteht. Dabei ist zu prüfen, ob wegen neuer Schadensfälle der Prämiensatz zu erhöhen oder wegen Schadensfreiheit zu verringern ist.

Die einzelnen Schritte werden anhand des folgenden Codes erläutert.

Firebird Quelltext

```
create or alter PROCEDURE Update_Praemiensatz
    ( Aktualisierung DATE default CURRENT_DATE )
    RETURNS ( Erledigt INTEGER )
AS
    DECLARE VARIABLE current_id INTEGER;
    DECLARE VARIABLE current_fz INTEGER;
    DECLARE VARIABLE current_aenderung DATE;
    DECLARE VARIABLE vergleich_aenderung DATE;
    DECLARE VARIABLE current_satz INTEGER;
    DECLARE VARIABLE current_value DECIMAL(16,2);
BEGIN
    Erledigt = 0;
    SUSPEND;
    /* Vorarbeit: Anfänger werden auf Praemiensatz 200 gesetzt; das kann
    aber
        nur für noch nicht behandelte Verträge gelten und muss deshalb noch
        vor der nächsten IS NULL-Prüfung erledigt werden. */
    UPDATE Versicherungsvertrag vv
        SET Praemiensatz = 200
        WHERE Praemienaenderung is null
            and Versicherungsnehmer_ID
                /* bestimme die Liste der Anfänger, aber nur für eigene Kunden
    */
                in ( SELECT ID
                    from Versicherungsnehmer vn
                        /* wenn der Führerschein beim Vertragsabschluss
    weniger als 2 Jahre alt ist */
                        where vn.Eigener_kunde = 'J'
                            and ( ( DATEADD( YEAR, 2, vn.Fuehrerschein ) >
    vv.Abschlussdatum )
                        /* wenn der VersNehmer beim Führerschein-Erwerb noch
    nicht 21 Jahre alt ist */
                            or ( DATEADD( YEAR, 21, vn.Geburtsdatum ) >
    vn.Fuehrerschein ) ) );
    /* Schritt 1: zu bearbeiten sind alle Verträge für eigene Kunden, deren
```

```

letzte
  Prämienänderung „zu lange“ zurückliegt */
for SELECT vv.ID, Fahrzeug_ID,
        CASE WHEN Praemienaenderung is null THEN Abschlussdatum
              ELSE Praemienaenderung
        END,
        Praemiensatz
  from Versicherungsvertrag vv
        join Versicherungsnehmer vn on vn.Id =
vv.Versicherungsnehmer_id
  where vn.Eigener_Kunde = 'J'
        and ( Praemienaenderung is null or Praemienaenderung <=
:Aktualisierung)
        into :current_id, :current_fz, :current_aenderung, :current_satz
DO BEGIN
  /* Schritt 2: wegen der Übersicht das mögliche Schlussdatum vorher
bestimmen */
  vergleich_aenderung = DATEADD( YEAR, 1, current_aenderung );
  vergleich_aenderung = DATEADD( DAY, -1, vergleich_aenderung );
  /* Schritt 3: weitere Bearbeitung, sofern die Aktualisierung über das
Vergleichsdatum hinausgeht */
  if (Aktualisierung >= vergleich_aenderung) THEN
  BEGIN
    /* Schritt 4: suche zu diesem Vertrag, d.h. diesem Fahrzeug alle
Schadensfälle in dieser Zeit
    und summiere die Schadenssumme nach Schuldanteil */
    select SUM( sf.Schadenshoehe * zu.Schuldanteil / 100 )
      from Zuordnung_SF_FZ zu
            join Schadensfall sf on zu.Schadensfall_id = sf.Id
            where zu.Fahrzeug_ID = :current_fz
            and sf.Datum between :current_aenderung and
:vergleich_aenderung
    into :current_value;
    /* Schritt 5: abhängig von (anteiliger) Schadenssumme und
bisherigem Prämienatz
    wird der neue Prämienatz bestimmt und das Datum weitergesetzt
*/
    update Versicherungsvertrag
      set Praemiensatz =
        CASE
          WHEN :current_value is null THEN CASE
            WHEN :current_satz >
100
              THEN :current_satz -
20
            WHEN :current_satz BETWEEN
40 AND 100
              THEN :current_satz -
10
          ELSE 30
        END
          WHEN :current_value = 0 THEN :current_satz - 10
          WHEN :current_value < 500 THEN :current_satz
          WHEN :current_value < 1000 THEN :current_satz + 10
          WHEN :current_value >= 1000 THEN :current_satz + 30
        END,
        Praemienaenderung = DATEADD( YEAR, 1, :current_aenderung )
  where ID = :current_id;

```

```
Erledigt = Erledigt + 1;  
END  
END  
SUSPEND;  
END
```

Als **Eingabe-Parameter** wird nur das Datum der derzeitigen *Aktualisierung* benötigt. Ein Vertrag wird dann geprüft, wenn der Zeitraum der nächsten Prämienrechnung – vom Datum der letzten *Prämienänderung* aus ein Jahr – das Datum der *Aktualisierung* enthält. Dieses Verfahren stellt sicher, dass immer nur „neue“ Schadensfälle und nur die jeweils anstehenden Prämienrechnungen berücksichtigt werden.

Als **Ausgabe-Parameter** nehmen wir nur die Anzahl der berechneten Verträge, also der geänderten Erledigt-Vermerke.

Als **Variable** benötigen wir Zwischenwerte für alle Angaben, die im Arbeitsablauf benutzt werden.

Der **Arbeitsablauf** umfasst die folgenden Punkte:

- Der Rückgabewert wird mit dem Anfangswert belegt und angezeigt zum Zeichen dafür, dass die Prozedur arbeitet.
- Als **Vorarbeit** wird ein „Anfänger“, der als *Eigener_Kunde* gespeichert ist, auf einen Prämienatz von 200 gesetzt. Als Anfänger gilt, wenn der Führerschein beim Abschluss des Vertrags noch nicht zwei Jahre alt ist oder wenn der Führerschein vor dem 21. Geburtstag erworben wurde.

Bei der Prüfung auf „zwei Jahre“ handelt es sich nicht um einen sachlichen Fehler: Dieser Arbeitsablauf wird innerhalb des ersten Versicherungsjahres ausgeführt; später ist die Hauptbedingung „Praemienaenderung IS NULL“ niemals mehr gegeben. In der Praxis müssen solche Bedingungen wegen möglicher Missverständnisse vorher ganz genau formuliert werden.

- Schritt 1 wählt die Verträge aus, die „im Moment“ zu prüfen und ggf. zu bearbeiten sind: alle eigenen Kunden, deren Prämienatz noch nie geprüft wurde (dann wird das Abschlussdatum, also das Datum der ersten Rechnung, zur Berechnung verwendet) oder deren letzte Prämienänderung vor dem Datum der Aktualisierung liegt. Durch die FOR-SELECT-Schleife wird jeder dieser Verträge innerhalb von DO BEGIN...END einzeln bearbeitet; die dazu benötigten Werte werden in den Variablen zwischengespeichert.
- Schritt 2 berechnet das Schlussdatum der letzten Prämienrechnung. Wir arbeiten hier nur mit Jahresrechnungen; da bei BETWEEN beide Grenzwerte einbezogen werden, müssen wir das Schlussdatum um einen Tag verringern.

- Schritt 3 erledigt die letzte Vorprüfung:
 - Wenn das Datum der Aktualisierung vor dem Schlussdatum liegt, soll noch keine neue Rechnung erfolgen.
 - Aber wenn das Datum der Aktualisierung das Schlussdatum der letzten Prämienrechnung überschreitet, ist eine neue Rechnung und damit eine Überprüfung des Prämienatzes fällig.
- Schritt 4 berechnet die Summe aller Schadensfälle:
 - Berücksichtigt wird die Summe der Schadenshöhe je Vorfall, also aus der Tabelle *Schadensfall*.
 - Dieser Wert wird bei der Summierung anteilig nach dem Schuldanteil aus der Tabelle *Zuordnung_SF_FZ* berücksichtigt.
 - Das Ergebnis bei *current_value* lautet NULL, wenn das Fahrzeug nicht in einen Schadensfall verwickelt wurde, es lautet 0 bei fehlender Teilschuld und größer als 0 bei Mit- oder Vollschuld.
- Schritt 5 berechnet den Prämienatz neu:
 - Ohne Schadensfall wird er um 10 Punkte verringert, wenn er bisher kleiner/gleich 100 beträgt, und um 20 Punkte, wenn er bisher größer ist.
 - Das Minimum von 30 kann nicht unterschritten werden.
 - Mit Schadensfall ohne Teilschuld wird der Prämienatz um 10 Punkte verringert.
 - Bei einem sehr geringen Schaden bleibt der Prämienatz unverändert.
 - Bei einem kleineren Schaden wird er um 10 Punkte, sonst um 30 Punkte erhöht.
- Gleichzeitig wird das Datum der Prämienänderung um 1 Jahr weitersetzt.

Mit dieser Prozedur können die neuen Prämienätze berechnet und für die nächste Prämienrechnung vorbereitet werden:

► **Aufgabe:** Berechne die Prämienätze für alle (eigenen) Verträge, bei denen im 3. Quartal 2009 eine neue Prämienrechnung ansteht.

```
EXECUTE PROCEDURE Update_Praemiensatz('30.09.2009');
```

```
----- Procedure executing results: -----
Erledigt = 7
```

31.3.4. Testdaten in einer Tabelle erstellen

Das übliche Vorgehen, wenn in einer Tabelle viele Testdaten gespeichert werden sollen, werden wir im Kapitel *Testdaten erzeugen* verwenden:

- In Zusatztabellen werden geeignete Feldinhalte gesammelt: eine Tabelle mit möglichen Vornamen, eine mit Nachnamen usw.
- Wie im Kapitel *Einfache Tabellenverknüpfung* beschrieben, werden als „kartesisches Produkt“ alle Zeilen und Spalten aller Zusatztabellen miteinander kombiniert.
- Oder die Zeilen und Spalten der Zusatztabellen werden per Zufallsfunktion miteinander verknüpft.

Bei wenigen Spalten der Zieltabelle kann auf Zusatztabellen verzichtet werden; dieses Verfahren nutzen wir im folgenden Beispiel. (Das Verfahren ist theoretisch auch für komplexere Tabellen möglich, wird dann aber schnell unübersichtlich; schon die folgenden CASE-Konstruktionen deuten solche Probleme an.)

► **Aufgabe:** Erzeuge eine bestimmte Anzahl von Datensätzen in der Tabelle *Fahrzeug*; die Anzahl der neuen Einträge soll als Parameter vorgegeben werden.

Firebird Quelltext

```
CREATE OR ALTER PROCEDURE Insert_Into_Fahrzeug
  ( Anzahl INTEGER = 0)
RETURNS ( Maxid INTEGER)
AS
  DECLARE VARIABLE Temp INTEGER = 0;
  DECLARE VARIABLE Listekz CHAR( 66) = 'E E E DU DU BOTRE RE OB OB GE
  GE HERHAMBO BO DO DO UN UN EN EN ';
  DECLARE VARIABLE Listekza CHAR( 26) = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  DECLARE VARIABLE Listefrb CHAR(100)
    = 'elfenbein schwarz gelb orange ocker blau
  silbern grau braun weiss ';
  DECLARE VARIABLE Kz VARCHAR( 3);
  DECLARE VARIABLE Name VARCHAR( 30);
  DECLARE VARIABLE Rand1 INTEGER = 0;
  DECLARE VARIABLE Rand2 INTEGER = 0;
  DECLARE VARIABLE Rand3 INTEGER = 0;
  DECLARE VARIABLE Rand4 INTEGER = 0;
  DECLARE VARIABLE Rand5 INTEGER = 0;
BEGIN
  Maxid = 0;
  WHILE (Temp < Anzahl) DO
  BEGIN
    /* neue Zufallszahlen */
    Rand1 = FLOOR(1 + (RAND() * 21)); /* für Kfz-Kz. eine Zufallszahl
  1 bis 22 */
    Rand2 = FLOOR(0 + (RAND() * 26)); /* ebenso eine Zufallszahl
  0 bis 26 */
    Rand3 = FLOOR(1 + (RAND() * 25)); /* ebenso eine Zufallszahl
  1 bis 26 */
    Rand4 = FLOOR(1 + (RAND() * 9)); /* für Farbe eine Zufallszahl
  1 bis 10 */
    Rand5 = FLOOR(1 + (RAND() * 22)); /* für Fz.Typ eine Zufallszahl
```

```

1 bis 23 */
/* hole per Zufall Rand1 eines der Kürzel aus Listekz */
Kz = TRIM(SUBSTRING(:Listekz from (:Rand1*3 - 2) for 3));
/* mache daraus ein vollständiges Kfz-Kennzeichen */
Name = Kz || '-' || CASE :Rand2
                        WHEN 0 THEN "
                        ELSE SUBSTRING(:Listekza from :Rand2 for 1)
                        END
      || SUBSTRING(:Listekza from :Rand3 for 1)
      || ' ' || CAST( (CASE CHAR_LENGTH(Kz)
                      WHEN 1 THEN FLOOR(1 + (RAND() *
9998))
                      ELSE FLOOR(1 + (RAND() *
998))
                      END)
                    AS INT);
/* Speichern des neuen Datensatzes */
insert into Fahrzeug (Kennzeichen, Farbe, Fahrzeugtyp_ID)
VALUES ( :Name,
        TRIM(SUBSTRING(:Listefrb from (:Rand4*10-9) for 10)),
        :Rand5 );
Temp = Temp + 1;
END
select MAX(ID) from fahrzeug into :Maxid;
SUSPEND;
END

```

Dies definiert die Prozedur mit folgenden Bestandteilen:

- Mit *Anzahl* als einzigem **Eingabe-Parameter** wird festgelegt, wie viele Datensätze erzeugt werden sollen. Dazu gehören der Name und Typ der Variablen sowie ein Vorgabewert.
- Mit *Maxid* als einzigem **Ausgabe-Parameter** wird angegeben, dass ein INTEGER-Wert mit diesem Namen erwartet wird. Der Anfangswert wird im ersten Befehl des „Rumpfes“ der Prozedur festgelegt.
- Zwischen AS und BEGIN stehen die verwendeten **Variablen** mit ihren Anfangswerten.
 - *Listekz* ist eine Liste von 22 möglichen Kfz-Kennzeichen, *Listekza* enthält das Alphabet für die Zufallssuche nach dem zweiten Teil des Kennzeichens und *Listefrb* eine Liste von 10 Farben (zu je 10 Zeichen). Diese Listen ersetzen die temporären Tabellen.
 - *Temp* wird als Zähler für die neuen Datensätze verwendet, *Kz* und *Name* als Zwischenspeicher bei der Bildung des neuen Kennzeichens.
 - *Rand1* usw. sind Variablen für Zufallszahlen. Darauf könnte auch verzichtet werden, weil die Formeln für die Zufallszahlen auch direkt in der String-Verknüpfung und im VALUES-Teil verwendet werden könnten; aber die Trennung macht es übersichtlicher.

Zwischen BEGIN und END steht der eigentliche Arbeitsablauf mit **Anweisungen** gemäß SQL-Programmierung. Innerhalb der WHILE-Schleife zwischen DO BEGIN und END wird jeweils ein neuer Datensatz erzeugt und gespeichert:

- Zuerst werden neue Zufallszahlen geholt.
- Durch Rand1 wird eine der Zahlen 1, 4, 7 usw. berechnet und gemäß diesem Wert eines der Kennzeichen aus *Listekz* geholt.
- Anschließend wird daraus ein vollständiges Kfz-Kennzeichen:
 - Wenn Rand2 ungleich 0 ist, folgt ein Buchstabe nach der Zufallszahl Rand2 aus dem Alphabet, sonst nichts.
 - Danach folgt ein weiterer Buchstabe nach der Zufallszahl Rand3 aus dem Alphabet.
 - Danach folgt eine (Zufalls-) Zahl: bei einstelligem Kennzeichen eine vierstellige Zahl, sonst eine dreistellige.
- Schließlich wird ein neuer Datensatz eingetragen mit dem eben erzeugten Kennzeichen sowie:
 - einer Farbe, die mit der Zufallszahl Rand4 aus *Listefrb* geholt wird
 - einer Fahrzeugtyp-ID, die nach der Zufallszahl Rand5 einem der vorhandenen Werte in der Tabelle *Fahrzeugtyp* entspricht

Zusätzlich wird die Variable Temp weitergezählt; nach dem Ende der Arbeit wird der größte Wert von ID bestimmt, in die Variable Maxid eingetragen und per SUSPEND als Rückgabewert übernommen.

Ausgeführt wird diese Prozedur durch einen einfachen Befehl. ► **Aufgabe:** Erzeuge 10 neue Datensätze in der Tabelle *Fahrzeug*.

```
EXECUTE PROCEDURE Insert_into_Fahrzeug (10);
```

```
----- Procedure executing results: -----  
MAXID = 145
```

31.4. Zusammenfassung

In diesem Kapitel lernten wir Prozeduren für Arbeitsabläufe kennen, die „nur“ innerhalb der Datenbank ausgeführt werden:

- Eine Prozedur kann mit oder ohne Argumenten und mit oder ohne Rückgabewerte ausgeführt werden.

- Sie dient zur automatischen Erledigung von Aufgaben, die „von außen“ angestoßen werden, aber keine zusätzlichen Maßnahmen des Anwenders oder der Anwendung benötigen.
- Bei einem solchen Arbeitsablauf werden viele Bestandteile einer Programmiersprache benutzt.

31.5. Übungen

Unter „Skizzieren“ (Übung 3, 5, 6) ist gemeint: Angeben aller Eingabe- und Ausgabeparameter sowie Variablen mit sinnvollen Namen und Datentypen, möglichst genaue Beschreibung des Arbeitsablaufs mit Pseudo-Code oder normaler Sprache. *Tipp: Die Parameter ergeben sich in der Regel aus der Aufgabenstellung. Aus der Überlegung des Arbeitsablaufs folgen die Variablen.*

Übung 1 – Prozeduren verwenden

In welchen der folgenden Situationen ist eine Prozedur sinnvoll, in welchen nicht? *Gehen Sie davon aus, dass alle benötigten Informationen in der Firmen-Datenbank gespeichert sind.*

1. Erstelle neue Rechnungen nach den aktuellen Prämiensätzen.
2. Berechne die Weihnachtsgratifikationen der Mitarbeiter nach den erfolgten Abschlüssen.
3. Ein Versicherungsvertrag wird gekündigt.

Übung 2 – Definition einer Prozedur kontrollieren

Nennen Sie in der folgenden Definition Punkte, die unabhängig vom SQL-Dialekt falsch sind. (Je nach DBMS sind noch andere Punkte falsch, danach wird aber nicht gefragt.) Die Prozedur soll folgende Aufgabe erledigen:

Mit einem Aufruf sollen bis zu 5 Abteilungen neu gespeichert werden. Die Angaben sollen wie folgt in jeweils einem String zusammengefasst werden:

```
'AbCd-Name der Abteilung-Ort der Abteilung'  
zuerst 4 Zeichen für das Kürzel, Bindestrich, der Name der  
Abteilung, Bindestrich, der Ort der Abteilung
```

In einer Schleife werden die 5 Zeichenketten verarbeitet, nämlich aufgeteilt und als Neuaufnahme in der Tabelle *Abteilung* gespeichert. Als Rückgabewert soll die letzte neu vergebene ID dienen.

```
create Insert_Abteilungen as PROCEDURE
/* Eingabe: mehrere neue Abteilungen einzutragen in der
Schreibweise */
'AbCd-Name der Abteilung-Ort der Abteilung' */
INPUTS Inhalt1, Inhalt2, Inhalt3, Inhalt4, Inhalt5 VARCHAR(70)
OUTPUTS lastid INTEGER
variables x1, pos INTEGER,
temp VARCHAR(70),
krz CHAR(4),
name, ort VARCHAR(30)
as BEGIN
x1 = 0;
WHILE (x1 < 5)
DO BEGIN
x1 = x1 + 1;
temp = CASE x1
WHEN 1 THEN Inhalt1
WHEN 2 THEN Inhalt2
WHEN 3 THEN Inhalt3
WHEN 4 THEN Inhalt4
WHEN 5 THEN Inhalt5
END
/* vorzeitiger Abbruch, falls NULL übergeben wird */
IF (temp IS NULL)
THEN break;
/* bestimme durch '-', wo der Name aufhört und der Ort anfängt */
pos = POSITION( '-', temp, 6 );
krz = SUBSTRING( temp from 1 for 4 );
name = SUBSTRING( temp from 5 for (pos-5) );
ort = SUBSTRING( temp from (pos+1) );
/* neuen Datensatz speichern */
insert into Abteilung
( Kuerzel, Bezeichnung, Ort )
values ( krz, name, ort );
lastid = SELECT ID from Abteilung WHERE Kuerzel = krz;
END
END
```

Übung 3 – Prozedur *Insert_Fahrzeugtyp* erstellen

Skizzieren Sie eine Prozedur **Insert_Fahrzeugtyp** zum Speichern von Fahrzeugtyp und Hersteller in einem Schritt: Es ist zu prüfen, ob der Hersteller schon gespeichert ist; wenn ja, ist diese ID zu verwenden, andernfalls ist ein neuer Eintrag zu erstellen und dessen ID zu übernehmen. Mit dieser Hersteller-ID ist der Fahrzeugtyp zu registrieren.

