1 Vorwort

Das vorliegende Dokument dient als Referenz zum Erlernen von SQL. Da es für schulische Zwecke eingesetzt wird, liegt der Fokus bei den RDBMS auf MySQL[1]. Im Unterricht wird primär eine mobile Version genutzt, welche als „USBWebserver“ unter www.usbwebserver.net geladen werden kann.

Unter www.CodeConcert.de kann zusätzlich noch eine Übungsdatenbank heruntergeladen werden, welche vor allem für die SELECT Befehle hilfreich ist, da die hier gemachten Beispiele alle auf das Datenmodell dieser Übungsdatenbank ausgerichtet sind. Die Übungsdatenbanken befinden sich unter:

www.codeconcert.de/SubClassMaterial12SQL.html

Dort finden Sie unter „Datenbanken“ das File Übungsdatenbank.zip. Dieses File kann entpackt werden und unter Windows Systemen mittels eines mitgelieferten Batch Files installiert werden.

Da dieses Dokument auch Fehler haben kann, wird keinerlei Haftung für außerschulische Nutzung übernommen. Sollten Fehler oder Verbesserungsvorschläge existieren, so können diese gerne an maik.aicher@gmx.net weitergereicht werden.

Das Dokument hat nicht die Datenmodellierung im Fokus. ER Diagramme, Normalform und Notationen von Datenmodellen können aus diesem Dokument nicht ersehen werden. Informationen hierzu können über www.CodeConcert.de bezogen werden.

[1] MySQL, siehe http://www.oracle.com/us/products/mysql/index.html bzw. http://www.mysql.com/

2 Inhaltsverzeichnis

Inhalt

3 Datenbanken Allgemein

3.1 SQL erzeugen

Datenbanken werden mittels SQL angesprochen. Die SQL Statements können entweder über den SQL Client mysql.exe erzeugt und abgeschickt werden, mittels SQL Editoren, PHP Administrationsseiten, oder über eigens geschriebene Software, welche über JDBC oder ODBC auf die Datenbank zugreift.
Grundsätzlich empfiehlt es sich, bei der Entwicklung von SQL Befehlen diese nicht direkt im Client zu erzeugen, sondern über SQL Skripte auszuführen, da sie so archiviert und nachvollziehbar getestet werden können. Der Befehl in mysql.exe zur Ausführung von SQL Skripten ist `SOURCE`

```
SOURCE C:/Temp/CreateTable.sql;
```

Achten Sie hierbei darauf, dass die Pfadtrenner entweder mit Forwardslashes, oder mit einem Escapezeichen vor den Backslashes (also „\\“) ausgeführt werden müssen. Innerhalb des referenzierten Files liegen die SQL Befehle sequenziell vor und werden entsprechend abgearbeitet. Um sich im Skript zurechtzufinden, können Kommentare entsprechend des C Syntax angegeben werden `/* dies ist ein Kommentar */`:

```
/* Die Anpassungen werden in der Datenbank testDB vorgenommen */
USE testDB;

/* hier wird eine neue Tabelle erzeugt */
CREATE TABLE Namen(
Vorname VARCHAR(20),
Nachname VARCHAR(30));
```

RDBMS sind im Regelfall darauf ausgelegt, alle Befehle „irgendwie“ auszuführen – also selbst wenn „unpassende“ Daten wie bspw. für die Spalten zu lange Strings oder nicht existierende Datumswerte wie der 31.02.2017 geschrieben werden. MySQL ändert die Werte so ab, dass sie geschrieben werden können und liefert nach der Ausführung die Anzahl der Warnungen, welche Auskunft über die Anpassungen geben. Mit folgendem Befehl können die Warnings angezeigt werden:

```
SHOW WARNINGS;
```

Dies funktioniert im Regelfall aber nur im direkten Anschluss an den Befehl, der die Warnings ausgelöst hat.

3.2 Erzeugung von Datenbanken

Ein RDBMS kann mehrere Datenbanken verwalten. Insofern ist es notwendig, zuerst eine Datenbank zu erstellen, in der anschließend die Tabellen erzeugt werden können. Folgend wird eine Datenbank mit dem Namen `TestDB` erstellt:

```
CREATE DATABASE TestDB;
```

Eigentlich muss bei der Erzeugung einer Datenbank nicht viel mehr beachtet werden. Wenn allerdings standardmäßig ein besonderer Zeichensatz existieren soll, empfiehlt es sich, diesen bei der Datenbank gleich zu hinterlegen (hier im Beispiel UTF8)[2]:

```
CREATE DATABASE TestDB DEFAULT CHARACTER SET utf8
   DEFAULT COLLATE utf8_general_ci;
```

Hiermit setzen wir den Zeichensatz standartmäßig auf UTF8 und mit „COLLATE“ die Regeln, wie mit UTF8 gearbeitet wird (also Vergleiche, Sortierungen etc.) ebenfalls auf UTF8.

Mit dem Befehl `SHOW DATABASES` können die auf der Installation existierenden Datenbanken angezeigt werden.

[2] Je nach RDBMS Version wird UTF8 zwar als Standard gesetzt, wenn sie allerdings sicher gehen wollen, geben sie es explizit an.

3.3 Nutzen von Datenbanken

Bevor man nun mit den eigentlichen Tabellen arbeiten kann, muss die Datenbank ausgewählt werden, auf der die Aktionen durchgeführt werden sollen. Der Befehl `USE` wählt die angegebene Datenbank für alle folgenden Aktivitäten aus:

```
USE TestDB;
```

3.4 Löschen von Datenbanken

Existierende Datenbanken werden mit dem `DROP DATABASE` Befehl wieder gelöscht. Aber Achtung – im Regelfall wird der Befehl ohne Zusatzabfrage wie bspw. „Sind Sie sicher?“ durchgeführt! **Die Daten sind damit weg!**

```
DROP DATABASE TestDB;
```

Wenn Drop Befehle innerhalb von Skripten ausgeführt werden sollen, so ist es natürlich ratsam nur dann die Datenbank zu löschen, wenn sie auch existiert, da sonst eventuelle Skriptabbrüche auftreten könnten. Insofern sollte die Existenz zuvor überprüft werden. Der entsprechende Syntax lautet wie folgt:

```
DROP DATABASE IF EXISTS TestDB;
```

4 Tabellen allgemein

4.1 Erzeugen von Tabellen

Wenn Tabellen erzeugt werden, muss sich der Entwickler vorher genaue Gedanken über die Datenbankstruktur machen. Hierzu sind neben den Datentypen und den Normierungen auch die potentiellen Nutzungsmöglichkeiten der Datenbank in Betracht zu ziehen. Denn: es ist immer mit Aufwand verbunden Datenbanken und Tabellen nachträglich zu ändern!
Eine einfache Tabelle wird mit dem CREATE TABLE Statement erstellt, gefolgt mit dem eindeutigen Tabellennamen. Anschließend werden in Klammern die einzelnen Spalten definiert, indem der (innerhalb der Tabelle) eindeutige Name, gefolgt von dem Datentyp[3] eingetragen wird.

```
CREATE TABLE Stammdaten (
Vorname VARCHAR(20),
Nachname VARCHAR(30),
Strasse VARCHAR(30),
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE);
```

Anmerkung: Bei MySQL kann die für die Tabelle zuständige Speicher Engine explizit angegeben werden. Dies wird bspw. für MyISAM Tabellen mit dem Zusatz `ENGINE = MYISAM` erreicht. Details hierzu entnehmen Sie bitte dem MySQL Manual.