Übung 4 – Prozedur *Insert_Fahrzeugtyp* erstellen

Erstellen Sie die Prozedur **Insert_Fahrzeugtyp** aus der vorigen Übung.

Übung 5 – Prozedur *Insert_Schadensfall* erstellen

Skizzieren Sie eine Prozedur **Insert_Schadensfall** zum Speichern eines neuen Schadensfalls; dabei soll nur das Fahrzeug des Versicherungsnehmers beteiligt sein.

Übung 6 – Prozedur *Update_Schadensfall* erstellen

Skizzieren Sie eine Prozedur **Update_Schadensfall** zum Ändern eines Schadensfalls; dabei soll jeweils ein weiteres beteiligtes Fahrzeug registriert werden. (Hinweise: Sie müssen auch berücksichtigen, wie sich die Schadenshöhe neu verteilt. Sie können davon ausgehen, dass der Versicherungsnehmer schon gespeichert ist – egal, ob es sich um einen *eigenen Kunden* handelt oder nicht.) Beschreiben Sie zusätzlich, welche Punkte beim Arbeitsablauf und damit bei den Eingabe-Parametern noch geklärt werden müssen.

31.6. Lösungen

Lösung zu Übung 1 – Prozeduren verwenden

1. sinnvoll, weil es nach den gespeicherten Informationen automatisch erledigt werden kann
2. nicht sinnvoll, weil zwar die Angaben automatisch zusammengestellt werden können, aber die Gratifikation eine individuelle Entscheidung der Geschäftsleitung ist
3. sinnvoll, weil mehrere zusammenhängende Maßnahmen auszuführen sind

Lösung zu Übung 2 – Definition einer Prozedur kontrollieren

Zeile 1: Der Befehl lautet: CREATE PROCEDURE <name>.

Zeile 2/3: Der Kommentar ist falsch abgeschlossen.

Zeile 4 ff.: Die gemeinsame Deklaration von Parametern oder Variablen ist nicht zulässig.

Zeile 4: Die Eingabe-Parameter müssen in Klammern stehen.

Zeile 5: Die Ausgabe-Parameter werden bei keinem DBMS durch eine OUTPUTS-Klausel angegeben.

Zeile 6 ff.: Die Variablen folgen erst hinter AS und werden anders deklariert.

Zeile 21: Es fehlt das Semikolon am Ende der Zuweisung.

Zeile 28: Die Längenangabe im SUBSTRING ist nicht korrekt.

Zeile 31 ff.: Die Eindeutigkeit der Namen von Variablen und Spalten wird teilweise nicht beachtet.

Lösung zu Übung 3 – Prozedur *Insert_Fahrzeugtyp* erstellen

- Eingabe-Parameter: TypBezeichnung Varchar(30), HerstellerName Varchar(30), HerstellerLand Varchar(30)
- Ausgabe-Parameter: TypID Integer, HerstellerID Integer
- Variable: werden nicht benötigt
- Arbeitsablauf:
 1. Suche in der Tabelle *Fahrzeughersteller* den gegebenen *HerstellerName* und registriere die gefundene ID im Parameter *HerstellerID*.
 2. Wenn dieser Parameter jetzt den Wert NULL hat, fehlt der Eintrag noch; also muss er neu aufgenommen werden unter Verwendung der Angaben aus *HerstellerName* und *HerstellerLand* und liefert dadurch den Wert von *HerstellerID*.
 3. Damit kann der neue Datensatz in der Tabelle *Fahrzeugtyp* registriert werden; die neue ID dieses Datensatzes wird als Wert *TypID* zurückgegeben.

Lösung zu Übung 4 – Prozedur *Insert_Fahrzeugtyp* erstellen

Diese Variante der Lösung benutzt die Firebird-Syntax, vor allem hinsichtlich der Trennung von Eingabe- und Ausgabeparametern.

```
CREATE OR ALTER PROCEDURE Insert_Fahrzeugtyp
( TypBezeichnung VARCHAR(30),
```

```

        HerstellerName VARCHAR(30),
        HerstellerLand VARCHAR(30)
    )
    RETURNS( TypID INTEGER, HerstellerID INTEGER )
AS
BEGIN
    /* ist der Hersteller schon registriert? */
    select ID from Fahrzeughersteller
    where Name = :HerstellerName
    into :HerstellerID;
    /* nein, dann als Hersteller neu aufnehmen */
    IF (HerstellerID is null) THEN
    BEGIN
        HerstellerID = NEXT VALUE FOR Fahrzeughersteller_ID;
        insert into Fahrzeughersteller
            values ( :HerstellerID, :HerstellerName, :HerstellerLand );
    END
    /* anschließend den Typ registrieren */
    TypID = NEXT VALUE FOR Fahrzeugtyp_ID;
    INSERT INTO Fahrzeugtyp
        VALUES ( :TypID, :TypBezeichnung, :HerstellerID );
END

```

Lösung zu Übung 5 – Prozedur *Insert_Schadensfall* erstellen

- Eingabe-Parameter: die Angaben zum Schadensfall (Datum Date, Ort varchar(200), Beschreibung varchar(1000), Schadenshoehe number, Verletzte char(1), Mitarbeiter_ID Integer), Fahrzeugbeteiligung (ID Integer oder Kennzeichen Varchar(10), Schuldanteil Integer)
- Ausgabe-Parameter: neue ID des Schadensfalls
- Arbeitsablauf:
 1. Insert into Schadensfall: Registriere den Schadensfall, notiere die neue ID
 2. Select from Fahrzeug: Suche, sofern nötig, zum Kennzeichen die Fahrzeug-ID
 3. Insert into Zuordnung_SF_FZ: Registriere die Zuordnung zwischen Schadensfall und Fahrzeug
- Variable werden voraussichtlich nicht benötigt.

Lösung zu Übung 6 – Prozedur *Change_Schadensfall* erstellen

- Eingabe-Parameter: Schadensfall-ID Integer, Fahrzeugbeteiligung (Kennzeichen Varchar(10), anteilige Schadenshöhe Number)
- Ausgabe-Parameter: eigentlich nicht erforderlich, aber als „Erfolgsmeldung“ die ID der neuen Zuordnung
- Arbeitsablauf:

1. Select from Fahrzeug: Suche zum Kennzeichen die Fahrzeug-ID.
 2. Insert into Zuordnung_SF_FZ: Registriere die Zuordnung zwischen Schadensfall und dem neuen Fahrzeug.
 3. Übernimm die ID dieser neuen Zuordnung als Ausgabe-Parameter.
 4. Update Zuordnung_SF_FZ: Ändere im ersten Eintrag zu diesem Schadensfall die anteilige Schadenshöhe unter Berücksichtigung des neu registrierten beteiligten Fahrzeugs. (Der benötigte „erste Eintrag“ kann durch eine passende WHERE-Klausel direkt geändert werden; ersatzweise kann er auch durch einen eigenen SELECT-Befehl bestimmt werden – dann wird für die ID eine Variable benötigt.)
- Variable werden voraussichtlich nicht benötigt.
 - Unklarheiten: Bei diesem Arbeitsablauf wird davon ausgegangen, dass zu jedem später registrierten Fahrzeug ein Teil des ursprünglichen Schadens gehört; es wird nicht berücksichtigt, dass sich die Schadensverteilung insgesamt ändern kann. Völlig offen ist, wie sich der Verschuldensanteil beim ersten Eintrag und bei jedem weiteren Eintrag ändern kann. In beiden Punkten ist lediglich klar, dass alle Einzelwerte zu summieren sind und den Maximalwert (100 beim Schuldanteil bzw. den Gesamtschaden) nicht überschreiten dürfen.

31.7. Siehe auch

Teile dieses Kapitels beziehen sich auf Erläuterungen in den folgenden Kapiteln:

- SQL-PROGRAMMIERUNG¹
- VIEWS: EINE VIEW MIT VARIABLER SELEKTION²
- TESTDATEN ERZEUGEN³
- EINFACHE TABELLENVERKNÜPFUNG⁴ – alle Kombinationen aller Datensätze
- FUNKTIONEN (2)⁵ – u. a. RAND für Zufallszahlen

1 Kapitel 29 auf Seite 363

2 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20ERSTELLEN%20VON%20VIEWS%23EINE%20VIEW%20MIT%20VARIABLER%20SELEKTION](http://de.wikibooks.org/wiki/Einf%FChrung%20in%20SQL%3A%20Erstellen%20von%20Views%23Eine%20View%20mit%20variabler%20Selektion)

3 Kapitel 35 auf Seite 455

4 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20EINFACHE%20TABELLENVERKN%FCPFUNG%23ALLE%20KOMBINATIONEN%20ALLER%20DATENS%E4TZE](http://de.wikibooks.org/wiki/Einf%FChrung%20in%20SQL%3A%20Einfache%20TabelleNverkn%FCpfung%23Alle%20Kombinationen%20aller%20Datens%E4tze)

5 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20FUNKTIONEN%20%282%29%23ABS%2C%20RAND%2C%20SIGN%20%20%13%20VERSCHIEDENE%20FUNKTIONEN](http://de.wikibooks.org/wiki/Einf%FChrung%20in%20SQL%3A%20Funktionen%20%282%29%23Abs%2C%20Rand%2C%20Sign%20%20%13%20verschiedene%20Funktionen)

- TIPPS UND TRICKS⁶

Bei Wikipedia gibt es grundlegende Hinweise:

- FAT CLIENT⁷ als ein Prinzip der Datenverarbeitung

6 Kapitel 33 auf Seite 433

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/FAT%20CLIENT](http://de.wikipedia.org/wiki/Fat%20Client)

32. 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.

Bitte haben Sie Nachsicht: Wegen der vielen Varianten bei den DBMS beschränkt sich dieses Kapitel bei den Hinweisen und Beispielen weitgehend auf Firebird. Zusammen mit den speziellen Hinweisen zur SQL-Programmierung sollten sie problemlos an andere DBMS-Regeln angepasst werden können.

32.1. Ein Überblick

Schon die kurze Einleitung weist darauf hin: Ein Trigger wird niemals vom Benutzer gezielt aufgerufen, sondern immer automatisch vom DBMS erledigt; darin unterscheidet er sich wesentlich von eigenen Funktionen oder Prozeduren.

Eine deutsche Übersetzung wäre eigentlich *Auslöser*; besser passen würde „*etwas, das ausgelöst wird*“. Allenfalls spricht man noch von *Ereignisbehandlungsroutinen*. Solche Formulierungen sind aber äußerst ungebräuchlich; der Begriff *Trigger* hat sich eingebürgert.

Ein Trigger ist in folgenden Situationen nützlich:

- Werte in einer Zeile einer Tabelle sollen festgelegt werden.
 - Dies wird vor allem benutzt, wenn es keine AutoInc-Spalte gibt, siehe das Beispiel mit der nächsten ID.
- Werte sollen vor dem Speichern auf ihre Plausibilität geprüft werden.
- Veränderungen in der Datenbank sollen automatisch protokolliert werden.
- Die Regeln der referenziellen Integrität sollen mit Hilfe der Fremdschlüssel-Beziehungen überwacht werden.

32.2. Trigger definieren

32.2.1. Trigger erstellen

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

```
CREATE OR ALTER TRIGGER <routine-name> FOR <Tabellenname>
  [ ACTIVE | INACTIVE ]
  { BEFORE | AFTER }          /* bei MS-SQL heißt es INSTEAD OF
*/
  { INSERT | UPDATE | DELETE }
  [ POSITION <zahl> ]
AS
BEGIN
  [ <variablenliste> ]
  <routine body>
END
```

Auch dabei sind wieder die Besonderheiten des jeweiligen DBMS zu beachten. Aber die Ähnlichkeit zu den Routinen fällt auf.

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
Oracle kennt über die o. g. Varianten hinaus auch eine WHEN-Bedingung.
- 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. MS-SQL benutzt andere Verfahren; bei Oracle ist die Dokumentation nicht eindeutig.¹

- **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 (siehe die Beispiele).

Bei den Befehlen innerhalb des Triggers handelt es sich (eher selten) um „normale“ SQL-Befehle und überwiegend um Bestandteile der SQL-Programmiersprache. Zum Semikolon, das den Abschluss des CREATE-Befehls darstellen sollte, aber innerhalb des Inhalts bereits für jede einzelne Anweisung benutzt wird, beachten Sie bitte auch die Hinweise zur SQL-Programmierung unter *Anweisungen begrenzen*.

32.2.2. Trigger deaktivieren

Natürlich kann ein Trigger **gelöscht** werden:

```
DROP TRIGGER <name>;
```

Wenn das DBMS es ermöglicht, kann er auch vorübergehend deaktiviert werden.

```
ALTER TRIGGER <name> INACTIVE;
```

Bei ALTER sind nur die Angaben erforderlich, die sich ändern.

Einen Trigger vorübergehend abzuschalten, ist vor allem während eines längeren Arbeitsablaufs in zwei Situationen hilfreich:

- Ein Trigger bremst sehr stark; dann kann z. B. auf eine Protokollierung jeder kleinen Änderung verzichtet werden.
- Wenn Widersprüche in den Daten auftreten könnten, die sich nach vollständiger Erledigung „von selbst auflösen“, können sie durch die Deaktivierung vermieden werden.

¹ Einerseits heißt es, dass diese Variablen immer bekannt sind; andererseits sind sie sowohl in der Definition als auch in den meisten Beispielen in einer REFERENCING-Klausel aufgeführt.

32.3. Beispiele

32.3.1. Lege die nächste ID fest

Wenn ein DBMS keinen automatischen Zähler, also keine AUTOINCREMENT-Spalte ermöglicht, kann eine ID vom Anwender vergeben werden. Viel besser ist aber, wenn sich das DBMS selbst darum kümmert. Dies wird in der Beispieldatenbank bei Firebird und Oracle benutzt. Zum einen wird eine SEQUENCE definiert; der Trigger holt den nächsten Wert:

Oracle Quelltext

```
CREATE OR REPLACE TRIGGER Abteilung_BI
  BEFORE INSERT ON Abteilung
  FOR each row
  WHEN (new.ID IS NULL)
BEGIN
  SELECT Abteilung_ID.NEXTVAL INTO :new.ID FROM DUAL;
END
```

Der Trigger wird vor einem INSERT-Befehl aufgerufen. Wenn für den neuen Datensatz – repräsentiert durch die Variable *new* – keine *ID* angegeben ist, dann soll der NEXTVAL gemäß der SEQUENCE *Abteilung_ID* geholt werden.

Der Name des Triggers *Abteilung_BI* soll andeuten, dass er zur Tabelle *Abteilung* gehört und wie folgt aktiv ist: B (= Before) I (= Insert). Bei Firebird fügen wir noch eine '0' an zur Festlegung, dass er an Position null auszuführen ist.

32.3.2. Protokolliere Zeit und Benutzer bei Änderungen

In vielen Situationen ist es wichtig, dass Änderungen in einer Datenbank kontrolliert werden können. (Das DBMS macht es für sich sowieso, aber oft soll es auch „nach außen hin“ sichtbar sein.) Dafür gibt es in Tabellen Spalten wie folgt (in der Beispieldatenbank verzichten wir darauf):

```
CREATE TABLE /* usw. bis */
  Last_User    VARCHAR(30),
  Last_Change  TIMESTAMP,
  /* usw. */
```

Wenn solche Spalten bei der Tabelle *Mitarbeiter* vorhanden wären, gäbe es einen Trigger wie folgt:

```
Firebird Quelltext
```

```
CREATE OR ALTER TRIGGER Mitarbeiter_BU1 FOR Mitarbeiter
  ACTIVE BEFORE INSERT OR UPDATE POSITION 1
AS
BEGIN
  new.Last_User = CURRENT_USER;
  new.Last_Change = CURRENT_TIMESTAMP;
END
```

Ohne dass sich der Anwender darum kümmern müsste, werden Nutzer (laut Anmeldung an der Datenbank) und aktuelle Zeit immer gespeichert.

32.3.3. Aktualisiere abhängige Daten

In vielen Fällen gibt es Abhängigkeiten zwischen verschiedenen Tabellen. Durch Fremdschlüssel-Beziehungen können diese Verbindungen bei Änderungen automatisch berücksichtigt werden. Solche Anpassungen sind auch durch Trigger möglich.

Mit einer Prozedur (siehe dort) hatten wir ein INSERT in mehrere Tabellen ausgeführt. Ähnlich kann ein Trigger abhängige Datensätze löschen:

- Der Anwender ruft ein DELETE für einen Versicherungsvertrag auf.
- Der Trigger löscht den dazugehörigen Versicherungsnehmer und das Fahrzeug.
- Ein Trigger zur Tabelle *Fahrzeug* löscht alle dazu registrierten Schadensfälle usw.

Aber das ist so komplex und hat Auswirkungen auf weitere Tabellen, dass wir darauf verzichten.

In einer „echten“ Firma müssten solche Informationen sowieso dauerhaft gespeichert werden – zumindest so lange, wie die Buchhaltungsdaten aufbewahrt werden müssen (teilweise bis zu zehn Jahren). Ein Vertrag wird also im normalen Betrieb niemals gelöscht, sondern nur als „nicht mehr aktiv“ markiert.

Aber auch in unserer Firma gibt es sinnvolle Maßnahmen:

► **Aufgabe:** Beim Ausscheiden eines Mitarbeiters wird (soweit vorhanden) ein persönlicher Dienstwagen gestrichen.

Firebird Quelltext

```
create or alter trigger Mitarbeiter_BD1 for Mitarbeiter
  ACTIVE BEFORE DELETE POSITION 1
AS
BEGIN
  update Dienstwagen
    set Mitarbeiter_ID = null
    where Mitarbeiter_ID = old.ID;
END
```

Bevor ein Mitarbeiter gelöscht wird, wird geprüft, ob ihm ein persönlicher Dienstwagen zugewiesen ist. Dieser Vermerk wird (sofern vorhanden) auf NULL gesetzt; die WHERE-Klausel greift dabei auf die ID des bisherigen Datensatzes der Tabelle *Mitarbeiter* zu.

32.4. Zusammenfassung

In diesem Kapitel lernten wir Trigger als automatisch ausgeführte Prozeduren kennen:

- Ein Trigger wird vor oder nach einem Speichern-Befehl (Insert, Update, Delete) ausgeführt.
- Er dient zur automatischen Prüfung oder Vervollständigung von Werten.
- Damit vereinfacht er die Arbeit für den Anwender und
- sorgt für die Einhaltung von Sicherheitsregeln in der Datenbank.

Hinweis:

Bedenken Sie, welche konkreten Situationen in den letzten Kapiteln und den Übungen behandelt werden. Diese Zusammenhänge zwischen den Tabellen müssen bei der Planung einer Datenbank mit ihren Fremdschlüsseln, Prozeduren und Triggern für „gleichzeitiges“ Einfügen, Ändern oder Löschen in mehreren Tabellen und notwendigen Maßnahmen in anderen Tabellen berücksichtigt werden.

Auch in dieser Hinsicht bietet unsere Beispieldatenbank nur einen kleinen Einblick in das, was mit SQL möglich ist.

32.5. Übungen

Übung 1 – Trigger definieren

Welche der folgenden Aussagen sind wahr, welche sind falsch?

1. Ein Trigger wird durch eine Prozedur aufgerufen.
2. Ein Trigger hat keine Eingabe-Parameter.
3. Ein Trigger hat keine (lokalen) Variablen.
4. Ein Trigger kann nur die Tabelle bearbeiten, der er zugeordnet ist.
5. Ein Trigger kann sowohl „normale“ SQL-Befehle als auch Elemente der SQL-Programmierung enthalten.

Übung 2 – Trigger verwenden

In welchen der folgenden Situationen ist ein Trigger sinnvoll, in welchen nicht bzw. nicht möglich? *Gehen Sie davon aus, dass alle benötigten Informationen in der Firmen-Datenbank zur Speicherung vorgesehen sind.*

1. Wenn ein Versicherungsvertrag neu aufgenommen wird, sollen auch alle Datensätze in zugeordneten Tabellen neu aufgenommen werden.
2. Vorgabewerte für einzelne Spalten können eingetragen werden (wie bei einem DEFAULT-Wert).
3. Wenn eine Abteilung gelöscht (d. h. geschlossen) wird, sollen auch alle ihre Mitarbeiter gelöscht werden (weil die Abteilung ausgelagert wird).
4. Wenn ein Versicherungsvertrag gelöscht wird, sollen auch alle Datensätze in zugeordneten Tabellen bearbeitet werden: Datensätze, die noch in anderem Zusammenhang benutzt werden (z. B. Mitarbeiter), bleiben unverändert; Datensätze, die sonst nicht mehr benutzt werden (z. B. Fahrzeug), werden gelöscht.

Übung 3 – Trigger-Definition kontrollieren

Nennen Sie in der folgenden Definition Punkte, die unabhängig vom SQL-Dialekt falsch sind. Der Trigger soll folgende Aufgabe erledigen:

- Wenn ein neuer Fahrzeugtyp aufgenommen werden soll, ist der Hersteller zu überprüfen.
- Ist der angegebene Hersteller (d. h. die *Hersteller_ID*) in der Tabelle *Fahrzeughersteller* gespeichert? Wenn ja, dann ist nichts weiter zu erledigen.

- Wenn nein, dann ist ein Fahrzeughersteller 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.

Je nach DBMS sind noch andere Punkte falsch, danach wird aber nicht gefragt.

```
create Fahrzeugtyp_Check_Hersteller as trigger
to Fahrzeugtyp
for INSERT
as
DECLARE VARIABLE INTEGER fh_id
BEGIN
-- Initialisierung
fh_id = null;
-- prüfen, ob ID vorhanden ist
select ID
from Fahrzeughersteller
where ID = new.Hersteller_ID
-- wenn sie nicht gefunden wurde
if (fh_id is null) THEN
-- suche stattdessen den Hersteller 'unbekannt'
select ID
from Fahrzeughersteller
where Name = 'unbekannt'
-- wenn auch dazu die ID nicht gefunden wurde
if (fh_id is null) THEN
fh_id = new.Hersteller_ID;
insert into Fahrzeughersteller
values ( :fh_id, 'unbekannt', 'unbekannt' )
END
END
new.Hersteller_ID = :fh_id
END
```

Übung 4 – Trigger-Definition kontrollieren

Berichtigen Sie den Code der vorigen Aufgabe.

Übung 5 – Trigger *Mitarbeiter_On_Delete* erstellen

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.

Übung 6 – Trigger *Mitarbeiter_On_Delete* erstellen

Erstellen Sie den Trigger **Mitarbeiter_On_Delete** aus der vorigen Übung.

32.6. Lösungen

Lösung zu Übung 1 – Trigger definieren

Die Aussagen 2, 5 sind wahr. Die Aussagen 1, 3, 4 sind falsch.

Lösung zu Übung 2 – Trigger verwenden

1. nicht möglich, weil die Werte für die Datensätze in den zugeordneten Tabellen nicht bekannt sind
2. sinnvoll, aber die Festlegung per DEFAULT ist vorzuziehen
3. zur Not sinnvoll, wenn die Fremdschlüssel nicht durch ON DELETE geeignet gesteuert werden können; aber dies ist riskant wegen der „Nebenwirkungen“, siehe Übung 5/6
4. zur Not möglich, aber eher weniger sinnvoll wegen der vielen zugeordneten Tabellen und vieler „Nebenwirkungen“

Lösung zu Übung 3 – Trigger-Definition kontrollieren

- Zeile 1: Der Befehl lautet: CREATE TRIGGER <name>.
- Zeile 2: Die Tabelle wird je nach DBMS mit ON oder FOR angegeben, nicht mit TO.
- Zeile 3: Die Aktivierung BEFORE oder AFTER (bei MS-SQL INSTEAD OF) fehlt.
- Zeile 5: Zuerst kommt der Name der Variablen, dann der Datentyp; das abschließende Semikolon fehlt.
- Zeile 10/12 und 16/18: Es fehlt die Übergabe der ID, die durch SELECT gefunden wurde, an die Variable fh_id. *Je nach DBMS erfolgt dies unterschiedlich, aber es muss gemacht werden, damit das Ergebnis der Abfrage in der Variablen gespeichert und danach verwendet werden kann.*

- Zeile 12, 18, 23, 26: Es fehlt bei diesen Befehlen jeweils das abschließende Semikolon.
- Zeile 14 und 20: Mehrere Anweisungen sind in BEGIN...END einzuschließen.

Lösung zu Übung 4 – Trigger-Definition kontrollieren

```
-- Firebird-Version
create or alter trigger Fahrzeugtyp_Check_Hersteller
for Fahrzeugtyp
ACTIVE BEFORE INSERT POSITION 10
as
DECLARE VARIABLE fh_id INTEGER;
BEGIN
fh_id = null;
select ID
from Fahrzeughersteller
where id = new.Hersteller_ID
into :fh_id;
if (fh_id is null) THEN
BEGIN
select Id
from Fahrzeughersteller
where Name = 'unbekannt'
into :fh_id;
if (fh_id is null) THEN
BEGIN
fh_id = new.Hersteller_ID;
insert into Fahrzeughersteller
values ( :fh_id, 'unbekannt', 'unbekannt' );
END
END
new.Hersteller_ID = :fh_id;
END
```

Lösung zu Übung 5 – Trigger *Mitarbeiter_On_Delete* erstellen

- 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.

Lösung zu Übung 6 – Trigger *Mitarbeiter_On_Delete* erstellen

```

-- 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

```

32.7. Siehe auch

Verschiedene Einzelheiten stehen in folgenden Kapiteln:

- FREMDSCHLÜSSEL-BEZIEHUNGEN²
- SQL-PROGRAMMIERSPRACHE³
- PROZEDUREN⁴

Über Wikipedia sind grundlegende Informationen zu erhalten:

- DATENBANKTRIGGER⁵
- EREIGNISSE⁶ in der Programmierung, Behandlung von Ereignissen
- REFERENTIELLE INTEGRITÄT⁷

Hinweis

² Kapitel 28 auf Seite 343

³ Kapitel 29 auf Seite 363

⁴ Kapitel 31 auf Seite 399

⁵ [HTTP://DE.WIKIPEDIA.ORG/WIKI/DATENBANKTRIGGER](http://de.wikipedia.org/wiki/Datenbanktrigger)

⁶ [HTTP://DE.WIKIPEDIA.ORG/WIKI/EREIGNIS%20%28PROGRAMMIERUNG%29](http://de.wikipedia.org/wiki/Ereignis%20%28Programmierung%29)

⁷ [HTTP://DE.WIKIPEDIA.ORG/WIKI/REFERENTIELLE%20INTEGRIT%E4T](http://de.wikipedia.org/wiki/Referentielle%20Integrit%E4T)

<references />

33. Tipps und Tricks

In diesem Kapitel werden ein paar nützliche Verfahren vorgestellt, die in keines der anderen Kapitel passten und „zuwenig Stoff“ für ein eigenes Kapitel enthalten.

33.1. Die letzte ID abfragen

Wenn bei einer Tabelle für die Spalte *ID* die neuen Werte automatisch als `AUTO_INCREMENT` vergeben werden, benötigt man den neu vergebenen Wert häufig für die korrekte Behandlung der Fremdschlüssel-Beziehungen. Die verschiedenen DBMS bieten dafür unterschiedliche Verfahren.

33.1.1. Firebird: Rückgabewerte benutzen

Seit Firebird 2.x kann der `INSERT`-Befehl Maßnahmen, die durch einen Before-Insert-Trigger ausgeführt werden – also auch die Zuweisung einer Sequence – durch eine `RETURNING`-Klausel abfragen:

```
INSERT INTO <Tabelle> ( <Spaltenliste> )
    VALUES ( <Werteliste> )
    RETURNING <Spaltenliste> [ INTO <Variablenliste> ]
```

Die `INTO`-Klausel wird bei Aufgaben innerhalb der SQL-Programmierung benutzt. Beispiel für direkten Aufruf:

Firebird Quelltext

```
insert into Fahrzeug ( Kennzeichen, Fahrzeugtyp_ID )
    values ( 'B-JT 1234', 7 )
    RETURNING ID, Farbe;
```

```
----- Inserted values -----  
ID = 652  
FARBE = <null>
```

Die RETURNING-Klausel gibt es auch bei DB2 und Oracle.

33.1.2. MS-SQL: spezielle Abfragen

Je nach Situation wird nach einem INSERT in einem weiteren Befehl der neu zugeordnete Wert abgefragt.

Variante 1 mit einer lokalen Variablen:

```
SELECT @@IDENTITY
```

Dies liefert den letzten für eine IDENTITY-Spalte vergebenen Wert der aktuellen Verbindung zurück. Hierbei wird keine Tabelle angegeben; es kann aber auch der Wert einer anderen Tabelle geliefert werden (beispielsweise wenn indirekt über einen Trigger eine weitere Tabelle bearbeitet wird).

Variante 2 mit einer Funktion:

```
SELECT SCOPE_IDENTITY();
```

Dies liefert den letzten für eine IDENTITY-Spalte vergebenen Wert der aktuellen Verbindung zurück.

Variante 3 mit einer Funktion, die sich auf eine bestimmte Tabelle bezieht:

```
SELECT IDENT_CURRENT('Fahrzeug');
```

Zu den Unterschieden zwischen diesen Verfahren siehe MSDN: @@IDENTITY¹.

¹ [HTTP://MSDN.MICROSOFT.COM/DE-{}DE/LIBRARY/MS187342.ASPX](http://msdn.microsoft.com/de-{}de/library/ms187342.aspx)

33.1.3. MySQL: spezielle Abfragen

Unmittelbar nach einem INSERT wird in einem weiteren Befehl der neu zugeordnete Wert mit einer speziellen Funktion abgefragt:

```
SELECT LAST_INSERT_ID();
```

Dies liefert immer den letzten für eine AUTO_INCREMENT-Spalte vergebenen Wert der aktuellen Verbindung zurück. Hierbei wird keine Tabelle angegeben. Je nach Arbeitsumgebung gibt es auch die interne Funktion `mysql_insert_id()`, die manchmal (siehe Dokumentation) abweichende Ergebnisse liefert.

33.1.4. Oracle: Wert abfragen

In Oracle dient die SEQUENCE zur Vergabe eines automatischen Zählers. Mit `<sequence_name>.NEXTVAL` wird der nächste Wert zugewiesen, mit `<sequence_name>.CURRVAL` der aktuelle Wert abgefragt.

Im Skript zur Beispieldatenbank wird für die Tabelle *Mitarbeiter* eine SEQUENCE *Mitarbeiter_ID* definiert und verwendet. Für einen neuen Mitarbeiter erhält man so die zugewiesene ID:

Oracle Quelltext

```
INSERT INTO MITARBEITER
  ( ID, PERSONALNUMMER, NAME, VORNAME /* und weitere Angaben */ )
VALUES ( Mitarbeiter_ID.NEXTVAL, '80017', 'Schicker', 'Madelaine' );
select CONCAT('Last ID = ', to_char(Mitarbeiter_ID.CURRVAL)) from dual;
```

```
1 row inserted;
Last ID = 23
```

Weitere Informationen über Sequenzen siehe ORACLE: SEQUENZEN².

² [HTTP://DE.WIKIBOOKS.ORG/WIKI/ORACLE%3A%20SEQUENZEN](http://de.wikibooks.org/wiki/Oracle%3A%20Sequenzen)

33.2. Tabellenstruktur auslesen

Die Tabellen, Spalten, Views, Fremdschlüssel usw. werden in der Datenbank in systemeigenen Strukturen gespeichert. Sie können genauso wie die „eigenen“ Daten per SELECT abgefragt werden. Auch wenn der SQL-Standard dafür mit INFORMATION_SCHEMA ein detailliertes Muster vorgesehen hat, gibt es Unterschiede zwischen den DBMS.

33.2.1. DB2

Übersicht

Bei DB2 spricht man von Katalog-Tabellen. Sie befinden sich im Schema SYSIBM. Einige Beispiel-Zugriffe auf die Katalog-Tabellen.

```
-- Liste aller Tabellen in einer Datenbank, die von P123 erstellt
wurden.
SELECT creator, name
FROM sysibm.systables
WHERE creator = 'P123'
AND type = 'T'
--
-- Liste aller Views in einer Datenbank, die von P123 erstellt wurden.
SELECT creator, name
FROM sysibm.systables
WHERE creator = 'P123'
AND type = 'V'
--
-- Liste aller Fremdschlüssel-Beziehungen, die von P123 erstellt
wurden.
-- TBNAMe ist die (Detail-)Tabelle mit dem Fremdschlüssel
-- REFTBNAMe ist die (Master-)Tabelle, auf die der Fremdschlüssel
verweist
SELECT creator, tbnam, reftbnam
FROM sysibm.sysrels
WHERE creator = 'P123'
```

33.2.2. Firebird

Übersicht

Bei Firebird und Interbase beginnen alle Bezeichner der „Systemtabellen“ mit RDB\$.


```
-- listet die Tabellen einer Datenbank auf
SELECT rdb$relation_name from rdb$relations
where rdb$system_flag = 0
      and rdb$relation_type = 0;
-- listet die Views einer Datenbank auf
SELECT rdb$relation_name from rdb$relations
```

```
where rdb$system_flag = 0
  and rdb$relation_type = 1;
-- listet die Spaltennamen mit den dazugehörigen Datentypen einer
Tabelle auf
SELECT rdb$relation_name, rdb$field_name, rdb$field_source
  from rdb$relation_fields
 where rdb$system_flag = 0
   and rdb$view_context is null
 order by rdb$relation_name, rdb$field_position
-- listet die Fremdschlüssel einer Tabelle auf (i.d.R. nur zwischen je
einer Spalte)
SELECT rel.RDB$Constraint_Name AS ForeignKey,
       col.RDB$Relation_Name   AS DetailTable,
       CASE idx.RDB$Segment_Count
         WHEN 1 THEN f11.RDB$field_Name
         ELSE idx.RDB$Segment_Count
       END AS Fields,
       rel.RDB$Const_Name_UQ   AS PrimaryKey,
       co2.RDB$Relation_Name   AS MasterTable,
       f12.RDB$field_Name     AS MasterField
  FROM RDB$Ref_Constraints rel
/* RDB$Relation_Constraints wird 2x benötigt:
  als co1 für den Tabellennamen des ForeignKey
  als co2 für den Tabellennamen des PrimaryKey, auf den sich der
ForeignKey bezieht */
/* ebenso RDB$Index_Segments
  als f11 für den Spaltennamen des FK
  als f12 für den Spaltennamen des PK */
  JOIN RDB$Relation_Constraints col
    ON rel.RDB$Constraint_Name = col.RDB$Constraint_Name
  JOIN RDB$Indices idx
    ON rel.RDB$Constraint_Name = idx.RDB$Index_Name
  JOIN RDB$Relation_Constraints co2
    ON rel.RDB$Const_Name_UQ = co2.RDB$Constraint_Name
  JOIN RDB$Index_Segments f11
    ON rel.RDB$Constraint_Name = f11.RDB$Index_Name
  JOIN RDB$Index_Segments f12
    ON rel.RDB$Const_Name_UQ = f12.RDB$Index_Name
WHERE (NOT rel.RDB$Constraint_Name LIKE 'RDB$')
ORDER BY rel.RDB$Constraint_Name
/* Liste der Indizes
  die Zugehörigkeit zu den Tabellen und die Bedeutung
ergibt sich aus dem Namen des Index:
  PK = Primary Key
  FK = Foreign Key
  MI usw. = Foreign Key auf die Tabelle mi = Mitarbeiter
  Unter anderen Bedingungen braucht man geeignete JOINS. */
select * from RDB$Index_Segments
where rdb$index_name not starting with 'RDB$'
order by rdb$index_name, rdb$field_position
```

33.2.3. MS-SQL Server

Übersicht

```
-- listet die Tabellen einer Datenbank auf
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE TABLE'
-- listet die Views einer Datenbank auf
SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'VIEW'
-- listet die Spaltennamen mit den dazugehörigen Datentypen einer
Tabelle auf
SELECT COLUMN_NAME, DATA_TYPE FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME = 'Tabellenname'
```

33.2.4. Oracle

Übersicht

In Oracle gibt es fast alle Dictionary-Views in dreifacher Ausführung:

- Views mit dem Präfix `USER_` zeigen die eigenen Objekte an, also die Objekte, die im eigenen Schema erstellt sind.
- Views mit dem Präfix `ALL_` zeigen alle Objekte an, für die man eine Zugriffsberechtigung hat. Das sind die Objekte im eigenen Schema und auch Objekte in anderen Schemata, für die man durch den `GRANT`-Befehl eine Zugriffsberechtigung erhalten hat.
- Auf Views mit dem Präfix `DBA_` kann man nur zugreifen, wenn man das Administrations-Recht hat. In dieser View werden alle Objekte der gesamten Datenbank angezeigt, also auch die, auf die man keine Zugriffsrechte besitzt.

Alle Oracle Dictionary-Views sind im Manual Reference (nicht: SQL-Reference) beschrieben.

Beispiele:

```
-- listet alle eigenen Tabellen einer Datenbank auf.
SELECT TABLE_NAME FROM USER_TABLES
-- listet alle Tabellen auf, auf die man zugriffsberechtigt ist.
SELECT TABLE_NAME FROM ALL_TABLES
-- listet alle Tabellen auf, die es in der gesamten Datenbank gibt.
SELECT TABLE_NAME FROM DBA_TABLES
-- listet die Views auf
SELECT VIEW_NAME FROM USER_VIEWS
-- listet die Indices auf. In der Spalte UNIQUENESS ist angegeben,
-- ob es sich um einen eindeutigen Index handelt (UNIQUE)
-- oder ob die Index-Werte in der Tabelle mehrmals vorkommen dürfen
  (NONUNIQUE)
SELECT INDEX_NAME, TABLE_NAME, UNIQUENESS FROM USER_INDEXES
-- listet die Spaltennamen mit den dazugehörigen Datentypen einer
  Tabelle auf
SELECT TABLE_NAME, COLUMN_NAME, DATA_TYPE, DATA_LENGTH,
      DATA_PRECISION, DATA_SCALE, NULLABLE, COLUMN_ID
FROM USER_TAB_COLUMNS
WHERE TABLE_NAME = 'Tabellenname'
ORDER BY COLUMN_ID
-- Liste aller Fremdschlüssel-Beziehungen und anderen Constraints
-- Fremdschlüssel-Beziehungen haben den Typ 'R'
-- Bei DELETE_RULE = 'CASCADE' handelt es sich um eine Beziehung mit
  Löschweitergabe
-- bei 'NO ACTION' um eine Beziehung mit Lösch-Restriktion.
SELECT CONSTRAINT_NAME, TABLE_NAME, R_CONSTRAINT_NAME
      REFERENCED_TABLE, DELETE_RULE
FROM USER_CONSTRAINTS
WHERE CONSTRAINT_TYPE = 'R'

-- Anzeigen aller Sessions, die gerade aktiv sind oder zuletzt aktiv
  waren
SELECT * FROM V$SESSION
-- Anzeigen von Informationen über die aktuelle Datenbank-Instanz.
-- Datenbank-Name, auf welchem Server die Datenbank läuft,
  Oracle-Version,
-- seit wann die Datenbank aktiv ist, ob aktuell Logins möglich sind
-- uns in welchem Status sich die Datenbank befindet.
SELECT INSTANCE_NAME, HOST_NAME, VERSION, STARTUP_TIME, LOGINS,
      DATABASE_STATUS
FROM V$INSTANCE
```

Weitere Erläuterungen dazu stehen unter ORACLE: TABLE³; dort gibt es auch eine Anleitung, um die EINFÜGEREIHENFOLGE⁴ zu ermitteln.

33.3. Siehe auch

Weitere Erläuterungen sind in den folgenden Kapiteln zu finden:

- FREMDSCHLÜSSEL-BEZIEHUNGEN⁵
- SQL-PROGRAMMIERUNG⁶

5 Kapitel 28 auf Seite 343

6 Kapitel 29 auf Seite 363

34. Änderung der Datenbankstruktur

In diesem Kapitel werden wir die Erkenntnisse aus den vorhergehenden Kapiteln benutzen, um die Beispieldatenbank zu ergänzen und zu erweitern, wie es in der Einführung in *Anmerkungen* angesprochen wurde.

34.1. Spalten hinzufügen und ändern

Dazu wird der Befehl **ALTER TABLE ... ADD/ALTER** verwendet.

Die hier besprochenen Änderungen sowie weitere Angaben zu Spalten sind im **Skript-Spalten** zusammengefasst.

34.1.1. Neue Spalten einfügen

► **Aufgabe:** Die Tabelle *Versicherungsvertrag* benötigt mehrere zusätzliche Spalten wegen Prämienberechnung und Schadenfreiheitsrabatt.

```
alter table Versicherungsvertrag
  add [column] Basispraemie DECIMAL
      default 500 not null
      CONSTRAINT Vertrag_Basispraemie_Check check(Basispraemie > 0),
  add [column] Praemiensatz INTEGER
      default 100 not NULL
      CONSTRAINT Vertrag_Praemiensatz_Check check(Praemiensatz > 0),
  add [column] Praemienaenderung DATE;
```

Die CHECK-Prüfungen werden als CONSTRAINT mit Namen festgelegt. Ob das Wort COLUMN benötigt wird oder weggelassen werden kann oder muss, hängt (wie mehrfach gesagt) vom DBMS ab.