Die Struktur der erzeugten Tabelle kann nun mit dem Befehl `EXPLAIN Stammdaten;` oder alternativ auch mit `DESCRIBE Stammdaten;` bzw. `DESC Stammdaten;` angezeigt werden.

4.1.1 Umbenennen von Tabellen

Tabellennamen werden mit dem Befehl `RENAME TABLE` geändert:

```
RENAME TABLE Stammdaten TO Personendaten;
```

[3] INT als PLZ wird hier nur als Beispiel für Zahlen verwendet – in produktiven Datenbanken werden Postleitzahlen üblicherweise als CHAR modelliert.

4.1.2 Übersicht Datentypen

MySQL unterstützt eine Vielzahl von verschiedenen Datentypen. Dies ist für RDBMS auch sinnvoll, da bei sehr großen Datenmengen der Spaltentyp möglichst genau auf den Bedarf ausgerichtet sein muss um möglichst wenig Speicherplatz unnötig zu belegen. Grundsätzlich sind im SQL Standard eine Anzahl von Datentypen festgelegt, welche in der Regel von den Datenbanksystemen noch erweitert wurden. Dies ist zwar praktisch, erschwert mitunter aber die Migration von einem RDBMS zu einem anderen. Die gezeigten Datentypen sind lediglich eine Auswahl. Eine komplette Übersicht finden sie in der MySQL Dokumentation.

Ganzzahlige Datentypen:

Typ:		Bytes:	Min:	Max:	Bemerkung:
TINYINT		1	-128	127	Synonym wird auch BOOL verwendet.
SMALLINT		2	-32768	32767	
MEDIUMINT		3	-8388608	8388607	Ist nicht Teil des SQL Standards
INT		4	-2147483648	2147483647	Es geht auch INTEGER
BIGINT		8	-9223372036854775808	9223372036854775807	

Bei allen ganzzahligen Datentypen kann das Attribut "UNSIGNED" nachgestellt werden (also INT UNSIGNED). Damit können nur positive Zahlen abgelegt werden, was den Wertebereich auf 0 bis $2^{(8*\text{Bytes})}$ verschiebt. Hier ist allerdings zu beachten, dass Subtraktionen, bei denen mindestens ein UNIGNED Wert beteiligt ist, das Ergebnis nicht negativ sein kann.

***Anmerkung**:*
Eine Definition von INT(4) bewirkt nicht, dass INT nur mit 4 Stellen ablegbar ist, sondern, dass die Darstellung bei einem SELECT mindestens 4 Spalten benötigt (sofern ZEROFILL aktiv ist). Gehen wir von folgender Tabelle aus:

```
CREATE TABLE myTab (id INT(4) ZEROFILL);
```

und folgendem INSERT:

```
INSERT INTO myTab (id) VALUES (2), (12345);
```

Ein SELECT(*) liefert somit folgendes Ergebnis:

```
+-------+
| id    |
+-------+
|  0002 |
| 12345 |
+-------+
```

Der "zu kurze" Wert wird mit 0 aufgefüllt, der "zu lange" wird 1:1 übernommen.

Datentypen für Zahlen mit Nachkommastellen:

Typ:	Bytes:	Min:	Max:	Bemerkung:
FLOAT	4	-3.402823466E+38	3.402823466E+38	Es handelt sich hier um eine „Fließkommazahl". Die Werte können Hardware-bedingt abweichen.
DOUBLE	8	-1.7976931348623157E+308	1.7976931348623157E+308	Es handelt sich hier um eine „Fließkommazahl". Die Werte können Hardware-bedingt abweichen.
DECIMAL[(M[,D])]	-	-	-	Es handelt sich hier um eine „Festkommazahl". M gibt die Anzahl der Ziffern und D die Anzahl der Nachkomma-stellen an.

Wie bei den ganzen Zahlen auch, können hier die Attribute "UNSIGNED" und "ZEROFILL" angegeben werden.

Datentypen für Stringwerte:

Typ:	Bytes[4]:	Bemerkung:
CHAR(M)	1 Byte pro „M“	String mit fester Länge „M“. M darf die Werte 0 bis 255 annehmen.
VARCHAR(M)	1 Byte pro „L“ + 1 (bzw. wenn M > 255 + 2)	String mit variabler Länge „L“ bis maximal „M“. M darf die Werte 0 bis 65.532 annehmen.
TINYTEXT	L + 1 Byte	String mit variabler Länge „L“ bis maximal 2^8 Zeichen.
TEXT	L + 2 Byte	String mit variabler Länge „L“ bis maximal 2^{16} Zeichen.
MEDIUMTEXT	L + 3 Byte	String mit variabler Länge „L“ bis maximal 2^{24} Zeichen.
LONGTEXT	L + 4 Byte	String mit variabler Länge „L“ bis maximal 2^{32} Zeichen.

***Anmerkung**:*
Der Hauptunterschied zwischen CHAR und VARCHAR ist, dass CHAR immer die gesamte Länge abspeichert. Ein Feld CHAR(3) wird im Speicher immer alle drei Zeichen belegen, auch wenn nur ein Zeichen gespeichert wird. Die „freien“ Felder werden rechts mit Leerzeichen aufgefüllt. Gehen wir von folgender Tabelle aus:

```
CREATE TABLE myTab (v1 VARCHAR(3), v2 CHAR(3));
```

und folgendem INSERT:

```
INSERT INTO myTab (v1, v2) VALUES ("x", "x"), (" x ", " x ");
```

Zur Analyse setzen wir folgendes SELECT ab, welches vor und nach dem Ausgabestring ein "-" setzt:

```
SELECT CONCAT("-", v1, "-"), CONCAT("-", v2, "-") FROM myTab;
```

Wir erhalten folgendes Ergebnis:

```
+----------------------+----------------------+
| CONCAT("-", v1, "-") | CONCAT("-", v2, "-") |
+----------------------+----------------------+
| -x-                  | -x-                  |
| - x -                | - x-                 |
+----------------------+----------------------+
```

In der ersten Zeile sehen wir nun, dass beide Felder das "x" und nur das "x" ausgeben. Das scheint auf den ersten Blick der oben gemachten Aussage zu wiedersprechen. Wenn wir nun jedoch die zweite Zeile ansehen erkennen wir, dass bei der CHAR Spalte das rechte Leerzeichen verschwunden ist! Das heißt, dass die Ausgabe bei CHAR Feldern alle rechten Leerzeichen entfernt, da MySQL davon ausgeht, dass sie aufgrund des automatischen Auffüllens dort gelandet sind.