Der Vorgabewert wird bei den einzelnen Verträgen noch an die Versicherungsart angepasst, siehe später unter WEITERE ANPASSUNGEN¹, ebenso Prämienersatz

¹ [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23WEITERE%20ANPASSUNGEN](http://de.wikibooks.org/wiki/%23WEITERE%20ANPASSUNGEN)

und Prämienänderung auf einen einheitlichen Stand zum 1. Januar 2007. Danach werden sie durch eine **Stored Procedure** nach und nach auf den aktuellen Stand gebracht – siehe das Beispiel für eine Prozedur „automatisches UPDATE gemäß Bedingungen“.

Normalerweise sollte der Vorgabewert automatisch überall eingetragen werden. Falls das nicht geschieht, genügt ein einziger UPDATE-Befehl dafür.

34.1.2. Datentyp einer Spalte ändern

Bei dieser Aufgabe muss unbedingt darauf geachtet werden, dass die bisherigen Daten zum neuen Datentyp passen, d. h. dass sie automatisch (implizit) konvertiert werden können.

► **Aufgabe:** In der Tabelle *Abteilung* ist die Spalte *Kuerzel* als VARCHAR(10) festgelegt. Tatsächlich sind die Kürzel maximal vier Zeichen lang. Deshalb soll die Definition an die Realität angepasst werden.

MySQL Quelltext

```
alter table Abteilung
alter column Kuerzel TYPE CHAR(4);
```

Bei MySql wird dieser Befehl ausgeführt; Firebird erkennt die Möglichkeit der impliziten Konvertierung durch Abschneiden zu langer Werte nicht. (Zu lange Werte gibt es sowieso nur in der Theorie, aber nicht wirklich.)

Dann sind, wie schon bei den DDL-Einzelheiten für ALTER COLUMN erwähnt, mehrere Einzelmaßnahmen mit einer **temporären Spalte** erforderlich:

Firebird Quelltext

```
/* Erzeugen Sie eine neue, temporäre Spalte. */
alter table Abteilung add [column] TEMP VARCHAR(10);
/* Kopieren Sie alle Inhalte aus der „alten“ Spalte in die temporäre
Spalte. */
update Abteilung set temp = Kuerzel;
/* Löschen Sie die „alte“ Spalte. */
alter table Abteilung drop [column] Kuerzel;
/* Erzeugen Sie eine neue Spalte unter dem „alten“ Namen mit den „neuen“
Eigenschaften. */
alter table Abteilung add [column] Kuerzel CHAR(4) not null;
/* Kopieren Sie alle Inhalte aus der temporären Spalte in die neue
Spalte,
wobei sie passend konvertiert werden müssen. */
```



```
update Abteilung set Kuerzel = SUBSTRING(temp from 1 for 4);
/* Löschen Sie die temporäre Spalte. */
alter table Abteilung drop [column] temp;
```

34.1.3. Vorgabewert hinzufügen

Für diese Aufgabe gibt es die verschiedensten Varianten; wir beschränken uns auf eine.

► **Aufgabe:** In der Tabelle *Mitarbeiter* soll die Spalte *Ist_Leiter* in der Regel den Wert 'N' erhalten.

MySQL Quelltext

```
ALTER TABLE Abteilung MODIFY Ist_Leiter CHAR(1) NOT NULL DEFAULT 'N';
```

Bei Firebird beispielsweise kann ein Default-Wert nicht einfach hinzugefügt werden; dann ist der „Umweg“ mit einer **temporären Spalte** nötig.

Die gleiche Änderung wäre auch denkbar in der Tabelle *Versicherungsnehmer* für die Spalte *Eigener_Kunde*. Da aber nicht abgeschätzt werden kann, welcher Fall „wesentlich häufiger“ vorkommt, lohnt sich eine solche Einschränkung nicht.

34.2. Einschränkungen auf Spalten

Dazu wird der Befehl **ALTER TABLE – ADD CONSTRAINT** verwendet.

Die hier besprochenen Änderungen sowie weitere Einschränkungen sind im **Skript-Constraints** zusammengefasst.

34.2.1. Eindeutigkeit festlegen

Bei mehreren Tabellen gibt es Spalten, deren Werte eindeutig sein sollen. Dies wird durch eine **UNIQUE**-Einschränkung geregelt.

► **Aufgabe:** In der Tabelle *Mitarbeiter* ist die *Personalnummer* nicht als Primary Key vorgesehen, muss aber dennoch eindeutig sein.

```
ALTER TABLE Mitarbeiter
ADD CONSTRAINT Mitarbeiter_Nummer UNIQUE (Personalnummer)
USING INDEX Mitarbeiter_Nummer_UK;
```

Wir benutzen dafür einen CONSTRAINT mit Namen und legen außerdem fest, dass ein bestimmter Index (ebenfalls mit Namen) benutzt werden soll.

34.2.2. CHECK-Bedingungen für eine Spalte

Bei verschiedenen Spalten in mehreren Tabellen sind nur bestimmte Werte zulässig. Das wurde oben unter NEUE SPALTEN EINFÜGEN² bereits berücksichtigt, gilt aber auch für einige schon vorhandene Spalten.

► **Aufgabe:** In der Tabelle *Versicherungsvertrag* gibt es für die Spalte *Art* nur eine begrenzte Anzahl von Werten.

```
ALTER TABLE Versicherungsvertrag
ADD CONSTRAINT Versicherungsvertrag_Art
CHECK ( Art = 'HP' OR Art = 'TK' OR Art = 'VK' );
```

In gleicher Weise ist es vor allem bei den Spalten zu regeln, die als Ersatz für ein boolesches Feld nur die Werte 'J' und 'N' akzeptieren.

34.2.3. CHECK-Bedingungen für mehrere Spalten

Die bisherigen Prüfungen beziehen sich immer auf eine einzige Spalte. Genauso kann auch geprüft werden, ob die Werte mehrerer Spalten zusammenpassen. Bitte beachten Sie, dass für das folgende Beispiel die Spalte *Geschlecht* benutzt wird, die im obigen Abschnitt NEUE SPALTEN EINFÜGEN³ durch das **Skript-Spalten** eingefügt wird.

► **Aufgabe:** In der Tabelle *Versicherungsnehmer* muss es sich entweder um eine Firma handeln, d. h. *Geschlecht* IS NULL, oder eine Reihe anderer Spalten benötigt für Personen geeignete Werte.

2 [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23NEUE%20SPALTEN%20EINF%FCGEN](http://de.wikibooks.org/wiki/%23NEUE%20SPALTEN%20EINF%FCGEN)

3 [HTTP://DE.WIKIBOOKS.ORG/WIKI/%23NEUE%20SPALTEN%20EINF%FCGEN](http://de.wikibooks.org/wiki/%23NEUE%20SPALTEN%20EINF%FCGEN)

```
alter table Versicherungsnehmer
add CONSTRAINT Versicherungsnehmer_Person
CHECK( ( Geschlecht IS NULL )
or ( Vorname is not NULL
and Geburtsdatum is not null
and Fuehrerschein is not null
and Fuehrerschein >= Geburtsdatum + 365*16 )
);
```

Gleichzeitig wird geprüft, dass der Versicherungsnehmer den Führerschein frühestens im Alter von 16 Jahren (vereinfachte Berechnung, soweit möglich) erhalten haben kann.

34.3. Indizes

Dazu wird der Befehl **CREATE INDEX** verwendet.

Die hier besprochenen Indizes sowie weitere Suchschlüssel sind im **Skript-Indizes** zusammengefasst.

► **Aufgabe:** In der Tabelle *Versicherungsvertrag* ist häufig nach dem Fahrzeug, also nach *Fahrzeug_ID* zu suchen.

```
CREATE INDEX Versicherungsvertrag_FZ
ON Versicherungsvertrag (Fahrzeug_ID);
```

► **Aufgabe:** In der Tabelle *Schadensfall* ist häufig nach dem Datum zu suchen; jüngere Schadensfälle sollten zuerst kommen.

```
CREATE DESC INDEX Schadensfall_Datum
ON Schadensfall (Datum);
```

Für diesen Zweck benötigen wir also mit DESC einen absteigenden Index.

► **Aufgabe:** In der Tabelle *Versicherungsnehmer* ist häufig nach dem Namen – mit oder ohne Vorname – zu suchen.

```
CREATE INDEX Versicherungsnehmer_Name
ON Versicherungsnehmer (Name, Vorname);
```

Sie sehen, dass ein Schlüssel auch mehrere Spalten benutzen kann. Die DBMS arbeiten unterschiedlich, ob daraus für *Name* ein eigener Index wird oder ob es nur den „einheitlichen, gemeinsamen“ Index gibt.

► **Aufgabe:** In der Tabelle *Versicherungsnehmer* ist oft auch nach der Versicherungsgesellschaft zu suchen.

```
CREATE INDEX Versicherungsnehmer_Ges
ON Versicherungsnehmer (Versicherungsgesellschaft_ID);
```

Es ist nicht sicher, dass ein solcher Index eingerichtet werden kann; denn diese Spalte kann auch NULL-Werte enthalten (nämlich für die „Eigenen Kunden“), und dabei verhalten sich die DBMS unterschiedlich.

34.4. Fremdschlüssel

Dazu wird der Befehl **ALTER TABLE – ADD CONSTRAINT – FOREIGN KEY** verwendet. Alle Einzelheiten dazu wurden bei den Fremdschlüssel-Beziehungen behandelt.

Die Fremdschlüssel, die für die Beispieldatenbank vorzusehen sind, sind im **Skript-ForeignKeys** zusammengefasst.

34.5. Weitere Anpassungen

Durch die nachträglichen Änderungen in den vorstehenden Abschnitten müssen die bisherigen Daten angepasst werden. Das wollen wir jetzt erledigen.

Die hier besprochenen Anpassungen sind im **Skript-Anpassen** zusammengefasst. Dort ist eine der Versionen mit allen Werten enthalten.

34.5.1. Geschlecht ändern

In den Tabellen *Mitarbeiter* und *Versicherungsnehmer* wurde als Standardwert 'W' (= weiblich) eingetragen. Wir müssen also noch die Männer markieren; das geht durch manuelle Änderung jeder einzelnen Zeile, durch Bezug auf männliche Vornamen oder mit einer Liste der IDs von Männern. Im folgenden

Code werden alle drei Varianten notiert; sinnvoll ist es aber, nur eine davon zu benutzen.

```
update Mitarbeiter
  set Geschlecht = 'M'
  where ID = 1
     or Vorname in ('Kurt', 'Walter', 'Michael')
     or ID in (10, 11, 12);
```

Die gleiche Änderung ist in der Tabelle *Versicherungsnehmer* nötig. Mangels Vorgabewert setzen wir vorher alle Kunden, die keine Firma sind, auf 'W'.

Quelltext

Falsch

```
update Versicherungsnehmer
  set Geschlecht = 'W'
  where Vorname is not null;
```

```
Operation violates CHECK constraint on view or table.
Operation violates CHECK constraint VERSICHERUNGSNEHMER_PERSON
on view or table VERSICHERUNGSNEHMER.
```

Ach, was ist da denn wieder passiert? Bei der Firma mit ID=1 steht der *Vorname* als leere Zeichenkette, aber der oben eingerichtete CHECK mit der Prüfung „Ist Person“ vergleicht den Vornamen mit dem NULL-Wert. Also muss entweder der konkrete Wert auf NULL gesetzt werden, oder der CHECK muss geändert werden. Bei einer „echten“ Datenbank wäre der zweite Weg unbedingt vorzuziehen; wir machen es uns hier einfacher:

```
update Versicherungsnehmer
  set Vorname = null
  where Vorname = "";
```

Danach funktioniert der obige Befehl; wir müssten also wie folgt vorgehen:

- CHECK für „Ist Person“ löschen.
- CHECK für „Ist Person“ neu registrieren, sodass auch eine leere Zeichenkette für den Vornamen bei einer Firma zulässig ist.
- Geschlecht auf den Vorgabewert 'W' setzen.
- alle männlichen Personen auf 'M' ändern wie im ersten Beispiel.

34.5.2. Vorgabewert der Basisprämie

In der Tabelle *Versicherungsvertrag* wurde dieser Wert allgemein auf 500 (Euro pro Jahr) gesetzt. Damit wir später richtig damit arbeiten können, muss er natürlich von der Vertragsart abhängen. (Wir unterscheiden dabei nicht nach Fahrzeugtypen und ignorieren Prämienänderungen im Laufe der Jahre, sondern tun so, als ob sich Versicherungsprämien niemals erhöhen würden.) Da der Vorgabewert sowieso auf 500 gesetzt wurde, brauchen nur die davon abweichenden Werte geändert zu werden.

Quelltext

Falsch

```
update Versicherungsvertrag
  set Basispraemie = CASE Art
                        WHEN 'TK' THEN 550
                        WHEN 'VK' THEN 800
                        END ;
```

```
The update failed because a column definition includes validation
constraints.
validation error for column BASISPRAEMIE, value "**** null ****".
```

Natürlich, man muss aufpassen. Bei den „Nützlichen Erweiterungen“ hieß es zur CASE-Anweisung: „*wenn ELSE nicht vorhanden ist, wird NULL als Wert genommen.*“ Der bisherige Vorgabewert darf also nicht weggelassen werden:

```
update Versicherungsvertrag
  set Basispraemie = CASE Art
                        WHEN 'TK' THEN 550
                        WHEN 'VK' THEN 800
                        ELSE 500
                        END ;
```

34.5.3. Prämiensatz und Prämienänderung vorbereiten

Da sich unser Datenbestand nicht in der Wirklichkeit entwickelt hat, müssen wir diese Entwicklung simulieren. Die ersten Schadensfälle sind 2007 registriert; also muss der Datenbestand auf das Jahresende 2006 gebracht werden. Dazu benutzen wir eine Routine ohne feste Speicherung (siehe SQL-Programmierung).

Firebird Quelltext

```
EXECUTE BLOCK
AS
  DECLARE VARIABLE jj INTEGER;
  DECLARE VARIABLE T1 INTEGER;
  DECLARE VARIABLE T2 INTEGER;
BEGIN
  for SELECT vv.ID, EXTRACT(YEAR from Abschlussdatum), Praemiensatz
    from Versicherungsvertrag vv
      join Versicherungsnehmer vn on vn.Id =
vv.Versicherungsnehmer_Id
    where Praemienaenderung <= '31.12.2006'
      and vn.Eigener_kunde = 'J'
    into :T1, :jj, :T2
  DO BEGIN
    update Versicherungsvertrag
      set Praemienaenderung = DATEADD( YEAR, 2006 - :jj, Abschlussdatum
    ),
      Praemiensatz = CASE :T2
        WHEN 200 then CASE
          WHEN :jj <= 2000 THEN 80
          ELSE 200 - (2006 - :jj)*20
        END
        WHEN 100 then CASE
          WHEN :jj <= 2000 THEN 30
          ELSE 100 - (2006 - :jj)*10
        END
        ELSE :T2
      END
    where ID = :T1;
  END
END
```

Diese Berechnung kann nur für eigene Kunden ausgeführt werden. Deshalb benötigen wir die Verknüpfung mit der Tabelle *Versicherungsnehmer* und können es nicht mit einem einzigen UPDATE-Befehl erledigen.

Der Prämienatz 2006 errechnet sich aus der Dauer des Vertrags, also nach dem Abschlussdatum sowie natürlich dem bisher notierten pauschalen Prämienatz. Im UPDATE-Befehl werden für jeden Vertrag – notiert in der Variablen T1 – die neuen Werte berechnet:

- Das Datum *Praemienaenderung* wird vom Abschlussdatum aus auf das Jahr 2006 weitergesetzt.
- Der *Prämienatz* wird neu berechnet, und zwar:
 - für neuere Verträge um je 20 Punkte pro Jahr bis zu einem Minimum von 80
 - für andere Verträge um je 10 Punkte pro Jahr bis zu einem Minimum von 30
 - Werte, die von den Anfangssätzen 100 oder 200 abweichen, bleiben unverändert

34.6. Zusammenfassung

In diesem Kapitel wurden alle Änderungen, die sich für die Beispieldatenbank als sinnvoll erwiesen, besprochen. Für jede Situation wurden Beispiele angegeben; alle erforderlichen Befehle sind in Skripten zusammengefasst:

- **Skript-Spalten** zum Hinzufügen und Ändern einzelner Spalten
- **Skript-Constraints** zum Anlegen von UNIQUE KEYS und CHECK-Constraints
- **Skript-Indizes** zum Anlegen weiterer Schlüssel
- **Skript-ForeignKeys** zum Anlegen der Fremdschlüssel
- **Skript-Anpassung** zum Anpassen der betroffenen Datensätze

Weitere Hinweise dazu gibt es im Kapitel `DOWNLOADS`⁴.

34.7. Siehe auch

Die hier verwendeten Verfahren werden in den folgenden Kapiteln behandelt:

- ANMERKUNGEN ZUR BEISPIELDATENBANK⁵
- BEISPIEL EINER STOREDPROCEDURE: AUTOMATISCHES UPDATE GEMÄSS BEDINGUNGEN⁶
- DDL - EINZELHEITEN: ALTER COLUMN⁷
- FREMDSCHLÜSSEL-BEZIEHUNGEN⁸
- NÜTZLICHE ERWEITERUNGEN⁹
- SQL-PROGRAMMIERUNG: ROUTINE OHNE FESTE SPEICHERUNG¹⁰.

4 Kapitel 37 auf Seite 475

5 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20BEISPIELDATENBANK%23ANMERKUNGEN](http://de.wikibooks.org/wiki/Einf%C3%9Cfung%20in%20SQL%3A%20Beispieldatenbank%23Anmerkungen)

6 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20PROZEDUREN%23AUTOMATISCHES%20UPDATE%20GEM%E4%DF%20BEDINGUNGEN](http://de.wikibooks.org/wiki/Einf%C3%9Cfung%20in%20SQL%3A%20Prozeduren%23Automatisches%20Update%20gem%E4%DF%20Bedingungen)

7 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20DDL%20-%7D%20EINZELHEITEN%23ALTER%20COLUMN%20%20%13%20SPALTE%20%E4%NDERN](http://de.wikibooks.org/wiki/Einf%C3%9Cfung%20in%20SQL%3A%20DDL%20-%7D%20Einzelheiten%23Alter%20Column%20%20%13%20Spalte%20%E4%NDERN)

8 Kapitel 28 auf Seite 343

9 Kapitel 22 auf Seite 237

10 [HTTP://DE.WIKIBOOKS.ORG/WIKI/EINF%FCHRUNG%20IN%20SQL%3A%20PROGRAMMIERUNG%23ROUTINEN%20OHNE%20FESTE%20SPEICHERUNG](http://de.wikibooks.org/wiki/Einf%C3%9Cfung%20in%20SQL%3A%20Programmierung%23Routinen%20ohne%20Feste%20Speicherung)

34.7.1. Was ist ein Skript?

Unter einem SKRIPT¹¹ versteht man in der EDV eine Liste von Befehlen, die durch einen einzelnen Befehl aufgerufen und automatisch nacheinander ausgeführt werden. Diese Befehle sind meist in einer Datei zusammengefasst und werden dadurch ausgeführt, dass die betreffende Datei aufgerufen wird.

Bei SQL sind solche Dateien unter Windows üblicherweise mit der Endung ".sql" gespeichert.

¹¹ [HTTP://DE.WIKIPEDIA.ORG/WIKI/SKRIPT](http://de.wikipedia.org/wiki/Skript)

35. Testdaten erzeugen

In diesem Kapitel werden wir die Erkenntnisse aus den vorhergehenden Kapiteln benutzen, um die Beispieldatenbank um eine Vielzahl zusätzlicher Datensätze zu ergänzen.

Auch hier gilt: Wegen der vielen Varianten bei den DBMS beschränkt sich dieses Kapitel bei den Hinweisen und Beispielen auf Firebird. Zusammen mit den speziellen Hinweisen zur SQL-Programmierung sollten sie problemlos an andere DBMS-Regeln angepasst werden können.

35.1. Allgemeine Hinweise

Damit man in einer Datenbank viele Tests ausführen kann, sind umfangreiche, möglichst unterschiedliche Testdaten nötig. Dies wollen wir jetzt ansatzweise besprechen und dabei die bisherigen Erkenntnisse aus den vorigen Kapiteln benutzen.

Folgende Grundgedanken bei solchen Testdaten sind immer zu beachten:

- Die einzelnen Datensätze sollen zufällig erzeugte Werte erhalten.
- Die Werte in den Spalten sollen halbwegs realistisch sein: Ein Vorname wie 'Blabla' oder eine Schadenshöhe wie 0,15 € verbieten sich.
- Verschiedene Spalten müssen zusammenpassen: Zur PLZ 50667 darf es nur den Ortsnamen 'Köln' geben.
- Am besten sollte auch die Verteilung der Datensätze halbwegs realistisch sein: Es sollte mindestens so viele Fahrzeuge mit dem Kfz-Kennzeichen 'D' (Stadt Düsseldorf) wie mit 'RE' (Stadt- und Landkreis Recklinghausen) geben; ähnlich sollten auch die Versicherungsnehmer auf Stadt- und Landkreis passend verteilt sein.

 Hinweis:

Die hier genannten Schritte sind nur ein denkbare Verfahren. Wichtig ist, dass Sie einen Einblick in mögliche Vorgehensweisen und Probleme bekommen.

35.2. Neue Fahrzeuge registrieren

Für viele neue Datensätze in der Tabelle *Fahrzeug* lernten wir bereits zwei Varianten des gleichen Verfahrens kennen:

- Bei den Prozeduren werden mit einer WHILE-Schleife mehrere zufällige Werte für neue Zeilen zusammengesetzt.
- Dies geht auch als Routine ohne feste Speicherung.

Beide Varianten sind nur sinnvoll, wenn eine geringe Anzahl von Spalten mit einer geringen Anzahl möglicher Werte „vervielfältigt“ werden.

35.3. Neue Versicherungsverträge registrieren

Die Prozedur (siehe dort) *Insert_Versicherungsvertrag* ist für einen solchen automatischen Vorgang nicht geeignet, weil sie immer wieder mit neuen Werten aufgerufen werden müsste. In diesem Kapitel sollen deshalb neue Datensätze durch das „kartesische Produkt“ zusammengestellt werden. Damit die erforderlichen Beziehungen gesichert sind, müssen zuerst Fahrzeuge und Versicherungsnehmer erstellt werden; erst zuletzt dürfen die Verträge registriert werden.

Die folgenden Maßnahmen sind im **Skript-Testdaten** der [DOWNLOAD-SEITE](#)¹ zusammengefasst. Dort stehen auch diejenigen (einfacheren) SQL-Befehle, die im Folgenden nicht ausführlich angegeben und besprochen werden. Der Vermerk siehe Skript verweist bei den folgenden Schritten auf die Befehle, die in diesem Skript stehen.

1 Kapitel 37 auf Seite 475

35.3.1. Die Tabelle *Fahrzeug*

Eine **Menge von Fahrzeugen** wurde bereits durch die Prozedur *Insert_Into_Fahrzeug* gespeichert. Wir prüfen zunächst, wie viele Fahrzeuge (noch) nicht zu einem Vertrag gehören:

```
select COUNT(ID) from Fahrzeug
where ID not in (Select Fahrzeug_ID from Versicherungsvertrag);
```

```
COUNT : 120
```

Wenn das noch nicht genügt, speichern Sie jetzt mit der o. g. Prozedur weitere Datensätze. Auf diese Daten greifen wir in den nächsten Schritten zurück, damit alle Einträge zusammenpassen.

35.3.2. Hilfstabellen

Für die zufällige Verknüpfung erstellen wir uns zunächst einige Tabellen mit passenden Werten:

- *TempName* mit einer Liste von 20 Nachnamen, *TempVorname* mit einer Liste von 20 Vornamen und Geschlecht siehe Skript
- *TempPLZOrt* mit einer Liste von etwa 30 Postleitzahlen und zugehörigen Ortsnamen siehe Skript
- *TempStrasse* mit einer Liste von etwa 20 Straßennamen (eigentlich müssten diese zu den Orten passen; aber da es kein „freies“ Straßenverzeichnis für kleinere Städte und Orte gibt, soll uns das nicht interessieren) siehe Skript

35.3.3. Die Tabelle *Versicherungsnehmer*

Für eine Menge potenzieller *Versicherungsnehmer* benutzen wir eine Mischung aus Zufallsfunktionen und der direkten Verknüpfung der Daten aus den Hilfstabellen.

► **Aufgabe:** Erzeuge viele Testdaten *Versicherungsnehmer* durch das kartesische Produkt aus den Hilfstabellen.

```
Firebird Quelltext
```

```
insert into Versicherungsnehmer
  ( Name, Vorname, Geschlecht,
    PLZ, Ort,
    Strasse,
    Hausnummer,
    Geburtsdatum, Fuehrerschein, Eigener_Kunde,
    Versicherungsgesellschaft_ID )
select n.Text, v.Text, v.Geschlecht,
       p.Plz, p.Ort,
       s.Name,
       CAST( CAST( FLOOR( 1 + RAND()* 98) as INTEGER) as VARCHAR(10)),
       '01.01.1950', '31.12.2009', 'J', null
from TempName n, TempVorname v, TempPLZOrt p, TempStrasse s
where n.Text <= 'M' and v.Text <= 'P' and s.Name >= 'M';
```

Diese Anweisung kombiniert jeden Eintrag der beteiligten Tabellen mit jedem anderen Eintrag und erzeugen damit eine „Unmenge“ potenzieller Kunden (es gibt keine Verknüpfung zwischen den vier Hilfstabellen). Dies ist also das „kartesische Produkt“, das als im Normalfall ungeeignet bezeichnet wurde, aber in dieser Spezialsituation nützlich ist.

Nur für *Geburtsdatum* und *Fuehrerschein* benutzen wir nachträgliche UPDATE-Anweisungen. Der SELECT-Befehl würde zu kompliziert aussehen, wenn zunächst das Geburtsdatum mit mehreren Zufallsfunktionen erstellt würde und davon abhängig mit weiteren Zufallszahlen das Datum des Führerscheinserwerbs berechnet würde.

Achtung:

Diese Art der Verknüpfung mehrerer Tabellen kann sehr viele Datensätze erzeugen und belastet deshalb das DBMS für längere Zeit erheblich.

Nur aus diesem Grund wird die WHERE-Bedingung benutzt. Mit allen ursprünglichen Datensätzen der temporären Tabellen würden $20 \cdot 20 \cdot 47 \cdot 22$ (= 413 600) Zeilen erzeugt, was bei einem Testlauf etwa 15 Minuten dauerte. Mit der Einschränkung und einer kleineren PLZ/Orte-Tabelle gibt es etwa $12 \cdot 12 \cdot 25 \cdot 13$ (= 46 800) Zeilen in 15 Sekunden.

► **Aufgabe:** Mit den folgenden Befehlen werden zufällige Daten für *Geburtsdatum* und *Fuehrerschein* erzeugt und bei den neu erstellten Versicherungsnehmern gespeichert.

```
UPDATE Versicherungsnehmer
  SET Geburtsdatum = DATEADD( DAY,    CAST( FLOOR(RAND()*27) AS INTEGER),
                             DATEADD( MONTH, CAST( FLOOR(RAND()*11) AS INTEGER),
                             DATEADD( YEAR,  CAST( FLOOR(RAND()*40) AS INTEGER),
```

```
Geburtsdatum)))
WHERE Geburtsdatum = '01.01.1950';
COMMIT;

UPDATE Versicherungsnehmer
  SET Fuehrerschein = DATEADD( DAY,  CAST( FLOOR(RAND()*27) AS
INTEGER),
                                DATEADD( MONTH, CAST( FLOOR(RAND()*11) AS
INTEGER),
                                DATEADD( YEAR, CAST( FLOOR(18 + RAND()*(DATEDIFF
(YEAR, Geburtsdatum, CAST('31.12.2008' AS
DATE)))-18)
                                ) AS INTEGER), Geburtsdatum)))
WHERE Fuehrerschein = '31.12.2009';
COMMIT;
```

Das Geburtsdatum kann dabei nicht später als der '31.12.1990' liegen; der Führerschein kann frühestens mit dem 18. Geburtstag erworben worden sein.

35.3.4. Die Tabelle *Versicherungsvertrag*

Um die bisherigen Teildaten zu Verträgen zusammenzufassen, müssen wir anders vorgehen. Folgende Bedingungen sind zu berücksichtigen:

- Jedes Fahrzeug darf nur einmal benutzt werden.
- Ein Versicherungsnehmer darf auch in mehreren Verträgen stehen.
- Aber der Wohnort des Kunden sollte zum Kfz-Kennzeichen passen; wir machen es zur Bedingung.
- Das Abschlussdatum des Vertrags darf nur nach dem Führerschein-Erwerb liegen.
- Prämiensatz und Prämienänderung müssen ebenso vorbereitet werden wie im Kapitel *Änderung der Datenbankstruktur*.

Ein erster Versuch (zunächst beschränkt auf Fahrzeuge mit 'RE' als Kennzeichen) konnte nicht sinnvoll funktionieren:

Firebird Quelltext
Falsch

```
select fz.id, fz.Kennzeichen,
       vn.Id, vn.Name, vn.Vorname, vn.PLZ
  from Fahrzeug fz
    join Versicherungsnehmer vn on vn.ID =
      ( select FIRST 1 ID
        from Versicherungsnehmer
        where ID not in ( select Versicherungsnehmer_ID from
Versicherungsvertrag )
        and PLZ = ( select FIRST 1 PLZ
                   from TempPLZOrt
```

```
        where Kreis = 'RE'  
        order by RAND()  
    )  
    order by RAND()  
)  
where fz.Kennzeichen STARTS WITH 'RE-';
```

Das DBMS „hängt sich auf“.

Dieser Versuch sollte wie folgt vorgehen (lesen Sie den Befehl „von innen nach außen“):

- Hole nach Zufallsreihenfolge eine PLZ aus der Liste der Orte, die zum Kreis 'RE' gehören.
- Hole nach Zufallsreihenfolge einen Kunden mit dieser PLZ, sofern er noch nicht mit einem Vertrag registriert ist.
- Nur dessen Daten sollen mit einem einzelnen Fahrzeug verknüpft werden. *Wenn man auf das JOIN verzichten würde, würde der erste Versicherungsnehmer mit jedem Fahrzeug verknüpft.*

Dieses Verfahren verlangt bei jedem Fahrzeug zwei neue abhängige SELECT-Befehle, die per Zufallsfunktion aus fast 50 000 Datensätzen jeweils genau einen Datensatz liefern sollen – eine völlig unsachgemäße Belastung des DBMS.

Stattdessen wird mit einer weiteren Hilfstabelle *TempVertrag* schrittweise vorgegangen:

- Schreibe alle noch „freien“ Fahrzeuge aus den oben verwendeten Kreisen in die Hilfstabelle siehe Skript.
- Sortiere sie nach Kreis und nummeriere sie in dieser Reihenfolge siehe Skript.
- Hole per Zufallsreihenfolge – getrennt nach jedem Kreis – einen Eintrag aus der Tabelle *Versicherungsnehmer*.
- Übertrage diese Zusammenstellung in die Tabelle *Versicherungsvertrag*.
- Passe *Prämienatz* und *Prämienänderung* an (wie im früheren Kapitel ausgeführt).

Der erste, entscheidende Schritt ist die Zuordnung eines potenziellen Kunden zu jedem „freien“ Fahrzeug.

➤ **Aufgabe:** Erzeuge eine zufällige Reihenfolge der Kunden und trage sie in die (temporäre) Liste der Fahrzeuge ein.

```
Firebird Quelltext
```

```
EXECUTE BLOCK  
as  
DECLARE VARIABLE nextid INTEGER = 0;
```



```

DECLARE VARIABLE tempid INTEGER;
DECLARE VARIABLE tkreis VARCHAR(3);
DECLARE VARIABLE Tname CHAR(2);
DECLARE VARIABLE minnr INTEGER;
DECLARE VARIABLE maxnr INTEGER;
DECLARE VARIABLE Tdatum DATE;
BEGIN
  for select fz_Kreis, Min(Nr), Max(Nr)
    from TempVertrag
    group by fz_Kreis
    order by fz_Kreis
    into :tkreis, :Minnr, :Maxnr
  DO BEGIN
    /* hole alle möglichen potenziellen Kunden für diesen Kreis
    in Zufallsreihenfolge */
    nextid = :Minnr - 1;
    for select ID,
      /* diese komplizierte Konstruktion mit TRIM, CAST ist
wegen SUBSTRING nötig */
      CAST( TRIM(SUBSTRING(Name from 1 for 1))
        || TRIM(SUBSTRING(Ort from 1 for 1)) AS CHAR(2)),
      Fuehrerschein
    from Versicherungsnehmer
    where PLZ in ( select PLZ
      from TempPLZOrt
      where Kreis = :Tkreis
    )
    and ID not in
      ( select Versicherungsnehmer_ID
        from Versicherungsvertrag
      )
    order by RAND()
    into :Tempid, :Tname, :Tdatum
  DO BEGIN
    /* registriere jeden dieser Kunden nacheinander
    für eines der Fahrzeuge in diesem Kreis */
    nextid = nextid + 1;
    update TempVertrag
      set vn_ID = :Tempid,
          vn_Name = :Tname,
          /* per Zufall variable Daten vorbereiten */
          Abschlussdatum = DATEADD( DAY, CAST( FLOOR(RAND()*27) AS
INTEGER),
                                DATEADD( MONTH, CAST( FLOOR(RAND()*11) AS
INTEGER),
                                DATEADD( YEAR, CAST( FLOOR(RAND()*
(DATEDIFF (YEAR, :Tdatum,
CAST('31.12.2008' AS DATE))
                                ) AS INTEGER),
                                :Tdatum)))
      where Nr = :nextid;
    END
  END
END

```

Dieser Arbeitsablauf berücksichtigt die o. g. Grundgedanken:

- In der Hauptschleife werden die Fahrzeuge nach dem Kreis gruppiert.
 - Für jeden Kreis werden der kleinste und der größte Wert der laufenden Nummer aus der Hilfstabelle notiert.
 - Im nächsten SELECT-Befehl wird eine Schleife mit potenziellen Kunden vorbereitet:
 - Zum aktuellen Kreis werden alle Postleitzahlen aus der Hilfstabelle *Temp-PLZOrte* registriert.
 - Die Einträge der Kunden, die zu diesen PLZ gehören, werden in eine zufällige Reihenfolge gebracht.
 - In dieser Schleife wird jeder dieser Kunden bei einem Fahrzeug mit der nächsten laufenden Nummer eingetragen.
 - Der Anfangsbuchstabe vom Namen und Ort wird registriert, damit dieser Wert für die *Vertragsnummer* zur Verfügung steht.
 - Zusätzlich wird ein zufälliges Datum zwischen dem Führerschein-Erwerb und einem festen Schlussdatum erzeugt, das als *Abschlussdatum* des Vertrags zur Verfügung steht.
- **Aufgabe:** Erst jetzt wird für jede dieser Kombinationen von Fahrzeugen und Kunden, wie sie nun in der Hilfstabelle *TempVertrag* stehen, jeweils ein neuer Versicherungsvertrag erzeugt:

Firebird Quelltext

```

insert into Versicherungsvertrag
select null,                               /* ID nach
Generator */
      Vn_name || '-' || CAST( Nr AS VARCHAR(3) ), /*
Vertragsnummer */
      Abschlussdatum,                       /*
Vertragsabschluss */
      CASE MOD(Fz_id, 4)                    /* Vertragsart
nach Fahrzeug-ID */
      WHEN 0 THEN 'VK'
      WHEN 1 THEN 'HP'
      ELSE      'TK'
      END,
      CASE MOD(Fz_id, 4)                    /*
Mitarbeiter-ID nach Fahrzeug-ID */
      WHEN 0 THEN 9
      WHEN 1 THEN 10
      WHEN 2 THEN 11
      ELSE      12
      END,
      Fz_id,                               /* Fahrzeug-ID */
      Vn_id,                               /* Versicherungsnehmer-ID */
      100,                                  /* Prämienatz */
      Abschlussdatum,                      /* letzte Prämienänderung */
      CASE MOD(Fz_id, 4)                   /* Basisprämie
nach Vertragsart */

```

```
        WHEN 0 THEN 800
        WHEN 1 THEN 500
        ELSE      550
      END
FROM TempVertrag t;
```

Dabei wird jede Spalte in der Tabelle *Versicherungsvertrag* mit einem Wert versehen. Die meisten Angaben werden direkt aus der Hilfstabelle *TempVertrag* übernommen. Einige Werte werden als Ersatz für eine weitere Zufallsfunktion aus der Fahrzeug-ID errechnet.

35.4. Probleme mit Testdaten

Bei den vorstehenden Einzelschritten sind wiederholt Probleme aufgetreten. Das gilt natürlich auch für viele andere Versuche. Bitte vergessen Sie deshalb niemals diesen Ratschlag:

Hinweis

Vor jeder umfangreichen oder wichtigen Änderung ist eine Datensicherung vorzunehmen.

Das gilt fast immer für jeden einzelnen Teilschritt.

Die hier behandelten Verfahren sind beileibe nicht die einzig möglichen. Sie sollten aber sehen, wie zum einen verschiedene konstante Daten kombiniert und zum anderen Zufallswerte erzeugt werden können, um viele unterschiedliche Daten mit sinnvollen Kombinationen in verschiedenen Tabellen einzufügen. In gleicher Weise könnten wir noch viele weitere Datensätze in den Tabellen der Beispieldatenbank erzeugen.

Vor allem weitere **Schadensfälle** wären äußerst nützlich. Dabei sind aber viele Probleme zu beachten:

- Jeder Schadensfall benötigt eine Schadenssumme, die nicht nur zufällig, sondern auch sinnvoll festgelegt werden sollte.
- Die Beschreibung des Schadensfalls sollte wenigstens annähernd realistisch zur Schadenssumme passen.
- Wir benötigen Schadensfälle mit 1, 2 oder „vielen“ beteiligten Fahrzeugen. Auch deren Anzahl sowie die anteiligen Schadenssummen sollten annähernd realistisch zur Schadenssumme und zur Beschreibung passen.

- Wenn ein beteiligtes Fahrzeug (genauer: der Versicherungsnehmer) nicht zu den *eigenen Kunden* gehört, muss gleichzeitig ein Versicherungsvertrag (mit Versicherungsnehmer und Fahrzeugdaten) gespeichert werden.
- Zumindest in der Praxis gibt es auch Unfallfahrer ohne Versicherungsschutz. Wie soll man das berücksichtigen?

Eher unproblematisch sind Ergänzungen für die folgenden Tabellen; aber weil es sich in der Regel um kleinere Tabellen handelt, muss man sich darüber auch keine Gedanken machen, sondern kann jederzeit einzelne Datensätze oder kleine Prozeduren erstellen:

- Weitere **Abteilungen** sind kaum nötig; bei Bedarf macht man einzelne INSERT-Befehle.
- Weitere **Mitarbeiter** sind nur nötig, wenn die Datenbank auch für andere Arbeitsbereiche benutzt werden soll.
- Ebenso sind weitere **Dienstwagen** nicht wichtig.
- Viele weitere **Fahrzeugtypen** und **Fahrzeughersteller** wären nützlich, und zwar schon vor der Maßnahme mit den vielen neuen Versicherungsverträgen. Dafür sind aber keine zufällig entstehenden Kombinationen sinnvoll; besser ist eine einfache Prozedur: Der neue Typ und sein Hersteller werden im Klartext angegeben; wenn der Hersteller noch nicht existiert, wird er im gleichen Schritt (automatisch) registriert (siehe die Übungen zu den Prozeduren).

Insgesamt gibt es so viele praktische Probleme, die nur sehr schwer in zufällig erzeugten Datensätzen berücksichtigt werden könnten. Wir verzichten deshalb darauf.

Hinweis:

Für Ihre praktische Arbeit sollten Sie solche „Folgeprobleme“ immer beachten.

35.5. Zusammenfassung

In diesem Kapitel wurden eine Reihe Verfahren besprochen, mit denen automatisch eine Menge zusätzlicher Datensätze erstellt werden können:

- Mit dem kartesischen Produkt können aus Hilfstabellen schnell sehr viele Datensätze erzeugt werden.
- Prozeduren (oder nicht gespeicherte Routinen) sind in anderen Fällen hilfreich.

- In aller Regel sind viele Nebenbedingungen zu berücksichtigen.

35.6. Siehe auch

Dieses Kapitel benutzt Erkenntnisse der folgenden Kapitel.

- EINFACHE TABELLENVERKNÜPFUNG² zum „kartesischen Produkt“
- SQL-PROGRAMMIERUNG³ mit allgemeiner Beschreibung dazu sowie dem Abschnitt „Routinen ohne feste Speicherung“
- PROZEDUREN⁴ mit den Abschnitten „Testdaten in einer Tabelle erzeugen“, „INSERT in mehrere Tabellen“ und den Übungen
- ÄNDERUNG DER DATENBANKSTRUKTUR⁵ mit dem Abschnitt „Prämiensatz und Prämienänderung vorbereiten“

Das Skript-Testdaten – siehe DOWNLOADS⁶ – enthält alle Befehle, die für die Erzeugung der Testdaten benutzt werden.

2 Kapitel 18 auf Seite 193

3 Kapitel 29 auf Seite 363

4 Kapitel 31 auf Seite 399

5 Kapitel 34 auf Seite 443

6 Kapitel 37 auf Seite 475

Teil V.