Die Datentypen TINYBLOB, BLOB, MEDIUMBLOB und LARGEBLOB sind identisch mit den entsprechenden TEXT Datentypen mit dem Unterschied, dass die Bytes nicht als Zeichensatz interpretiert werden und sie somit beim Sortieren numerisch behandelt werden.

Datums- und Zeitdatentypen

Typ:	Bytes:	Bemerkung:
DATE	3 Bytes	Datum im Format „YYYY-MM-DD“.
TIME	3 Bytes	Zeit im Format „HH:MM:SS“
DATETIME	8 Bytes	Datum und Zeit im Format „YYYY-MM-DD HH:MM:SS“
TIMESTAMP	4 Bytes	Datum und Zeit im Format „YYYY-MM-DD HH:MM:SS“, wobei MySQL bei der Standardkonfiguration den Wert bei jeder Änderung des Datensatzes den Wert auf die Änderungszeit (+Datum) anpasst.

4.1.3 Nullable Felder / not nullable Felder

Bei Tabellen gibt es mitunter Felder, welche zwingend Einträge benötigen, da sonst der Inhalt nicht sinnvoll nutzbar ist. Bspw. ist eine Tabelle welche Personendaten speichert sinnlos, wenn weder Vor- noch Nachname oder Geburtsdatum enthalten sind. Beim Anlegen einer Tabelle ist es somit sinnvoll, diejenigen Spalten dergestalt zu definieren, dass nur Datensätze akzeptiert werden, welche in den Pflichtfeldern nicht auf NULL stehen (nicht festgelegte Werte werden auf den Standarddefaultwert bzw. wenn vorhanden den definierten Defaultwert gesetzt). Dies geschieht mit dem Definitionszusatz `NOT NULL`. Die Tabellendefinition sollte dementsprechend wie folgt aussehen:

[4] Wir gehen hier von einem Zeichensatz aus, der 1 Byte pro Zeichen benötigt.

```
CREATE TABLE Stammdaten (
Vorname VARCHAR(20) NOT NULL,
Nachname VARCHAR(30) NOT NULL,
Strasse VARCHAR(30),
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE NOT NULL);
```

Ob in einer existierenden Tabelle ein Feld Nullable ist, kann über den `DESCRIBE` Befehl ersehen werden.

4.1.4 Primary Key

Um zu unterbinden, dass Datensätze, bzw. relevante Spalten von Datensätzen doppelt existieren, können sogenannte Primärschlüssel definiert werden. Umgekehrt kann man dies auch so interpretieren, dass man diejenigen Attribute identifiziert, welche jeden Datensatz der Tabelle eindeutig definieren – man könnte also von einer Art „Adresse“ sprechen (also wenn ich einen Datensatz suche, indem ich pro Primärschlüsselattribut je einen Wert vorgebe, so erhalte ich entweder keinen, oder exakt einen Datensatz von der Tabelle). Aus diesem Grunde akzeptieren Primärschlüsselfelder auch nicht den Zustand NULL.

Es gibt zwei Möglichkeiten einen Primärschlüssel zu definieren. Wenn lediglich eine Spalte den Primärschlüssel darstellt, so kann dies mit dem Definitionszusatz `PRIMARY KEY` erfolgen:

```
CREATE TABLE Stammdaten (
Vorname VARCHAR(20) NOT NULL,
Nachname VARCHAR(30) PRIMARY KEY,
Strasse VARCHAR(30),
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE NOT NULL);
```

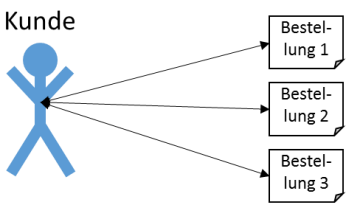
Wenn mehrere Spalten als Primärschlüssel fungieren sollen, so muss der PRIMARY KEY Zusatz am Ende der Spaltenaufzählung erfolgen:

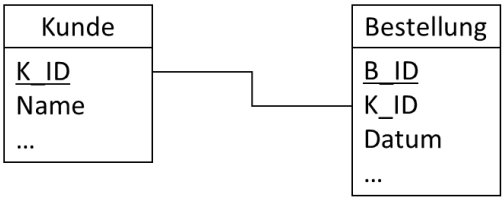
```
CREATE TABLE Stammdaten (
Vorname VARCHAR(20),
Nachname VARCHAR(30),
Strasse VARCHAR(30),
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE,
PRIMARY KEY(Vorname, Nachname, Geburtsdatum));
```

Spalten, welche als Primary Key definiert wurden sind automatisch als not nullable geflagged – der `NOT NULL` Zusatz kann somit entfallen.
Ob in einer existierenden Tabelle ein Feld Primärschlüssel ist kann über das `DESCRIBE` Statement ersehen werden.

4.1.5 Foreign Key

Während Primary Keys verhindern, dass ein Datensatz doppelt vorkommt, verhindert der Foreign Key (Fremdschlüssel), dass die referenzielle Integrität verletzt wird. Hierunter versteht man, dass jeder Wert, der für die Abbildung einer Beziehung verwendet wird, auch wirklich vorhanden ist. Sehen wir uns hierbei das nebenstehende Bild an – ein Kunde hat drei Bestellungen gemacht, was in der Datenbank jeweils mit zwei Tabellen und vier Datensätzen realisiert werden würde.





Wenn wir nun von „referenzieller Integrität“ sprechen meinen wir damit, dass jeder Datensatz in der Tabelle „Bestellung“ eine Kunden ID (K_ID) hat, welche auch in der Kundentabelle vorkommt.

Wenn dem nicht so ist, so ist dies eine Verletzung der referenziellen Integrität, welche somit die Brauchbarkeit der ganzen Datenbank in Frage stellt (es existiert Bestellung, aber der zugehörige Kunde nicht).