Anhang

36. Tabellenstruktur der Beispieldatenbank

Hier stehen die Definitionen der einzelnen Tabellen; dabei ist in Klammern jeweils der Tabellen-Alias genannt.

In den folgenden Kapiteln werden Änderungen an dieser Struktur behandelt:

- ÄNDERUNG DER DATENBANKSTRUKTUR¹
- ERZEUGEN VON TESTDATEN²

In **farbiger Schrift** stehen Spalten, die durch die nachträglichen Änderungen eingefügt oder geändert werden.

Versicherungsvertrag (vv)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Vertragsnummer	varchar(20)	Pflicht	eindeutig
Abschlussdatum	date	Pflicht	
Art	char(2)	Pflicht	nur 'HP' oder 'TK' oder 'VK' möglich
Mitarbeiter_ID	integer	Pflicht	Verweis auf Tabelle <i>Mitarbeiter</i>
Fahrzeug_ID	integer	Pflicht	Verweis auf Tabelle <i>Fahrzeug</i>
Versicherungsnehmer_ID	integer	Pflicht	Verweis auf Tabelle <i>Versicherungsnehmer</i>

1 Kapitel 34 auf Seite 443

2 Kapitel 35 auf Seite 455

Basispraemie	number	Pflicht	größer als 0
Praemiensatz	integer	Pflicht	größer als 0
Praemienaenderung	date	optional	

Zuordnung_SF_FZ (zu)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Schadensfall_ID	integer	Pflicht	Verweis auf die Tabelle <i>Schadensfall</i>
Fahrzeug_ID	integer	Pflicht	Verweis auf ein einzelnes beteiligtes Fahrzeug
Schadenshoehe	number	optional	anteiliger Schaden dieses Fahrzeugs
Schuldanteil	integer	Pflicht	größer / gleich 0

Schadensfall (sf)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Datum	date	Pflicht	
Ort	varchar(200)	Pflicht	genaue Angabe einschl. Straße und Umgebung
Beschreibung	varchar(1000)	Pflicht	genaue Angabe der Umstände
Schadenshoehe	number	optional	Angabe, soweit möglich
Verletzte	char(1)	Pflicht	nur 'J' oder 'N' möglich
Mitarbeiter_ID	integer	Pflicht	Verweis auf den Sachbearbeiter in der Tabelle <i>Mitarbeiter</i>

Versicherungsnehmer (vn)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Name	varchar(30)	Pflicht	
Vorname	varchar(30)	optional	bei natürlicher Person Pflicht
Geburtsdatum	date	optional	bei natürlicher Person Pflicht
Fuehrerschein	date	optional	bei natürlicher Person Pflicht
Ort	varchar(30)	Pflicht	
PLZ	char(5)	Pflicht	
Strasse	varchar(30)	Pflicht	
Hausnummer	varchar(10)	Pflicht	
Eigener_Kunde	char(1)	Pflicht	nur 'J' oder 'N' möglich
Versicherungsgesellschaft_ID	integer	optional	Verweis auf Tabelle <i>Versicherungsgesellschaft</i> , nur bei Fremdkunden
Geschlecht	char(1)	optional	nur 'W' oder 'M' oder NULL möglich

Fahrzeug (fz)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Kennzeichen	varchar(10)	Pflicht	eindeutig
Farbe	varchar(30)	optional	
Fahrzeugtyp_ID	integer	Pflicht	Verweis auf Tabelle <i>Fahrzeugtyp</i>

Dienstwagen (dw)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Kennzeichen	varchar(10)	Pflicht	eindeutig
Farbe	varchar(30)	optional	
Fahrzeugtyp_ID	integer	Pflicht	Verweis auf Tabelle <i>Fahrzeugtyp</i>
Mitarbeiter_ID	integer	optional	ggf. Verweis auf Tabelle <i>Mitarbeiter</i>

Mitarbeiter (mi)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Personalnummer	varchar(10)	Pflicht	eindeutig
Name	varchar(30)	Pflicht	
Vorname	varchar(30)	Pflicht	
Geburtsdatum	date	Pflicht	
Telefon	varchar(30)	optional	
Mobil	varchar(30)	optional	
Email	varchar(50)	optional	
Raum	varchar(10)	optional	
Ist_Leiter	char(1)	Pflicht	nur 'J' oder 'N' möglich
Abteilung_ID	integer	Pflicht	Verweis auf Tabelle <i>Abteilung</i>
Geschlecht	char(1)	Pflicht	nur 'W' oder 'M' möglich

Versicherungsgesellschaft (vg)

Spaltenname	Datentyp	Eigenschaft
ID	integer	Primärschlüssel
Bezeichnung	varchar(30)	Pflicht
Ort	varchar(30)	optional

Fahrzeugtyp (ft)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Bezeichnung	varchar(30)	Pflicht	
Hersteller_ID	integer	Pflicht	Verweis auf Tabelle <i>Fahrzeughersteller</i>

Abteilung (ab)

Spaltenname	Datentyp	Eigenschaft	Erläuterung
ID	integer	Primärschlüssel	
Kuerzel	varchar(10)	Pflicht	Datentyp auf CHAR(4) ändern
Bezeichnung	varchar(30)	Pflicht	
Ort	varchar(30)	optional	

Fahrzeughersteller (fh)

Spaltenname	Datentyp	Eigenschaft
ID	integer	Primärschlüssel
Name	varchar(30)	Pflicht
Land	varchar(30)	optional

37. Downloads

In diesem Kapitel wird beschrieben, wie die Dateien der Download-Seite zu diesem Buch verwendet werden können.

37.1. Die Download-Seite

Verschiedene Dateien, die zur Beispieldatenbank gehören, stehen unter EINFÜHRUNG IN SQL (DOWNLOADS)¹ zur Verfügung – gewissermaßen anstelle einer Buch-CD:

- Vollständige Beispieldatenbanken
- Skripte zur Erstellung der Beispieldatenbank
- Skripte zur späteren Erweiterung

In den folgenden Abschnitten erhalten Sie Erläuterungen dazu.

37.1.1. Was ist ein Skript?

Unter einem SKRIPT² versteht man in der EDV eine Liste von Befehlen, die durch einen einzelnen Befehl aufgerufen und automatisch nacheinander ausgeführt werden. Diese Befehle sind meist in einer Datei zusammengefasst und werden dadurch ausgeführt, dass die betreffende Datei aufgerufen wird.

Bei SQL werden solche Dateien unter Windows üblicherweise mit der Endung `' .sql '` gespeichert.

1 [HTTP://WWW.VS-{}POLIS.DE/CONTENT/EINFUEHRUNG_SQL.HTML](http://www.vs-{}polis.de/content/einfuehrung_sql.html)

2 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SKRIPTSPRACHE](http://de.wikipedia.org/wiki/Skriptsprache)

37.2. Verbindung zu den Datenbanksystemen

In der Regel gibt es verschiedene Möglichkeiten: über eine Befehlszeile oder mit einem GUI-Programm. Der jeweils gängigste Weg hängt vom DBMS ab und wird in dessen Dokumentation beschrieben; im Folgenden wird nur ein Verfahren besprochen.

37.2.1. Firebird

Wenn Sie nicht mit einem GUI-Programm arbeiten, funktioniert immer das Basisprogramm `isql`. Registrieren Sie dazu zunächst ein Kürzel für die Datenbank, wie in der QuickStart-Beschreibung unter *Use database aliases* beschrieben: In der Datei `aliases.conf` im Firebird-Verzeichnis ist ein Eintrag wie folgt einzufügen:

```
# unter Windows
wb-datenbank =
C:\Users\Public\Documents\WikiBooks\SQL\Beispieldatenbank.fdb
# unter Linux
wb-datenbank = /home/wikibooks/sql/Beispieldatenbank.fdb
```

Für diese Änderung werden Administrator-Rechte benötigt. Virtuelle Laufwerke werden nicht erkannt (auch das spricht für den Eintrag eines Alias-Namens für die Datenbank). Selbstverständlich müssen Sie im Verzeichnis der Datenbank alle erforderlichen Rechte erhalten.

Öffnen Sie nun eine Command-Box und wechseln in das Verzeichnis, in dem die Datenbank stehen soll oder sich bereits befindet. Starten Sie den SQL-Interpreter:

```
C:\Programme\Firebird\bin\isql.exe
```

Jeder der folgenden Befehle muss mit einem Semikolon abgeschlossen werden. Er kann sich auch auf mehrere Zeilen verteilen.

```
--- So erstellen Sie die neue Datenbank:
SQL> create database 'wb-datenbank'
CON> user 'SYSDBA' password 'masterkey' default character set UTF8;
```



```
--- Führen Sie das Skript zur Erstellung der Datenbank aus:  
SQL> input 'Script-Firebird.sql';
```


```
--- Schließen Sie den SQL-Interpreter:  
SQL> quit;
```

```
--- So öffnen Sie die die vorbereitete oder die erzeugte Datenbank  
zur Bearbeitung:  
SQL> connect 'wb-datenbank' user 'SYSDBA' password 'masterkey';
```


37.2.2. MS-SQL

Für dieses DBMS gibt es das Programm **Microsoft SQL Server Management Studio**, und zwar für jede Server-Version eine eigene Studio-Version. Beim Programmstart melden Sie sich am Server an; die Zusammenarbeit mit der Datenbank geht über den Objekt-Explorer im Abschnitt *Databases*.

So registrieren Sie eine vorhandene Datenbank, beispielsweise die fertig vorbereitete Beispieldatenbank:

- Wählen Sie mit einem Rechtsklick auf *Databases* die Option *Attach*.
- Wählen Sie über die Schaltfläche *Add* (Hinzufügen) im oberen Teil des Fensters die betreffende mdf-Datei aus und bestätigen Sie die Auswahl mit 

So erstellen Sie die Beispieldatenbank mit dem Skript neu:

- Gehen Sie über die Menü-Befehle *File | Open | File* und wählen Sie die *CreateScript-MSSQL2005.sql* aus.
- Führen Sie das Skript mit der Schaltfläche *Execute* bzw.  aus.

In gleicher Weise können Sie zu der Beispieldatenbank spätere Skripte laden und ausführen.

37.2.3. MySQL

Hinweis: Die Datenbankstruktur wurde während der Erstellung dieses Buches überarbeitet. Für MySQL wurde auch das Skript zur Erstellung der Beispieldatenbank angepasst; es fehlen noch saubere Formulierungen in diesem Abschnitt und ein abschließender Test durch einen MySQL-Fachmann.

Melden Sie sich am Datenbanksystem an.

Erstellen Sie die neue Datenbank:

```
> create database Beispieldatenbank;
```

Geben Sie an, dass diese Datenbank benutzt werden soll:

```
> use Beispieldatenbank;
```

Führen Sie das Skript aus:

```
> source Script-MySQL.sql
```

Sofern nicht wider Erwarten ein Fehler aufgetreten ist, steht die Datenbank jetzt zur Verfügung.

37.2.4. Oracle

Hinweis: Die Datenbankstruktur wurde während der Erstellung dieses Buches überarbeitet. Für Oracle wurde auch das Skript zur Erstellung der Beispieldatenbank angepasst; es fehlen noch saubere Formulierungen in diesem Abschnitt und ein abschließender Test durch einen Oracle-Fachmann.

Für dieses DBMS gibt es das Programm gibt es das Programm DBCA „Database Configuration Assistant“. Starten Sie es mit den Rechten als Administrator des Betriebssystems.

So benutzen Sie eine vorhandene Datenbank, beispielsweise die fertig vorbereitete Beispieldatenbank:

- ???

So erstellen Sie die Beispieldatenbank mit dem Skript neu:

- Wählen Sie den Punkt *Create a Database*.
- Bei allen folgenden Schritten können Sie die einfachsten Einstellungen vornehmen.
- Im Schritt *Database Content* geben Sie das Skript `CreateScript-Oracle.sql` an.

Nach *Fertigstellen* bzw. *Finish* steht, sofern nicht wider Erwarten ein Fehler aufgetreten ist, die Datenbank zur Verfügung.

37.2.5. SQLite

Hinweis: Die Datenbankstruktur wurde während der Erstellung dieses Buches überarbeitet. Für SQLite wurde auch das Skript zur Erstellung der Beispieldatenbank angepasst; es fehlen noch saubere Formulierungen in diesem Abschnitt und ein abschließender Test durch einen SQLite-Fachmann.

Für dieses DBMS gibt es das Kommandozeilen-Tool `sqlite3`.

So benutzen Sie eine vorhandene Datenbank, beispielsweise die fertig vorbereitete Beispieldatenbank:

```
sqlite3 Beispieldatenbank.db
```

So erstellen Sie die Beispieldatenbank mit dem Skript neu:

```
sqlite3 -init CreateScript-Sqlite.sql Beispieldatenbank.db
```

37.3. Die vollständige Beispieldatenbank

Eine fertige Beispieldatenbank steht für das eine oder andere DBMS als `zip`-Datei zur Verfügung. Sie müssen also so vorgehen:

- Drücken Sie auf der Download-Seite im Abschnitt **VOLLSTÄNDIGE BEISPIEL-DATENBANK³** auf die Schaltfläche, die zu Ihrem DBMS gehört.
- Speichern Sie die `zip`-Datei an einer geeigneten Stelle, in der Regel in Ihrem Download-Bereich.
- Öffnen Sie die `zip`-Datei zum Extrahieren.
- Speichern Sie die darin enthaltenen Dateien in einem Arbeitsverzeichnis für die Beispieldatenbank.

Danach können Sie die Beispieldatenbank direkt öffnen und bearbeiten, wie es oben beim DBMS beschrieben ist.

3 [HTTP://WWW.VS-{}POLIS.DE/CONTENT/EINFUEHRUNG_SQL.HTML#3](http://www.vs-{}polis.de/content/einfuehrung_sql.html#3)

37.4. Erstellen der Beispieldatenbank

Wenn Sie die Beispieldatenbank selbst erstellen wollen, sollen oder müssen, gehen Sie bitte so vor:

- Drücken Sie auf der Download-Seite im Abschnitt SKRIPTE ZUR ERSTELLUNG⁴ auf die Schaltfläche, die zu Ihrem DBMS gehört.
- Wenn mit der linken Maustaste der Download nicht automatisch gestartet wird, dann verwenden Sie die rechte Maustaste und wählen Sie *Ziel speichern unter*.
- Speichern Sie die Datei in dem Verzeichnis (Ordner), in dem die fertige Datenbank liegen soll, z. B. unter `C:\Users\Public\Documents\WikiBooks\SQL`.
- Speichern Sie die Datei dort unter einem sinnvollen Namen, z. B. als `CreateScript-MySQL.sql` (normalerweise wird ein richtiger Name vorgegeben).

Danach können Sie die Beispieldatenbank mit Hilfe dieses Skripts erzeugen, wie es oben beim DBMS beschrieben ist.

Wenn es zu Ihrer Version des DBMS **keine vorbereitete Skript-Datei** gibt, dann holen Sie sich eine andere (möglichst für eine ähnliche Version) und gehen Sie genauso vor. In diesem Fall ist anzunehmen, dass das DBMS beim Erzeugen der Datenbank Fehler meldet. Dann müssen Sie nach der Beschreibung des betreffenden SQL-Dialekts im Skript nach und nach alle Fehler beseitigen, bis die Tabellen und Datensätze gespeichert werden.

Bitte arbeiten Sie mit!

Die Autoren und die künftigen Leser sind Ihnen sehr dankbar, wenn Sie in einem solchen Fall die fehlerfreie Fassung der `sql`-Datei und die Beispieldatenbank dazu zur Verfügung stellen, wie auf der Download-Seite beschrieben. Damit helfen Sie bei der Verbesserung und der Erweiterung dieses Buches.

⁴ [HTTP://WWW.VS-{}POLIS.DE/CONTENT/EINFUEHRUNG_SQL.HTML#4](http://www.vs-{}polis.de/content/einfuehrung_sql.html#4)

37.5. Skripte für nachträgliche Änderungen

Für verschiedene Arbeiten in den Kapiteln *ÄNDERUNG DER DATENBANKSTRUKTUR*⁵ und *TESTDATEN ERZEUGEN*⁶ stehen ebenfalls Skript-Dateien bereit.

Achtung

Diese Änderungen dürfen erst an den entsprechenden Stellen in diesen Kapiteln ausgeführt werden; vorher fehlt die notwendige Sachkenntnis. Außerdem muss – wie dort erläutert – die Reihenfolge der Änderungen beachtet werden, und vergessen Sie nicht die Datensicherung vorher und zwischendurch.

Um eines dieser Skripte zu nutzen, gehen Sie bitte so vor:

- Drücken Sie auf der Download-Seite im Abschnitt *SKRIPT ZUR SPÄTEREN ERWEITERUNG*⁷ auf die Schaltfläche zu dem gewünschten Arbeitsschritt.
- Wenn mit der linken Maustaste der Download nicht automatisch gestartet wird, dann verwenden Sie die rechte Maustaste und wählen Sie *Ziel speichern unter*.
- Speichern Sie die Datei in dem Verzeichnis (Ordner), in dem die fertige Datenbank liegt, z. B. unter `C:\Users\Public\Documents\WikiBooks\SQL`.
- Speichern Sie die Datei dort unter einem sinnvollen Namen, z. B. als `Skript-Spalten.sql` (normalerweise wird ein richtiger Name vorgegeben).
- Öffnen Sie die Skript-Datei und ändern Sie all das, was „offensichtlich“ nicht zu Ihrem SQL-Dialekt passt.

Achtung

Diese Änderungen sind teilweise sehr komplex. Gehen Sie deshalb immer schrittweise vor:

1. Machen Sie ein *Backup* der Datenbank mit dem vorherigen, fehlerfreien Bestand.
2. Führen Sie den nächsten SQL-Befehl aus (niemals das vollständige Skript!).
3. Kontrollieren Sie das Ergebnis.

5 Kapitel 34 auf Seite 443

6 Kapitel 35 auf Seite 455

7 [HTTP://WWW.VS-{}POLIS.DE/CONTENT/EINFUEHRUNG_SQL.HTML#5](http://www.vs-{}polis.de/content/einfuehrung_sql.html#5)

4. Beim Auftreten von Fehlern ist das letzte Backup zurückzuholen (*Restore*).

Anstelle der einzelnen `sql`-Dateien können Sie auch alle Dateien „am Stück“ als `zip`-Datei holen und dann die `sql`-Dateien extrahieren und im Arbeitsverzeichnis der Datenbank speichern.

38. Befehlsreferenz

Für diese Übersicht¹ gelten die gleichen Hinweise, wie sie in der EINLEITUNG² genannt sind.

Die Struktur der Befehle richtet sich nach der SQL-Dokumentation. Beispiele sind nur dann erwähnt, wenn sie in diesem Buch nicht enthalten sind, aber dennoch von Bedeutung sein können.

38.1. DDL (Data Definition Language)

Zu dieser Kategorie gehören Befehle, die die Datenbank-Objekte beschreiben. Die grundlegenden Befehle lauten:

CREATE	<object>	<name>	<details>	erzeugt ein Objekt
ALTER	<object>	<name>	<details>	ändert ein Objekt
DROP	<object>	<name>		löscht ein Objekt

38.1.1. TABLE – Tabelle bearbeiten

CREATE TABLE

Legt eine neue Tabelle an.

Die **Syntax** lautet:

1 Diese Befehlsreferenz war Teil eines nicht mehr vorhandenen Buches **SQL**. Bei der Übernahme wurden Gliederung und Struktur angepasst, Beispiele weitgehend gestrichen, weil sie sowieso Teil des Buches sind, und auf die WebLinks verzichtet, weil diese in der EINLEITUNG ^{Kapitel 2 auf Seite 11} und den WEBLINKS ^{Kapitel 41 auf Seite 503} stehen.

2 Kapitel 2 auf Seite 11

```
CREATE TABLE <tabellenname> [(<felddefinitionsliste>) | AS  
<select-statement>]
```

Syntax einer einzelnen <felddefinition>:

```
<feldbezeichnung> <feldtyp> [(<feldgröße>)] [<nullable>] [DEFAULT  
<defaultwert>]
```

Als <feldtyp> stehen die Datentypen des jeweiligen DBMS zur Verfügung, z. B. CHAR, VARCHAR, INT, FLOAT.

<nullable> gibt an, ob das Feld leer bleiben darf (NULL) oder ob ein Wert vorhanden sein muss (NOT NULL).

Als <defaultwert> kann ein Wert vorgegeben werden, der automatisch gespeichert wird, falls das Feld nicht leer bleiben darf, aber kein Wert eingegeben wurde.

<select-statement> ist eine Abfrage durch SELECT, die als Grundlage für die neue Tabelle verwendet wird.

Beispiele Eine neue Tabelle mit 2 Feldern anlegen.

```
CREATE TABLE testtab (feld1 varchar (20), feld2 number (20))
```

Eine neue Tabelle aus einem SELECT-Statement erstellen

```
CREATE TABLE testtab AS SELECT feld1, feld2 FROM testtab2
```

ALTER TABLE

verändert eine bestehende Tabelle.

Basissyntax

```
ALTER TABLE <tabellenname> <alterstatement>
```

<alterstatement> kann eins der folgenden sein:

- Tabelle umbenennen


```
RENAME TO <neuername>
```

- **Feld(er) hinzufügen**

```
ADD (<felddefinitionsliste>)
```

- **Feld ändern**

```
MODIFY <felddefinition>
```

- **Feld(er) löschen**

```
DROP (<feldliste>)
```

- **Haupt-/ Fremdschlüssel hinzufügen**

```
ADD CONSTRAINT <schlüsselname> <schlüsseldefinition>
```

- **Haupt-/ Fremdschlüssel löschen**

```
DROP PRIMARY KEY          -- Hauptschlüssel
DROP FOREIGN KEY <schlüsselname> -- Fremdschlüssel
```

- **Index erstellen**

```
ADD INDEX <indexname> [<indextyp>] (<feldliste>)
```

- **Index löschen**

```
DROP INDEX <indexname>
```

<schlüsseldefinition> kann eine der folgenden sein:

```
-- Hauptschlüssel
PRIMARY KEY (<feldliste>)
-- Fremdschlüssel
FOREIGN KEY (<feldliste>) REFERENCES <herkunftstabelle> (<feldliste>)
```

<indextyp> kann (in den meisten Fällen) folgender sein:

```
UNIQUE
PRIMARY
FULLTEXT
```

Beispiele

Tabelle umbenennen

```
ALTER, TABLE testtab RENAME TO tab2
```

Feld hinzufügen

```
ALTER TABLE testtab ADD (feld1 number(59))  
ALTER TABLE testtab ADD (feld2 date NOT NULL)  
ALTER TABLE testtab ADD (feld3 number(50) DEFAULT 10)
```

Ein Feld auf *number*, Länge 50 und NULL ändern:

```
ALTER TABLE testtab MODIFY feld1 number (50) NULL
```

Feld löschen

```
ALTER TABLE testtab DROP feld1
```

Primary-Key erstellen:

```
ALTER TABLE testtab ADD CONSTRAINT primary_key_bezeichnung PRIMARY  
KEY (feld1)
```

Primary-Key löschen

```
ALTER TABLE testtab DROP PRIMARY KEY
```

Foreign-Key erstellen:

```
ALTER TABLE testtab ADD CONSTRAINT foreign_key_bezeichnung  
FOREIGN KEY (feld1) REFERENCES tab2 (feld2)
```

Foreign-Key löschen

```
ALTER TABLE testtab DROP FOREIGN KEY foreign_key_bezeichnung
```

Index erstellen

```
ALTER TABLE testtab ADD INDEX index_name index_typ (feld1)
```

Index löschen

```
ALTER TABLE testtab DROP INDEX index_name
```

DROP TABLE

Tabelle *testtab* löschen:

```
DROP TABLE testtab
```

Tabelle *testtab* inklusive Constraints löschen:

```
DROP TABLE testtab CASCADE CONSTRAINTS
```

38.1.2. INDEX – Index bearbeiten

CREATE INDEX

```
CREATE INDEX <indexname> on <tabellenname> (<feld>);  
CREATE INDEX <indexname> on <tabellenname> (<feld1>,<feld2>,...);
```

DROP INDEX

```
DROP INDEX <indexname>
```

38.1.3. VIEW – Ansicht auf Tabellen

Einzelheiten zu VIEWS siehe ERSTELLEN VON VIEWS³.

CREATE VIEW

Eine spezielle Sicht auf eine oder mehrere Tabellen erzeugen:

```
CREATE VIEW <name> AS <select expression>
```

Als <select expression> kann ein „beliebiger“ SELECT-Befehl verwendet werden.

DROP VIEW

```
DROP VIEW <name>
```

38.2. DML – Data Manipulation Language

Diese Kategorie umfasst die Befehle, die etwas mit einem Datenbestand machen.

38.2.1. Abfragen mit SELECT

Mit dem **SELECT**-Befehl werden Daten aus einer oder mehreren Tabellen abgerufen.

```
SELECT
  [ DISTINCT | ALL ]
  <select list>
  FROM   <table reference list>
  [ WHERE <bedingungsliste> ]
  [ GROUP BY <spaltenliste> ]
  [ HAVING <bedingungsliste> ]
  [ { UNION [ALL] | INTERSECT | MINUS } <abfrage-ausdruck> ]
  [ ORDER BY <spaltenliste> ]
```

Dieser Befehl und seine Klauseln werden ausführlich unter AUSFÜHRLICHE SELECT-STRUKTUR⁴ behandelt.

38.2.2. Neuaufnahmen mit INSERT

Mit dem **INSERT**-Befehl werden Daten in einer Tabelle erstmals gespeichert. Als Syntax gibt es vor allem zwei Varianten:

```
INSERT INTO <name> [ <spaltenliste> ] VALUES ( <werteliste> )
INSERT INTO <name> [ <spaltenliste> ] SELECT <select expression>
```

Als <name> ist eine vorhandene Tabelle anzugeben. Unter bestimmten starken Einschränkungen kann auch eine VIEW angegeben werden, siehe dazu ERSTELLEN VON VIEWS⁵.

Als <spaltenliste> können eine oder mehrere Spalten der Tabelle aufgeführt werden.

- Spalten, die in dieser Liste genannt werden, bekommen einen Wert aus der <werteliste> bzw. dem SELECT-Ausdruck zugewiesen.
- Spalten, die in dieser Liste nicht genannt werden, bekommen einen DEFAULT-Wert oder NULL zugewiesen; dies hängt von der Tabellendefinition für die betreffende Spalte ab.
- Die Reihenfolge der Spalten in der <spaltenliste> muss mit der Reihenfolge der Daten in der <werteliste> bzw. dem <select expression> übereinstimmen. Die Datentypen müssen übereinstimmen oder zumindest soweit „kompatibel“ sein, dass sie automatisch (implizit) konvertiert werden können.

Wenn durch <spaltenliste> keine Spalten angegeben werden, müssen alle Spalten aus der Tabellendefinition in dieser Reihenfolge mit Inhalt versehen werden, und sei es ausdrücklich mit NULL.

Die <werteliste> besteht aus einem oder mehreren Werten (entsprechend der <spaltenliste>). Dabei kann es sich um konstante Werte oder Ergebnisse einfacher Funktionen handeln, aber auch um Abfragen. Wenn komplexe Ausdrücke als Werte zugewiesen werden sollen, ist die Variante mit SELECT zu verwenden.

Einzelheiten werden behandelt in:

4 Kapitel 14 auf Seite 145

5 Kapitel 26 auf Seite 305

- SQL-BEFEHLE⁶
- DML (2) - DATEN SPEICHERN⁷
- UNTERABFRAGEN⁸

38.2.3. Änderungen mit UPDATE

Mit dem **UPDATE**-Befehl werden Daten in einer Tabelle geändert, und zwar für eine oder mehrere Spalten in einem oder mehreren Datensätzen. Der Befehl benutzt diese Syntax:

```
UPDATE <name>
  SET <spalte1> = <wert1> [ ,
      <spalte2> = <wert2> , usw.
      <spalten> = <wertn> ]
  [ WHERE <bedingungsliste> ];
```

Als <name> ist eine vorhandene Tabelle anzugeben. Unter bestimmten starken Einschränkungen kann auch eine VIEW angegeben werden, siehe dazu ERSTELLEN VON VIEWS⁹.

Die Wertzuweisungsliste in der SET-Klausel besteht aus einer oder (mit Komma verbunden) mehreren Wertzuweisungen.

Die WHERE-Klausel arbeitet mit einer oder mehreren Bedingungen in der gleichen Weise wie beim SELECT-Befehl. Wenn keine WHERE-Bedingung angegeben wird, werden alle Zeilen geändert.

Einzelheiten werden behandelt in:

- SQL-BEFEHLE¹⁰
- DML (2) - DATEN SPEICHERN¹¹
- UNTERABFRAGEN¹²

6 Kapitel 6 auf Seite 53

7 Kapitel 8 auf Seite 75

8 Kapitel 25 auf Seite 283

9 Kapitel 26 auf Seite 305

10 Kapitel 6 auf Seite 53

11 Kapitel 8 auf Seite 75

12 Kapitel 25 auf Seite 283

38.2.4. Löschungen mit DELETE

Mit dem **DELETE**-Befehl werden Daten in einer Tabelle gelöscht, und zwar null, eine, mehrere oder alle Zeilen mit einem einzigen Befehl. Dieser benutzt die folgende Syntax:

```
DELETE FROM <name>
[ WHERE <bedingungsliste> ];
```

Als <name> ist eine vorhandene Tabelle anzugeben. Unter bestimmten starken Einschränkungen kann auch eine VIEW angegeben werden, siehe dazu ERSTELLEN VON VIEWS¹³.

Die WHERE-Klausel arbeitet mit einer oder mehreren Bedingungen in der gleichen Weise wie beim SELECT-Befehl. Wenn keine WHERE-Bedingung angegeben wird, werden alle Zeilen gelöscht.

Einzelheiten werden behandelt in:

- SQL-BEFEHLE¹⁴
- DML (2) - DATEN SPEICHERN¹⁵

38.2.5. Tabelle leeren mit TRUNCATE

Mit dem **TRUNCATE**-Befehl werden sämtliche Daten in einer Tabelle gelöscht:

```
TRUNCATE TABLE <name>;
```

Dies entspricht dem DELETE-Befehl ohne WHERE-Klausel:

```
DELETE FROM <name>;
```

Bei diesen beiden Befehlen gibt es allerdings den Unterschied, dass die gelöschten Daten nach TRUNCATE nicht mehr mit ROLLBACK wiederhergestellt werden können, da die Transaktion implizit abgeschlossen wird.

13 Kapitel 26 auf Seite 305

14 Kapitel 6 auf Seite 53

15 Kapitel 8 auf Seite 75

Da deshalb keine Protokollierung stattfindet, ist die Anweisung üblicherweise schneller als eine entsprechende DELETE-Anweisung.

Achtung: TRUNCATE TABLE kann nicht mit Foreign Keys umgehen!

38.3. TCL – Transaction Control Language

TRANSAKTIONEN¹⁶ sorgen für die Datenintegrität: Anweisungen, die logisch zusammengehören, werden entweder alle vollständig oder überhaupt nicht ausgeführt. Die TCL-Anweisungen steuern Transaktionen.

38.3.1. Bestätigen mit COMMIT

Abschließen einer Transaktion, wobei alle seit Beginn der Transaktion durchgeführten Änderungen bestätigt und endgültig gespeichert werden.

```
COMMIT;
```

Nicht alle Datenbanken unterstützen Transaktionen. Einige können auch je nach Konfiguration oder verwendetem Tabellentyp in verschiedenen Modi arbeiten. So findet im sogenannten „Auto Commit“-Modus nach Ausführung jeder DML-Anweisung sofort ein COMMIT statt, die Änderungen können also nicht mehr über ROLLBACK rückgängig gemacht werden.

38.3.2. Widerruf mit ROLLBACK

Abschließen einer Transaktion, wobei alle seit Beginn der Transaktion durchgeführten Änderungen rückgängig gemacht werden.

```
ROLLBACK;
```

Soweit das verwendete DBMS Transaktionen unterstützt, wird durch die ROLLBACK-Anweisung der Dateninhalt auf den Zustand vor Ausführung der seit dem letzten COMMIT/ROLLBACK durchgeführten Datenmanipulationen zurückgesetzt.

16 Kapitel 10 auf Seite 97

38.4. DCL – Data Control Language

Eine „vollwertige“ SQL-Datenbank enthält umfassende Regelungen über die Vergabe von Rechten für den Zugriff auf Objekte (Tabellen, einzelne Felder, interne Funktionen usw.). Am Anfang stehen diese Rechte nur dem Ersteller der Datenbank und dem System-Administrator zu. Andere Benutzer müssen ausdrücklich zu einzelnen Handlungen ermächtigt werden.

Die vielfältigen Abstufungen der Rechte und Optionen sind in der jeweiligen DBMS-Dokumentation nachzulesen.

38.4.1. GRANT

Mit diesem Befehl wird ein Recht übertragen:

```
GRANT <privileg> ON <objekt> TO <benutzer>
```

Beispiele:

- Der Benutzer Herr_Mueller darf Abfragen auf die Tabelle Abteilungen ausführen.

```
GRANT SELECT ON Abteilungen TO Herr_Mueller
```

- Die Benutzerin Frau_Schulze darf die Tabelle Abteilungen ändern.

```
GRANT UPDATE ON Abteilungen TO Frau_Schulze
```

38.4.2. REVOKE

Mit diesem Befehl wird ein Recht widerrufen:

```
REVOKE <privileg> ON <objekt> FROM <benutzer>
```

Beispiel: Herr_Mueller darf künftig keine solche Abfragen mehr ausführen.

```
REVOKE SELECT ON Abteilungen FROM Herr_Mueller
```

39. Schlüsselwörter

Die **Schlüsselwörter** sorgen dafür, dass ein DBMS in einer beliebigen Zeichenkette SQL-Befehle, weitere feste Begriffe und ergänzende Angaben unterscheiden kann.

- **Reservierte Begriffe** [res] werden für SQL-Befehle und ähnliche feststehende Angaben verwendet.
- **Nicht-reservierte Begriffe** [non] sind nach SQL-Standard eigentlich freigegeben und für spezielle Verwendung nur vorbereitet.

In der Praxis macht diese Unterscheidung keinen Sinn; in der folgenden Aufstellung wird sie deshalb nur als Hinweis [res] bzw. [non] erwähnt.



Sie sollten Schlüsselwörter niemals für eigene Bezeichner wie Tabellen, Spalten, Prozeduren usw. verwenden!

39.0.3. A

[res] ABS ALL ALLOCATE ALTER AND ANY ARE ARRAY AS ASENSITIVE
ASYMMETRIC AT ATOMIC AUTHORIZATION AVG

[non] A ABSOLUTE ACTION ADA ADD ADMIN AFTER ALWAYS ASC ASSERTION
ASSIGNMENT ATTRIBUTE ATTRIBUTES

39.0.4. B

[res] BEGIN BETWEEN BIGINT BINARY BLOB BOOLEAN BOTH BY

[non] BEFORE BERNOULLI BREADTH

39.0.5. C

[res] CALL CALLED CARDINALITY CASCADED CASE CAST CEIL CEILING
CHAR CHAR_LENGTH CHARACTER CHARACTER_LENGTH CHECK CLOB CLOSE
COALESCE COLLATE COLLECT COLUMN COMMIT CONDITION CONNECT
CONSTRAINT CONVERT CORR CORRESPONDING COUNT COVAR_POP COVAR_-
SAMP CREATE CROSS CUBE CUME_DIST CURRENT CURRENT_DATE CURRENT_-
DEFAULT_TRANSFORM_GROUP CURRENT_PATH CURRENT_ROLE CURRENT_TIME
CURRENT_TIMESTAMP CURRENT_TRANSFORM_GROUP_FOR_TYPE CURRENT_USER
CURSOR CYCLE

[non] C CASCADE CATALOG CATALOG_NAME CHAIN CHARACTER_SET_-
CATALOG CHARACTER_SET_NAME CHARACTER_SET_SCHEMA CHARACTERISTICS
CHARACTERS CLASS_ORIGIN COBOL COLLATION COLLATION_CATALOG
COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME COMMAND_FUNCTION
COMMAND_FUNCTION_CODE COMMITTED CONDITION_NUMBER CONNECTION
CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME CONSTRAINT_-
SCHEMA CONSTRAINTS CONSTRUCTOR CONTAINS CONTINUE CURSOR_NAME

39.0.6. D

[res] DATE DAY DEALLOCATE DEC DECIMAL DECLARE DEFAULT DELETE
DENSE_RANK Deref DESCRIBE DETERMINISTIC DISCONNECT DISTINCT
DOUBLE DROP DYNAMIC

[non] DATA DATETIME_INTERVAL_CODE DATETIME_INTERVAL_PRECISION
DEFAULTS DEFERRABLE DEFERRED DEFINED DEFINER DEGREE DEPTH
DERIVED DESC DESCRIPTOR DIAGNOSTICS DISPATCH DOMAIN DYNAMIC_-
FUNCTION DYNAMIC_FUNCTION_CODE

39.0.7. E

[res] EACH ELEMENT ELSE END END-EXEC ESCAPE EVERY EXCEPT EXEC
EXECUTE EXISTS EXP EXTERNAL EXTRACT

[non] EQUALS EXCEPTION EXCLUDE EXCLUDING

39.0.8. F

[res] FALSE FETCH FILTER FLOAT FLOOR FOR FOREIGN FREE FROM FULL
FUNCTION FUSION

[non] FINAL FIRST FOLLOWING FORTRAN FOUND

39.0.9. G

[res] GET GLOBAL GRANT GROUP GROUPING

[non] G GENERAL GENERATED GO GOTO GRANTED

39.0.10. H

[res] HAVING HOLD HOUR

[non] HIERARCHY

39.0.11. I

[res] IDENTITY IN INDICATOR INNER INOUT INSENSITIVE INSERT INT
INTEGER INTERSECT INTERSECTION INTERVAL INTO IS

[non] IMMEDIATE IMPLEMENTATION INCLUDING INCREMENT INITIALLY
INPUT INSTANCE INSTANTIABLE INVOKER ISOLATION

39.0.12. J

[res] JOIN

39.0.13. K

[non] K KEY KEY_MEMBER KEY_TYPE

39.0.14. L

[res] LANGUAGE LARGE LATERAL LEADING LEFT LIKE LN LOCAL
LOCALTIME LOCALTIMESTAMP LOWER

[non] LAST LENGTH LEVEL LOCATOR

39.0.15. M

[res] MATCH MAX MEMBER MERGE METHOD MIN MINUTE MOD MODIFIES
MODULE MONTH MULTISSET

[non] M MAP MATCHED MAXVALUE MESSAGE_LENGTH MESSAGE_OCTET_LENGTH
MESSAGE_TEXT MINVALUE MORE MUMPS

39.0.16. N

[res] NATIONAL NATURAL NCHAR NCLOB NEW NO NONE NORMALIZE NOT
NULL NULLIF NUMERIC

[non] NAME NAMES NESTING NEXT NORMALIZED NULLABLE NULLS NUMBER

39.0.17. O

[res] OCTET_LENGTH OF OLD ON ONLY OPEN OR ORDER OUT OUTER OVER
OVERLAPS OVERLAY

[non] OBJECT OCTETS OPTION OPTIONS ORDERING ORDINALITY OTHERS
OUTPUT OVERRIDING

39.0.18. P

[res] PARAMETER PARTITION PERCENT_RANK PERCENTILE_CONT
PERCENTILE_DISC POSITION POWER PRECISION PREPARE PRIMARY
PROCEDURE

[non] PAD PARAMETER_MODE PARAMETER_NAME PARAMETER_ORDINAL_
POSITION PARAMETER_SPECIFIC_CATALOG PARAMETER_SPECIFIC_NAME
PARAMETER_SPECIFIC_SCHEMA PARTIAL PASCAL PATH PLACING PLI
PRECEDING PRESERVE PRIOR PRIVILEGES PUBLIC

39.0.19. Q**39.0.20. R**

[res] RANGE RANK READS REAL RECURSIVE REF REFERENCES REFERENCING
 REGR_AVGX REGR_AVGY REGR_COUNT REGR_INTERCEPT REGR_R2 REGR_SLOPE
 REGR_SXX REGR_SXY REGR_SYY RELEASE RESULT RETURN RETURNS REVOKE
 RIGHT ROLLBACK ROLLUP ROW ROW_NUMBER ROWS

[non] READ RELATIVE REPEATABLE RESTART RESTRICT RETURNED_-
 CARDINALITY RETURNED_LENGTH RETURNED_OCTET_LENGTH RETURNED_-
 SQLSTATE ROLE ROUTINE ROUTINE_CATALOG ROUTINE_NAME ROUTINE_-
 SCHEMA ROW_COUNT

39.0.21. S

[res] SAVEPOINT SCOPE SCROLL SEARCH SECOND SELECT SENSITIVE
 SESSION_USER SET SIMILAR SMALLINT SOME SPECIFIC SPECIFICTYPE SQL
 SQLEXCEPTION SQLSTATE SQLWARNING SQRT START STATIC STDDEV_POP
 STDDEV_SAMP SUBMULTISET SUBSTRING SUM SYMMETRIC SYSTEM SYSTEM_-
 USER

[non] SCALE SCHEMA SCHEMA_NAME SCOPE_CATALOG SCOPE_NAME SCOPE_-
 SCHEMA SECTION SECURITY SELF SEQUENCE SERIALIZABLE SERVER_-
 NAME SESSION SETS SIMPLE SIZE SOURCE SPACE SPECIFIC_NAME STATE
 STATEMENT STRUCTURE STYLE SUBCLASS_ORIGIN