Solche Inkonsistenzen können durch drei Aktionen entstehen:

- In die Tabelle „Bestellung“ ein Datensatz mit einer K_ID geschrieben, welche in der Kundentabelle nicht existiert
- Es wird ein Datensatz aus der Kundentabelle gelöscht, deren K_ID in der Tabelle „Bestellung“ vorhanden ist
- Es wird in der Kundentabelle oder der Tabelle „Bestellung“ die K_ID verändert.

Bevor wir nun loslegen, noch ein Wort zu den Speicherengines. Von den beiden am meisten verbreiteten Speicherengines InnoDB und MyISAM beherrscht nur die InnoDB die Prüfung der referenziellen Integrität. Insofern müssen wir dies beim Tabellenerzeugen (der Eltern und Kindtabelle) angeben (wobei oftmals die InnoDB der Standard ist).

***Achtung**: bei dem von uns im Unterricht genutzten USBWebserver ist bei den älteren Versionen in der my.ini Konfigurationsdatei der Eintrag „skip-innodb“, welcher verhindert, dass die InnoDB Speicherengine genutzt wird. Wenn wir also die Prüfung der referenziellen Integrität durchführen wollen, kommentieren wir das skip-innodb aus:*
`#skip-innodb`
Anschließend muss Mysql neu gestartet werden (Achtung – nicht der USBWebserver, denn das würde die Konfiguration wieder zurücksetzen).

Nachdem wir sichergestellt haben, dass unser Server die InnoDB akzeptiert, werden die Tabellen erstellt:

```
CREATE TABLE Kunde (
K_ID INT PRIMARY KEY,
Name VARCHAR(30)
) ENGINE INNODB;

CREATE TABLE Bestellung (
B_ID INT PRIMARY KEY,
K_ID INT,
Datum DATE,
FOREIGN KEY (K_ID) REFERENCES Kunde(K_ID)
) ENGINE INNODB;
```

Wenn wir nun versuchen, die referenzielle Integrität zu verletzen (sei es über INSERT, UPDATE oder DELETE), so wird die Datenbank dies nicht erlauben und einen Fehler melden.

Das Verhalten der Datenbank kann bezüglich UPDATE und DELETE jedoch noch etwas angepasst werden. Beispielhaft sei hier der CASCADE Zusatz genannt. Dieser sorgt dafür, dass bei UPDATE und/oder DELETE die Änderung von der Elterntabelle auf die Kindtabelle vererbt wird:

```
CREATE TABLE Bestellung (
B_ID INT PRIMARY KEY,
K_ID INT,
Datum DATE,
FOREIGN KEY (K_ID) REFERENCES Kunde(K_ID)
ON UPDATE CASCADE
ON DELETE CASCADE
) ENGINE INNODB;
```

In diesem Fall würde ein Löschen des Datensatzes in der Kundentabelle zur Löschung aller korrespondierenden Datensätze in der Tabelle „Bestellung“ führen. Weiterhin würde eine Änderung des Feldes K_ID in einem Datensatz der Kundentabelle alle korrespondierenden Datensätze der Tabelle „Bestellung“ auch ändern.

4.1.6 Defaultwerte

Für gewisse Felder macht es Sinn, Defaultwerte vorzusehen – also Werte, welche automatisch dann gesetzt werden, wenn beim Insertstatement kein dedizierter Inhalt vorgegeben wird. Defaultwerte können in einer existierenden Tabelle mittels dem `DESCRIBE` Befehl identifiziert werden. Vorgegeben werden die Werte mittels dem `DEFAULT` Zusatz:

```
CREATE TABLE Stammdaten (
Vorname VARCHAR(20),
Nachname VARCHAR(30),
Strasse VARCHAR(30) DEFAULT 'Unbekannt',
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE,
PRIMARY KEY(Vorname, Nachname, Geburtsdatum));
```

Defaultwerte werden nur dann gesetzt, wenn kein Wert beim Insertstatement vorgegeben ist. Wenn explizit NULL gesetzt wird, so wird auch NULL eingetragen (wenn die Spalte NULL erlaubt).
Zu beachten ist, dass mit wenigen Ausnahmen die gesetzten Werte Konstanten sein müssen (also keine vom RDBMS dynamisch generierten Inhalte). Weiterhin gibt es in MySQL Datentypen, welche überhaupt nicht mit einem Defaultwert versehen werden können (TEXT und BLOB).

4.1.7 Auto Increment

Eine weitere Option zur Vorbelegung von Werten ist das sogenannte Auto Increment bei Integerfeldern. Hier wird jeder Datensatz mit einem neuen Wert versehen, der um eins höher ist, als der höchste Wert dieses Attributs innerhalb der Tabelle. Dies ist in MySQL pro Tabelle immer nur für eine Spalte möglich, welche entweder ein Index, oder ein Primärschlüssel sein muss. Der Definitionszusatz hierfür ist `AUTO_INCREMENT`.

```
CREATE TABLE Stammdaten (
StammID INT PRIMARY KEY AUTO_INCREMENT,
Vorname VARCHAR(20) NOT NULL,
Nachname VARCHAR(30) NOT NULL,
Strasse VARCHAR(30) DEFAULT 'Unbekannt',
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE NOT NULL);
```

Bei einem Insert Statement wird nun (beginnend mit 1) jeweils der nächst höhere Wert eingesetzt. Wenn also die Zahlen aufgrund früherer Manipulationen nicht lückenlos aufsteigend sind (bspw. 1, 2, 3, 7, 8), so wird beim nächsten Insert der Wert 9 vergeben – die Lücken werden nicht aufgefüllt.
Ein Auto Increment Feld kann auch manuell mit einem festen Wert gesetzt werden, sofern er noch nicht in der Spalte vergeben ist.

Diese Auto Increment Felder werden gerne für die Erstellung von künstlichen IDs verwendet. Hier ist jedoch Vorsicht geboten, da diese IDs oftmals in mehreren Tabellen wiederverwendet werden. Wenn nun Datenbanken neu aufgesetzt werden, so müssen vorher diese erzeugten IDs gesichert werden!
Anmerkung: `AUTO_INCREMENT` Felder bei Tabellen mit multiplen Primärschlüssel werden von MySQL nur von MyISAM und BDB Tabellen akzeptiert.