39.0.22. T

[res] TABLE TABLESAMPLE THEN TIME TIMESTAMP TIMEZONE_HOUR
 TIMEZONE_MINUTE TO TRAILING TRANSLATE TRANSLATION TREAT TRIGGER
 TRIM TRUE

[non] TABLE_NAME TEMPORARY TIES TOP_LEVEL_COUNT TRANSACTION
 TRANSACTION_ACTIVE TRANSACTIONS_COMMITTED TRANSACTIONS_ROLLED_-
 BACK TRANSFORM TRANSFORMS TRIGGER_CATALOG TRIGGER_NAME TRIGGER_-
 SCHEMA TYPE

39.0.23. U

[res] UESCAPE UNION UNIQUE UNKNOWN UNNEST UPDATE UPPER USER
USING

[non] UNBOUNDED UNCOMMITTED UNDER UNNAMED USAGE USER_DEFINED_
TYPE_CATALOG USER_DEFINED_TYPE_CODE USER_DEFINED_TYPE_NAME
USER_DEFINED_TYPE_SCHEMA

39.0.24. V

[res] VALUE VALUES VAR_POP VAR_SAMP VARCHAR VARYING

[non] VIEW

39.0.25. W

[res] WHEN WHENEVER WHERE WIDTH_BUCKET WINDOW WITH WITHIN
WITHOUT

[non] WORK WRITE

39.0.26. X

39.0.27. Y

[res] YEAR

39.0.28. Z

[non] ZONE

Diese Aufstellung wurde übernommen aus SQL-2003 DOCUMENTS¹:

- Jim Melton: Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation). August 2003. URL: http://www.wiscorp.com/sql_2003_standard.zip (abgerufen am 10. April 2009), darin enthalten 5WD-02-Foundation-2003-09.pdf pp.136

¹ [HTTP://WWW.WISCORP.COM/SQL_2003_STANDARD.ZIP](http://www.wiscorp.com/sql_2003_standard.zip)

40. Literaturverzeichnis

- Faeskorn-Woyke, Heide; Bertelsmeier, Birgit; Riemer, Petra; Bauer, Elena: *Datenbanksysteme, Theorie und Praxis mit SQL2003, Oracle und MySQL*. Pearson-Studium, München 2007, ISBN 978-3-8273-7266-6.
- Hernandez, Michael J.; Viescas, John: *Go To SQL*. Addison-Wesley, München 2001, ISBN 3-8273-1772-X.
- Kosch, Andreas: *InterBase-Datenbankentwicklung mit Delphi*. Software & Support Verlag, Frankfurt a.M. 2001, ISBN 3-935042-09-4.
- Kuhlmann, Gregor; Müllmerstadt, Friedrich: *SQL*. Rowohlt, Reinbek 2004, ISBN 3-499-61245-3.
- Marsch, Jürgen; Fritze, Jörg: *Erfolgreiche Datenbankanwendung mit SQL3. Praxisorientierte Anleitung – effizienter Einsatz – inklusive SQL-Tuning*. Vieweg + Teubner, Wiesbaden 2002, ISBN 3-528-55210-7.
- Matthiessen, Günter; Unterstein, Michael: *Relationale Datenbanken und Standard-SQL – Konzepte der Entwicklung und Anwendung*. Addison-Wesley, München 2007, ISBN 3-8273-2085-2.
- Schicker, Edwin: *Datenbanken und SQL – Eine praxisorientierte Einführung mit Hinweisen zu Oracle und MS-Access*. Vieweg + Teubner, Wiesbaden 2000, ISBN 3-519-22991-9.
- Türker, Can: *SQL 1999 & SQL 2003*. dpunkt.Verlag, Heidelberg 2003, ISBN 3-89864-219-4.
- Warner, Daniel, Günter Leitenbauer: *SQL*. Franzis, Poing 2003, ISBN 3-7723-7527-8.

41. Weblinks

Hier gibt es weitere Informationen zum Thema SQL.

41.0.29. Bücher in Wikibooks

- WIKIBOOKS: PL/SQL¹
- WIKIBOOKS: ORACLE²

41.0.30. Allgemeines zur Datenbank-Programmierung

Wikipedia bietet folgende Erläuterungen:

- SQL³
- DUALES LIZENZSYSTEM⁴
- OPEN SOURCE⁵
- CASE INSENSITIVE⁶

Außerdem gibt es Artikel zum Überblick:

- RELATIONALE DATENBANK⁷
- OBJEKTRATIONALE DATENBANK⁸
- OBJEKTORIENTIERTES DATENBANKMODELL⁹
- LISTE DER DATENBANKMANAGEMENTSYSTEME¹⁰

1 [HTTP://DE.WIKIBOOKS.ORG/WIKI/PL%2FSQL](http://de.wikibooks.org/wiki/PL%2FSQL)

2 [HTTP://DE.WIKIBOOKS.ORG/WIKI/ORACLE](http://de.wikibooks.org/wiki/Oracle)

3 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SQL](http://de.wikipedia.org/wiki/SQL)

4 [HTTP://DE.WIKIPEDIA.ORG/WIKI/MEHRFACHLIZENZIERUNG](http://de.wikipedia.org/wiki/Mehrfachlizenzierung)

5 [HTTP://DE.WIKIPEDIA.ORG/WIKI/OPEN%20SOURCE](http://de.wikipedia.org/wiki/Open%20Source)

6 [HTTP://DE.WIKIPEDIA.ORG/WIKI/CASE%20SENSITIVITY](http://de.wikipedia.org/wiki/Case%20Sensitivity)

7 [HTTP://DE.WIKIPEDIA.ORG/WIKI/RELATIONALE%20DATENBANK](http://de.wikipedia.org/wiki/Relationale%20Datenbank)

8 [HTTP://DE.WIKIPEDIA.ORG/WIKI/OBJEKTRATIONALE%20DATENBANK](http://de.wikipedia.org/wiki/Objektrationale%20Datenbank)

9 [HTTP://DE.WIKIPEDIA.ORG/WIKI/OBJEKTORIENTIERTES%20DATENBANKMODELL](http://de.wikipedia.org/wiki/Objektorientiertes%20Datenbankmodell)

10 [HTTP://DE.WIKIPEDIA.ORG/WIKI/LISTE%20DER%20DATENBANKMANAGEMENTSYSTEME](http://de.wikipedia.org/wiki/Liste%20der%20Datenbankmanagementsysteme)

41.0.31. Beiträge zu SQL allgemein

- SQL-2003 DOCUMENTS¹¹ (abgerufen am 10. April 2009), vor allem:
Jim Melton: Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation). August 2003 (als 5WD-02-Foundation-2003-09.pdf in den Dokumenten enthalten).
- Links zum Thema SQL¹² im OPEN DIRECTORY PROJECT¹³
- SQL-REFERENZ (DEUTSCH)¹⁴ (Referenz mit Stichwortverzeichnis und Tutorials)
- THE 1995 SQL REUNION: PEOPLE, PROJECTS, AND POLITICS (ZUR FRÜHEN GESCHICHTE VON SQL)¹⁵
- INTERAKTIVER SQL-TRAINER¹⁶

41.0.32. Bestimmte SQL-Datenbanksysteme

Wenn nichts anderes gesagt wird, handelt es sich um deutschsprachige Seiten.

- allgemein: Das DB2 SQL COOKBOOK VON GRAEME BIRCHALL¹⁷ (englisch) enthält viele nützliche SQL-Tipps, die auch bei anderen Datenbanken anwendbar sind.
- DB2 (IBM):
 - DB2¹⁸ (Wikipedia-Artikel)
 - DOKUMENTATION ZU DB2 V9 LUW¹⁹
 - STARTSEITE VON IBM ZU DB2 Z/OS²⁰ (englisch)
- Firebird:
 - FIREBIRD²¹

11 [HTTP://WWW.WISCORP.COM/SQL_2003_STANDARD.ZIP](http://www.wiscorp.com/sql_2003_standard.zip)

12 [HTTP://DMOZ.ORG/WORLD/DEUTSCH/COMPUTER/PROGRAMMIEREN/SPRACHEN/SQL/](http://dmoz.org/World/Deutsch/Computer/Programmieren/Sprachen/SQL/)

13 [HTTP://DE.WIKIPEDIA.ORG/WIKI/OPEN%20DIRECTORY%20PROJECT](http://de.wikipedia.org/wiki/Open%20Directory%20Project)

14 [HTTP://WWW.SQL-{}REFERENZ.DE](http://www.sql-{}referenz.de)

15 [HTTP://WWW.MCJONES.ORG/SYSTEM_R/SQL_REUNION_95/](http://www.mcjones.org/system_r/sql_reunion_95/)

16 [HTTP://EDB.GM.FH-{}KOELN.DE/SQLTRAINER/START.JSP?ACTION=WL/](http://edb.gm.fh-{}koeln.de/sqltrainer/start.jsp?action=wl/)

17 [HTTP://MYSITE.VERIZON.NET/GRAEME_BIRCHALL/ID1.HTML](http://mysite.verizon.net/Graeme_Birchall/id1.html)

18 [HTTP://DE.WIKIPEDIA.ORG/WIKI/DB2](http://de.wikipedia.org/wiki/DB2)

19 [HTTP://WWW-{}306.IBM.COM/SOFTWARE/DATA/DB2/UDB/SUPPORT/MANUALSNLVV9.HTML#DE_MAIN](http://www-{}306.ibm.com/software/data/db2/udb/support/manualsNLVv9.html#de_main)

20 [HTTP://WWW-{}306.IBM.COM/SOFTWARE/DATA/DB2/ZOS/](http://www-{}306.ibm.com/software/data/db2/zos/)

21 [HTTP://DE.WIKIPEDIA.ORG/WIKI/FIREBIRD%20%28DATENBANK%29](http://de.wikipedia.org/wiki/Firebird%20%28Datenbank%29)

- Über FIREBIRD DOCUMENTATION INDEX²² (englisch) sind viele Einzelteile zu finden – teils als HTML-Seiten, teils als PDF-Dateien.
- FIREBIRD 2.0 PSQL REFERENCE MANUAL²³ (englisch) ist eine Ergänzung für die PROGRAMMIERUNG MIT SQL²⁴.
- INFORMIX²⁵ (Wikipedia-Artikel)
- INTERBASE²⁶ (Wikipedia-Artikel)
- Microsoft SQL Server:
 - MICROSOFT SQL SERVER²⁷ (Wikipedia-Artikel)
 - SQL Server 2008: TECHNISCHE REFERENZ²⁸, vor allem im Bereich **Transact-SQL-Sprachreferenz**
 - SQL Server 2005: TRANSACT-SQL-REFERENZ²⁹
- MySQL:
 - MYSQL³⁰ (Wikipedia-Artikel)
 - MYSQL DOCUMENTATION³¹ (englisch)
 - MYSQL 5.1 REFERENZHANDBUCH³²
 - REFERENCE MANUAL 5.5³³ (englisch) für die neueste Version
- Oracle:
 - ORACLE³⁴ (Wikipedia-Artikel)
 - ORACLE DOCUMENTATION³⁵ (englisch) als Überblick, speziell beispielweise:
 - DATABASE REFERENCE 11G RELEASE 2 (11.2)³⁶
 - DATABASE REFERENCE 10G RELEASE 2 (10.2)³⁷

22 [HTTP://WWW.FIREBIRDSQL.ORG/INDEX.PHP?OP=DOC](http://www.firebirdsql.org/index.php?op=doc)

23 [HTTP://WWW.JANUS-{}SOFTWARE.COM/FBMANUAL/INDEX.PHP?BOOK=PSQL](http://www.janus-{}software.com/fbmanual/index.php?book=psql)

24 Kapitel 29 auf Seite 363

25 [HTTP://DE.WIKIPEDIA.ORG/WIKI/INFORMIX](http://de.wikipedia.org/wiki/Informix)

26 [HTTP://DE.WIKIPEDIA.ORG/WIKI/INTERBASE](http://de.wikipedia.org/wiki/Interbase)

27 [HTTP://DE.WIKIPEDIA.ORG/WIKI/MICROSOFT%20SQL%20SERVER](http://de.wikipedia.org/wiki/Microsoft%20SQL%20Server)

28 [HTTP://MSDN.MICROSOFT.COM/DE-{}DE/LIBRARY/BB500275.ASPX](http://msdn.microsoft.com/de-{}de/library/bb500275.aspx)

29 [HTTP://MSDN.MICROSOFT.COM/DE-{}DE/LIBRARY/MS189826%28SQL.90%29.ASPX](http://msdn.microsoft.com/de-{}de/library/ms189826%28SQL.90%29.aspx)

30 [HTTP://DE.WIKIPEDIA.ORG/WIKI/MYSQL](http://de.wikipedia.org/wiki/MySQL)

31 [HTTP://DEV.MYSQL.COM/DOC/](http://dev.mysql.com/doc/)

32 [HTTP://DEV.MYSQL.COM/DOC/REFMAN/5.1/DE/](http://dev.mysql.com/doc/refman/5.1/de/)

33 [HTTP://DEV.MYSQL.COM/DOC/REFMAN/5.5/EN/](http://dev.mysql.com/doc/refman/5.5/en/)

34 [HTTP://DE.WIKIPEDIA.ORG/WIKI/ORACLE%20%28DATENBANKSYSTEM%29](http://de.wikipedia.org/wiki/Oracle%20%28Datenbanksystem%29)

35 [HTTP://WWW.ORACLE.COM/TECHNOLOGY/DOCUMENTATION/INDEX.HTML](http://www.oracle.com/technology/documentation/index.html)

36 [HTTP://DOWNLOAD.ORACLE.COM/DOCS/CD/E11882_01/SERVER.112/E17110/TOC.HTM](http://download.oracle.com/docs/cd/E11882_01/server.112/e17110/toc.htm)

37 [HTTP://DOWNLOAD.ORACLE.COM/DOCS/CD/B19306_01/SERVER.102/B14237/TOC.HTM](http://download.oracle.com/docs/cd/B19306_01/server.102/B14237/toc.htm)

- PostgreSQL:
 - [POSTGRESQL³⁸](#) (Wikipedia-Artikel)
 - [DOKUMENTATION³⁹](#)
- SQLite:
 - [SQLITE⁴⁰](#) (Wikipedia-Artikel)
 - [DOKUMENTATION⁴¹](#) (englisch)
- [SYBASE⁴²](#) (Wikipedia-Artikel)

41.0.33. Andere Datenverwaltung

Über Wikipedia gibt es einführende Erläuterungen und Verweise:

- [DBASE⁴³](#)
- [MS ACCESS⁴⁴](#)
- [OPENOFFICE⁴⁵](#)
- [PARADOX⁴⁶](#)

38 [HTTP://DE.WIKIPEDIA.ORG/WIKI/POSTGRESQL](http://de.wikipedia.org/wiki/PostgreSQL)

39 [HTTP://WWW.POSTGRESQL.ORG/DOCS/BOOKS/PGHANDBUCH.HTML.DE](http://www.postgresql.org/docs/books/pghandbuch.html.de)

40 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SQLITE](http://de.wikipedia.org/wiki/SQLite)

41 [HTTP://WWW.SQLITE.ORG/DOCS.HTML](http://www.sqlite.org/docs.html)

42 [HTTP://DE.WIKIPEDIA.ORG/WIKI/SYBASE](http://de.wikipedia.org/wiki/Sybase)

43 [HTTP://DE.WIKIPEDIA.ORG/WIKI/DBASE](http://de.wikipedia.org/wiki/DBase)

44 [HTTP://DE.WIKIPEDIA.ORG/WIKI/MICROSOFT%20ACCESS](http://de.wikipedia.org/wiki/Microsoft%20Access)

45 [HTTP://DE.WIKIPEDIA.ORG/WIKI/OPENOFFICE.ORG](http://de.wikipedia.org/wiki/OpenOffice.org)

46 [HTTP://DE.WIKIPEDIA.ORG/WIKI/PARADOX%20%28DATENBANK%29](http://de.wikipedia.org/wiki/Paradox%20%28Datenbank%29)

42. Autoren

Edits	User
4	AK-PALME ¹
6	ANDORNA ²
1	BREKLA ³
1	BURNSTONE ⁴
1	C.HAHN ⁵
1	CARSRACBOT ⁶
1	COMPUTATOR ⁷
2	DANIEL B ⁸
12	DIRK HUENNIGER ⁹
2	EZHIK ¹⁰
2	FELIX REINWALD ¹¹
2	GANDI ¹²
963	JUETHO ¹³
34	JULIUS-M ¹⁴
1	KLAUS EIFERT ¹⁵
2	MARTIN FUCHS ¹⁶

1	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:AK-PALME
2	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:ANDORNA
3	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:BreKLA
4	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:BURNSTONE
5	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:C.HAHN
6	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:CARSRACBOT
7	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:COMPUTATOR
8	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:DANIEL_B
9	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:DIRK_HUENNIGER
10	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:EZHIK
11	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:FELIX_REINWALD
12	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:GANDI
13	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:JUETHO
14	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:JULIUS-M
15	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:KLAUS_EIFERT
16	HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:MARTIN_FUCHS

- 2 MATTHIASKABEL¹⁷
- 40 MCFLASH¹⁸
- 3 MJCHAE¹⁹
- 2 PHOENIX9999²⁰
- 1 SKITTLE²¹
- 1 SUNDANCE RAPHAEL²²
- 7 THEPACKER²³
- 2 TIRKON²⁴
- 16 TURELION²⁵

17 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:MATTHIASKABEL](http://de.wikibooks.org/w/index.php?title=BENUTZER:MATTHIASKABEL)

18 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:MCFLASH](http://de.wikibooks.org/w/index.php?title=BENUTZER:MCFLASH)

19 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:MJCHAE](http://de.wikibooks.org/w/index.php?title=BENUTZER:MJCHAE)

20 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:PHOENIX9999](http://de.wikibooks.org/w/index.php?title=BENUTZER:PHOENIX9999)

21 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:SKITTLE](http://de.wikibooks.org/w/index.php?title=BENUTZER:SKITTLE)

22 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:SUNDANCE_RAPHAEL](http://de.wikibooks.org/w/index.php?title=BENUTZER:SUNDANCE_RAPHAEL)

23 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:THEPACKER](http://de.wikibooks.org/w/index.php?title=BENUTZER:THEPACKER)

24 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:TIRKON](http://de.wikibooks.org/w/index.php?title=BENUTZER:TIRKON)

25 [HTTP://DE.WIKIBOOKS.ORG/W/INDEX.PHP?TITLE=BENUTZER:TURELION](http://de.wikibooks.org/w/index.php?title=BENUTZER:TURELION)

43. Bildnachweis

In der nachfolgenden Tabelle sind alle Bilder mit ihren Autoren und Lizenzen aufgelistet.

Für die Namen der Lizenzen wurden folgende Abkürzungen verwendet:

- GFDL: Gnu Free Documentation License. Der Text dieser Lizenz ist in einem Kapitel dieses Buches vollständig angegeben.
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. Der Text dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by-sa/3.0/> nachgelesen werden.
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. Der Text dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by-sa/2.5/> nachgelesen werden.
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by-sa/2.0/> nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. Der Text dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by-sa/1.0/> nachgelesen werden.
- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text der englischen Version dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by/2.0/> nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- cc-by-2.0: Creative Commons Attribution 2.0 License. Der Text dieser Lizenz kann auf der Webseite

<http://creativecommons.org/licenses/by/2.0/deed.en> nachgelesen werden.

- cc-by-2.5: Creative Commons Attribution 2.5 License. Der Text dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by/2.5/deed.en> nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- cc-by-3.0: Creative Commons Attribution 3.0 License. Der Text dieser Lizenz kann auf der Webseite <http://creativecommons.org/licenses/by/3.0/deed.en> nachgelesen werden. Mit dieser Abkürzung sind jedoch auch die Versionen dieser Lizenz für andere Sprachen bezeichnet. Den an diesen Details interessierten Leser verweisen wir auf die Onlineversion dieses Buches.
- GPL: GNU General Public License Version 2. Der Text dieser Lizenz kann auf der Webseite <http://www.gnu.org/licenses/gpl-2.0.txt> nachgelesen werden.
- PD: This image is in the public domain. Dieses Bild ist gemeinfrei.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. Der Text dieser Lizenz kann auf der Webseite <http://artlibre.org/licence/lal/de> nachgelesen werden.
- CFR: Copyright free use. Der Urheberrechtsinhaber erlaubt es jedem, dieses Bild für jeglichen Zweck, inklusive uneingeschränkter Weiterveröffentlichung, kommerziellem Gebrauch und Modifizierung, zu nutzen.
- EPL: Eclipse Public License. Der Text dieser Lizenz kann auf der Webseite <http://www.eclipse.org/org/documents/epl-v10.php> nachgelesen werden.

Bild	Autor	Lizenz
1		
2		PD
3		PD
4		PD
5		PD
6		PD
7		PD
8		PD
9		PD