4.2 Ändern von Tabellen

4.2.1 Allgemeines

Mitunter kommt es vor, dass Tabellen strukturell verändert werden müssen. Hierzu wird das `ALTER TABLE` Statement verwendet. Dieser Befehl ist sehr mächtig und kann extrem vielseitig eingesetzt werden. Aus diesem Grund werden hier nur die wichtigsten Einsatzmöglichkeiten besprochen. Bei der Änderung von Tabellen ist immer darauf zu achten, dass sich üblicherweise bereits Daten in den Tabellen befinden. Diese könnten bei der Durchführung der Änderung für Probleme sorgen (wenn z.B. die Daten in der Spalte zwar für die alte Definition OK sind, in der neuen jedoch nicht ladbar wären – also bspw. die Umwandlung von `VARCHAR(30)` nach `VARCHAR(20)` für den Fall das es mindestens einen Datensatz mit mehr als zwanzig Zeichen gibt). MySQL versucht zwar im Regelfall eine Konvertierung durchzuführen, dies geht aber mitunter mit einem Datenverlust einher, was lediglich eine Warnung nach der Anpassung der Daten mit sich führt!
Grundsätzlich fängt jeder Befehl mit `ALTER TABLE` gefolgt von dem Tabellennamen an.

4.2.2 Hinzufügen von Spalten

Der wahrscheinlich häufigste Fall ist das Hinzufügen von Spalten.

```
ALTER TABLE Stammdaten
ADD COLUMN Geburtsname VARCHAR(30);
```

Hier ist darauf zu achten, dass bei einer Tabelle, welche bereits Daten beinhaltet die Werte in der neuen Spalte NULL sind. Um dies zu verhindern kann auch bei der nachträglichen Ergänzung um eine Spalte jede Zusatzdefinition durchgeführt werden, wie beim initialen Anlegen einer Tabelle. Für den geschilderten Fall wäre dies:

```
ALTER TABLE Stammdaten
ADD COLUMN Geburtsname VARCHAR(30) DEFAULT 'NN';
```

Bei Durchführung dieses Befehls würden nun alle existierenden Datensätze in der neuen Spalte den Wert “NN” erhalten. Eine Alternative zu `DEFAULT` ist mittels SQL Skripten dediziert Inhalte einzufügen.

4.2.3 Entfernen von Spalten

Spalten können auch wieder entfernt werden. Auch hier ist der `DROP` Befehl zu verwenden, wobei hier der Zusatz `COLUMN` zu erfolgen hat:

```
ALTER TABLE Stammdaten
DROP COLUMN Geburtsname;
```

Wie bei allen anderen Fällen von `DROP` ist auch hier zu beachten, dass die Datenbank das Löschen ohne weitere Nachfragen durchführt. Die Dateninhalte der Tabellenspalte sind danach nicht mehr vorhanden!

4.2.4 Umbenennen von Spalten

Sollten Spalten Umbenannt werden, so steht hierfür der `CHANGE` Befehl bereit. Da dieser jedoch nicht einfach die Spalte umbenennt, sondern eine neue erzeugt, die Inhalte kopiert und die alte Spalte gelöscht wird, müssen Strukturinformationen jedes Mal aufs Neue mitgegeben werden. Gerade bei Primärschlüssel, Indexspalten und Auto Increment Spalten ist hier besondere Vorsicht geboten – es ist wenn irgend möglich zu vermeiden. Versuchen Sie also beim Design einer Tabelle von vorne herein richtige Namen zu vergeben, indem Sie sich vorher eine Namenskonvention verbindlich überlegen und diese auch schriftlich festhalten! Folgend der Befehl für die Änderung des Spaltennamens von Nachname in Familienname:

```
ALTER TABLE Stammdaten
CHANGE Nachname Familienname VARCHAR(30) NOT NULL;
```

4.2.5 Ändern von Spalteneigenschaften

Prinzipiell kann jede Eigenschaft einer Spalte nachträglich verändert werden, sofern die Dateninhalte dies zulassen. MySQL versucht, soweit irgend möglich die Änderung durchzuführen und die Dateninhalte zu konvertieren. Das Ändern von Datentypen ist prinzipiell mit zwei verschiedenen Befehlen möglich. Zum einen ist dies der Befehl `CHANGE`, welche oben bereits beschrieben wurde. Hier wird der ursprüngliche Name gleich dem zu ändernden Namen zu setzen und der gewünschte Datentyp zu setzen:

```
ALTER TABLE Stammdaten
CHANGE Nachname Nachname VARCHAR(40) NOT NULL;
```

Die zweite Alternative ist der Befehl `MODIFY`. Hier muss nur der Spaltenname mit dem gewünschten neuen Datentyp gesetzt werden:

```
ALTER TABLE Stammdaten
MODIFY Nachname VARCHAR(40) NOT NULL;
```

Natürlich können auch mit dem `MODIFY` Befehl fast alle anderen Eigenschaften verändert werden. Primärschlüsseländerungen sind hiervon im Regelfall ausgenommen. Dies erfolgt über den Befehl `ADD`:

```
ALTER TABLE Stammdaten
ADD PRIMARY KEY (Vorname, Nachname);
```

Gelöscht werden kann ein Primärschüssel nur komplett über den `DROP` Befehl:

```
ALTER TABLE Stammdaten
DROP PRIMARY KEY;
```

4.2.6 Anlegen von Indizes

Ein Index ist eine Hilfestellung für Datenbanken, die entsprechenden Datensätze schneller zu finden. Wie in einem Buch wird eine Art Inhaltsverzeichnis angelegt, um nicht sämtliche Datensätze (also Seiten) nach dem gesuchten Wert durchzuprüfen. Indizes werden entweder automatisch bei der Erzeugung von Primärschlüsseln angelegt, oder mittels dem Befehl `CREATE INDEX`:

```
CREATE INDEX id_stammdaten ON Stammdaten (StammID);
```

Dieser Befehl kann auch direkt in das CREATE TABLE Statement eingebunden werden:

```
CREATE TABLE Stammdaten (
StammID INT,
Vorname VARCHAR(20) NOT NULL,
Nachname VARCHAR(30) NOT NULL,
Strasse VARCHAR(30),
PLZ INT,
Ort VARCHAR(30),
Geburtsdatum DATE NOT NULL,
INDEX id_stammdaten (StammID));
```

Indizes sollten immer dann verwendet werden, wenn die entsprechenden Spalten für Joins verwendet werden, die Tabellen relativ groß sind und die Abfragen relativ häufig und zeitkritisch durchgeführt werden.
Anmerkung: Zum Testen der Indexfunktionalitäten können Sie die bereitgestellten Tabellen der BigDB laden (aufgrund der Dateigröße nur im Schulnetz vorhanden). Versuchen Sie vorher anhand der Daten die richtigen Tabellenstrukturen zu definieren und umzusetzen.

Wichtig zu wissen ist noch, dass beim Anlegen eines Primary Key Feldes dort der Index automatisch erstellt wird – es ist also nicht notwendig hier noch einen expliziten Index anzulegen. Gleiches gilt bei InnoDB, wenn ein Foreign Key Feld angegeben wird. Sollte ein manuell erstellter Index wieder entfernt werden, so hilft das DROP INDEX Statement:

```
DROP INDEX id_stammdaten ON Stammdaten;
```

4.3 Einfügen von Daten

Früher oder später müssen in die erzeugten Tabellen auch Daten hineinfließen. Dies geht entweder über die RDBMS spezifischen Tools wie mysqlimport.exe, oder über SQL Befehle. Hier gibt es das Datensatzweise Schreiben via `INSERT`, oder die Möglichkeit ganze Datenfiles via `LOAD DATA INFILE` zu laden. In diesem SQL Manual wird auf die beiden letztgenannten Möglichkeiten eingegangen. Wichtig zu erwähnen ist, dass die Angaben in diesem Kapitel das Thema Transaktionssicherheit nicht abdecken.

4.3.1 Laden von Datenfiles

Der Befehl, um Datenfiles in Textform zu laden ist `LOAD DATA INFILE`. Bei der Standardisierten Nutzung von `LOAD DATA INFILE` sind folgende Eigenschaften zu beachten:

- Es wird nach Zeilenbegrenzungen bei Zeilenumbrüchen gesucht. Diese müssen eventuell explizit mit `LINES TERMINATED BY '\r\n'` angegeben werden (im Regelfall bei Windows Rechnern).
- Die Daten werden via Tabulatoren getrennt
- Felder müssen nicht in Anführungszeichen eingeschossen werden
- Tabulatoren, Zeilenumbrüche oder Escapezeichen \, denen ein \ vorausgeht, werden als literale Zeichen und nicht als Kontrollzeichen interpretiert.
- Die Reihenfolge der Felder im Textfile wird auf die Reihenfolge der Spalten der Tabelle zugeordnet.

Dem `LOAD DATA INFILE` Befehl folgt noch der Filename als Stringliteral:

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
LINES TERMINATED BY '\r\n';
```

Das Verhalten der Ladeprozedur kann noch weiter Konfiguriert werden. Hier seien nur die wichtigsten Punkte angesprochen, tiefergreifende Konfigurationen können dem MySQL Manual entnommen werden:

Ignorieren der ersten Zeilen:
Sollten im Textfile Überschriftenzeilen existieren, welche keine Dateninhalte haben, so müssen diese beim Einlesen übersprungen werden. Der Befehl für bspw. 1 Zeile lautet `IGNORE 1 LINES`.

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
LINES TERMINATED BY '\r\n'
IGNORE 1 LINES;
```

Abschneiden von Zeilenpräfixen:
Manche Textfiles besitzen pro Zeile einen Präfix (wie z.B. Nummerierungen). Diese können mit `LINES STARTING BY` ignoriert werden (wenn die Zeilen mit *Nummer->* beginnen):

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
LINES STARTING BY '->'
      TERMINATED BY '\r\n';
```

Ändern des Trennungszeichens:
Sollte die vorliegende Datei nicht mit Tabulatoren, sondern von einem anderen Zeichen (bspw. Semikolon) getrennt sein, so kann dies mit `FIELDS TERMINATED BY ';'` konfiguriert werden:

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\r\n';
```

Ignorieren von einschließenden Zeichen:

Bei manchen Datenfiles ist jedes Feld mit einem Zeichensatz eingeschlossen (bspw. mit Anführungszeichen – die Daten sehen dann wie folgt aus: "1. Wert" "2. Wert" "3. Wert"). Da die Anführungszeichen nicht Teil der Daten sind, müssen diese Zeichen mit `FIELDS ENCLOSED BY '"'` ignoriert werden.

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
FIELDS ENCLOSED BY '"'
LINES TERMINATED BY '\r\n';
```

Sind die Anführungszeichen nur bei Strings und ähnlichen Spalten gesetzt, so ist dies mit dem Zusatz `OPTIONALLY` anzugeben:

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
FIELDS OPTIONALLY ENCLOSED BY '"'
LINES TERMINATED BY '\r\n';
```

Sollten mehrere Anpassungen auf Zeilen – oder Feldebene durchgeführt werden müssen, so können die Anweisungen auch hintereinander geschrieben werden;

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
FIELDS TERMINATED BY ';' ENCLOSED BY '"'
LINES TERMINATED BY '\r\n';
```

Änderung der Spaltenreihenfolge:
Liegen die Daten nicht in der Reihenfolge wie die Tabellenspalten vor, bzw. gibt es Spalten, welche überhaupt nicht Teil des Datenfiles sind, so muss dies explizit angegeben werden:

```
LOAD DATA INFILE 'C:/Temp/Stammdaten.txt'
INTO TABLE Stammdaten
LINES TERMINATED BY '\r\n'
(Nachname, Vorname, Geburtsdatum);
```

Anmerkung: Sollten Sie weitere Konfigurationseinstellungen gemäß dem MySQL Manual vornehmen achten Sie unbedingt auf die vorgeschriebene Reihenfolge der Einstellungen. MySQL gibt hier im Regelfall nur sehr schlechte Fehlermeldungen aus, was die Fehlersuche erschwert.

4.3.2 Insert Statements

Einzelne Datensätze werden üblicherweise mit dem `INSERT` Statement geschrieben. Hierfür müssen die Spalten und die Werte getrennt voneinander angegeben werden:

```
INSERT INTO Stammdaten (
Vorname, Nachname, Strasse, PLZ, Ort, Geburtsdatum)
VALUES ('Michael', 'Mayer', 'Huberweg 3', 86799,
'Musterdorf', '1976-12-04');
```

Bei diesem Befehl werden die Spalten und Werte entsprechend der angegebenen Reihenfolge zugeordnet. Fehlt eine Spalte, so wird diese mit NULL, oder dem gesetzten Defaultwert belegt.

Als Alternative kann im Insertstatement auch das SET Keyword verwendet werden:

```
INSERT INTO Stammdaten
SET Vorname = 'Michael', Nachname = 'Mayer', Strasse = 'Huberweg 3', PLZ = 86799,
Ort = 'Musterdorf', Geburtsdatum = '1976-12-04';
```

Beim INSERT INTO Statement können auch Rechenoperationen bzw. Expressions eingebettet werden. Diese werden später noch weiter vertieft. Beispielhaft soll folgender Befehl die Möglichkeiten andeuten:

```
INSERT INTO Prozentangaben (
ProzentKey, Absolutwert, Prozentwert)
VALUES ('1/4', 0.25, Absolutwert*100);
```

Auch ist es möglich, mit einem INSERT INTO Statement mehrere Datensätze zu schreiben:

```
INSERT INTO Stammdaten (
Vorname, Nachname, Strasse, PLZ, Ort, Geburtsdatum)
VALUES
('Michael', 'Mayer', 'Huberweg 3', 86799, 'Musterdorf', '1976-12-04'),
('Peter', 'Huber', 'Mayerweg 5', 36799, 'Musterstadt', '1973-02-02');
```

Basierend hierauf ist es auch möglich, die Werte aus einem Select zu erzeugen:

```
INSERT INTO ZielTabelle (v1, v2)
SELECT v1, v2 FROM Quelltabelle;
```

Hierbei ist jedes Select erlaubt, bei dem die Spaltenanzahl der Spaltenanzahl des Insertstatements entspricht.

4.4 Ändern von Daten

Datenänderungen können über zwei Arten realisiert werden. Entweder der komplette Datensatz wird mittels REPLACE INTO ersetzt, oder Teile von Datensätzen werden mittels UPDATE geändert. Das REPLACE INTO Statement wird exakt wie ein INSERT INTO Statement genutzt und aufgebaut. Der Unterschied ist, dass für den Fall, dass die Werte des Primärschlüssels bereits in der Tabelle gefunden werden, der entsprechende Datensatz ersetzt wird. Wird kein entsprechender Primärschlüssel gefunden, so wird ein neuer Datensatz erzeugt.

```
REPLACE INTO Stammdaten (
Vorname, Nachname, Strasse, PLZ, Ort, Geburtsdatum, StammID)
VALUES ('Michael', 'Mayer', 'Huberweg 3', 26793,
'Musterdorf', '1976-12-04', 4);
```

Der zweite Weg über UPDATE benötigt neben den Spalten, die geändert werden müssen, noch eine explizite Auswahl der Datensätze mittels WHERE Klausel. Diese kann, im Gegensatz zum REPLACE INTO Statement auch mehrere Datensätze gleichzeitig auswählen – wenn bspw. für eine ganze Gruppe von Datensätzen ein bestimmtes Feld auf einen neuen Wert gesetzt werden muss. Details zur WHERE Klausel werden weiter unten bei der Behandlung von SELECT Befehlen erörtert.

```
UPDATE Stammdaten SET Vorname = 'Heinz'
WHERE Nachname = 'Mayer';
```

Der obenstehende Befehl ändert somit bei sämtlichen Einträgen mit Nachname = Mayer den Vorname auf Heinz.

4.5 Löschen von Tabellen

Auch Tabellen können durch den DROP Befehl (mit dem Zusatz TABLE) gelöscht werden. Genauso wie bei allen anderen Nutzungsarten des DROP Kommandos ist hier darauf zu achten, dass keine weiteren Fragen gestellt werden, sondern MySQL den Befehl sofort ausführt.

```
DROP TABLE Stammdaten2;
```

4.6 Auslesen von Daten

Wenn einmal Daten in der Datenbank enthalten sind, ist der nächste logische Schritt, dass Daten ausgelesen werden. Hierzu dient der `SELECT` Befehl. Die allermeisten Operationen auf Tabellen sind `SELECT` Befehle. Die verschiedenen RDBMS Hersteller halten sich zwar an den Standard SQL Syntax, jedoch bieten sie darüber hinaus noch sehr viele weiteren Funktionen an, welche das Leben eines Datenbanknutzers durchaus erleichtern können. Hier ist jedoch Vorsicht geboten, da im Zweifelsfall die Kompatibilität der Skripte zu anderen Datenbankherstellern nicht mehr gegeben ist.
Die hier angegebenen `SELECT` Statements laufen alle auf der Ihnen zur Verfügung gestellten Testdatenbank.

4.6.1 Einfaches Select

Ein `SELECT` Befehl dient der Auswahl von Daten aus einer Datenmenge. Diese Datenmenge ist im Regelfall eine Tabelle – kann jedoch auch eine View, oder gar das Ergebnis einer vorausgegangenen `SELECT` Abfrage sein. Das einfachste `SELECT` ist die Abfrage aller Daten einer bestimmten Tabelle:

```
SELECT * FROM Produkt;
```

	Sp. 1	Sp. 2	Sp. 3	Sp. 4
Z. 1				
Z. 2				
Z. 3				
Z. 4				
Z. 5				
Z. 6				
Z. 7				
Z. 8				
Z. 9				

Als Ergebnis sieht man die komplette Tabelle mitsamt Inhalten in einer Textausgabe.

Sollte die Tabelle viele Spalten besitzen kann es sein, dass die Ausgabe aufgrund von Zeilenumbrüchen unübersichtlich wird. Hier kann entweder die Zeilenbreite der Konsole erweitert werden, oder besser noch werden nur diejenigen Spalten ausgewählt, die den Abfragenden auch interessieren:

```
SELECT ProduktID, Bezeichnung FROM Produkt;
```

	Sp. 1	Sp. 2	Sp. 3	Sp. 4
Z. 1				
Z. 2				
Z. 3				
Z. 4				
Z. 5				
Z. 6				
Z. 7				
Z. 8				
Z. 9				

Wie man erkennen kann, werden nun nur noch die Ergebnisse der Spalten ProduktID und Bezeichnung angezeigt.

Wichtig: In produktiven Datenbanken gibt es oftmals Tabellen mit mehreren hunderttausend Einträgen. Ein `SELECT * ` auf diese Tabellen ohne Einschränkung würde eine sehr lange Ausführungsdauer aufweisen, da hier die Daten nicht nur aus der Datenbank gelesen werden müssen, sondern auch über das Netzwerk an den Client gesendet und hier wiederum angezeigt werden müssen. Dies kann durchaus über mehrere Minuten dauern. Insofern ist es sinnvoll zuerst die Anzahl der abzufragenden Daten zu prüfen:

```
SELECT COUNT(*) FROM Produkt;
```

Hier wird ausschließlich die Anzahl der Datensätze gezeigt, welche durch das `SELECT *` angezeigt werden würden. Wenn die Zahl zu groß ist, dürfte das `SELECT` Statement auch keinen Sinn machen, da eine derart große Anzahl von Daten manuell nicht ausgewertet werden kann.

Üblicherweise weiß man jedoch vorher, wie viele Daten in einer Tabelle sind. Schwieriger wird es, wenn man das Select von der Tiefe her einschränkt, also die zu selektierenden Datensätze klassifiziert. Dies erfolgt mit der Ergänzung um die `WHERE` Klausel.

```
SELECT * FROM Produkt WHERE ProduktID = 772204;
```

	Sp. 1	Sp. 2	Sp. 3	Sp. 4
Z. 1				
Z. 2				
Z. 3				
Z. 4				
Z. 5				
Z. 6				
Z. 7				
Z. 8				
Z. 9				

Bei komplizierteren `WHERE` Klauseln macht es somit durchaus Sinn, sich vorher die Anzahl der ausgewählten Datensätze anzeigen zu lassen, um nicht sinnlos riesige Datenkolonnen auf dem Client durchlaufen zu lassen. Der Clientprozess kann zwar üblicherweise mittels der Tastenkombination CTRL+C gestoppt werden, dies hält im Regelfall die Datenbank jedoch nicht davon ab, den Prozess Serverseitig weiterlaufen zu lassen – sogenannte Ghostprozesse sind die Folge (und bei Datenbanksystemen mit eigenen DB-Administratoren entsprechend auch genervte Admins).

Eine zweite Option, um die Funktionalität einer Abfrage vorab zu testen ist, die Anzahl der ausgegebenen Datensätze zu limitieren. Hier sollte jedoch mit Vorsicht herangegangen werden, damit nicht aus Versehen die Limitierung in ein produktives Datenbank Skript gelangt und somit inhaltliche Fehler verursacht – derartige Fehler sind schwer zu finden. Mit dem Zusatz `LIMIT` können Bereiche aus dem Ergebnisset herausgelöst und angezeigt werden:

```
SELECT * FROM Kontenumsatz LIMIT 10,5;
```

Das Oben genannte Statement liefert die Zeilen 11-16 aus der Tabelle Kontenumsatz, da die erste Zahl (hier 10) den Versatz zur ersten Zeile angibt und die zweite Zahl (hier 5) die maximale Anzahl der Datensätze vorgibt. Sollten nur die ersten Datensätze benötigt werden, so reicht eine Zahl:

```
SELECT * FROM Kontenumsatz LIMIT 5;
```

Fortgeschrittene Anwender können sich noch mit der `SELECT` Optimierung beschäftigen. Soviel sei an dieser Stelle erwähnt – jeder `SELECT` Vorgang kann vorab von MySQL analysiert werden. Mit Hilfe des `EXPLAIN` Statements können (gerade bei Nutzung mehrerer Tabellen in einem Statement –Join siehe Unten) wichtige Informationen über die temporären Datensatzmengen ersehen werden. Das sehr einfache Beispiel des vorausgegangenen `SELECT` Statements wird mit dem folgenden Befehl analysiert:

```
EXPLAIN SELECT * FROM Kontenumsatz LIMIT 5;
```

Abschließend ist zu merken, dass bei einem `SELECT` mit der Spaltenangabe und der `WHERE` Klausel sowohl die Breite, als auch die Tiefe einer Abfrage gesteuert werden kann. Schematisch kann ein `SELECT` Statement also wie folgt angesehen werden:

SELECT	Welcher Typ?	**FROM**	Welche Datenmenge?	**WHERE**	Welche Datensätze?

Satement Teil:	**Bedeutung:**
SELECT	Schlüsselwort zur Einleitung des Select Satements
Welcher Typ?	Welche Datentypen, bzw. Welche Spalten der genutzten Datenmengen sollen ausgegeben werden.
FROM	Schlüsselwort für Select Statement zur Angabe der Datenmengen, aus denen die Informationen extrahiert werden sollen.
Welche Datenmenge?	Grunddatenmenge, aus denen die Informationen extrahiert werden sollen. Dies können Tabellen sein, Views, aber auch Subselects.
WHERE	Schlüsselwort für Select Statement zur Angabe, welche Datensätze extrahiert werden müssen – also Angabe eines Datenfilters.
Welche Datensätze?	Spezifikation des Datenfilters. Hier wird angegeben, welche Eigenschaften die Datensätze haben müssen, damit sie in die Ergebnismenge übernommen werden.

Zusätzlich zu dem einfachen `SELECT` Statement können noch weitere Funktionen berücksichtigt werden. Als erstes ist die Sortierfunktion zu nennen: `ORDER BY`:

```
SELECT * FROM Produkt ORDER BY ProduktID;
```

Hierbei wird die Ausgabe aufsteigend nach der ProduktID sortiert (dies kann mit dem Schlüsselwort `ASC` auch explizit angegeben werden). Absteigend kann sortiert werden, indem `DESC` mit angegeben wird:

```
SELECT * FROM Produkt ORDER BY ProduktID DESC;
```

Für den Fall, dass es mehrere Spalten gibt nach denen sortiert werden muss, können diese auch nacheinander angegeben werden, wobei die Sortierreihenfolge `DESC` pro Spalte angegeben wird:

```
SELECT * FROM Kontenumsatz ORDER BY Name, Datum DESC;
```

Die nächste Erweiterung ist das Vergeben von Alias Namen. Dies wird vor allem bei der Nutzung der weiter Unten beschriebenen Funktionalitäten benötigt. Ein Alias Name wird verwendet, um die Spalte, in der die Daten angezeigt werden mit einer anderen Überschrift zu versehen, als es die Tabelle vorgibt:

```
SELECT ProduktID Kurzform, Bezeichnung Volltextbezeichnung FROM Produkt;
```

Im gezeigten Statement wird nun die erste Spalte mit „Kurzform“ und die zweite Spalte mit „Volltextbezeichnung“ überschrieben.

Für den Fall, dass von einer bestimmten Spalte nur eineindeutige Werte ausgelesen werden, muss in dem `SELECT` Statement das Schüsselwort `DISTINCT` (oder als Synonym auch `DISTINCTROW`) angegeben werden.

```
SELECT DISTINCT Kontonummer FROM Kontenumsatz;
```

Damit wird jede existierende Wertausprägung der angegebenen Spalte nur einmal ausgegeben. Das `DISTINCT` kann auch über mehrere Spalten erweitert werden, womit die Eindeutigkeit über die Wertausprägungskombinationen der angegebenen Spalten sichergestellt wird:

```
SELECT DISTINCT Kontonummer, Datum FROM Kontenumsatz;
```

Eine weitere Einschränkungsmöglichkeit bei der Abfrage ist der Einsatz von Funktionen. Beispielhaft sei hier `MIN()` und `MAX()` angegeben. Weitere Beispiele werden im Kapitel Expressions behandelt.

```
SELECT MAX(Umsatzbetrag) FROM Kontenumsatz;
```

Dieser Befehl gibt den maximalen Umsatzbetrag der Tabelle Kontenumsatz aus. Da ein einzelner Betragswert nicht aussagekräftig ist, werden diese Funktionen im Regelfall bei gruppierten Abfragen (siehe Kapitel Aggregatsfunktionen) eingesetzt.

4.6.2 Joins

Eine große Stärke von SQL ist es, Daten miteinander in Relation zu bringen (daher der Name „relationale Datenbank“). Um dies zu bewerkstelligen wurden Joins eingeführt. Mit Hilfe eines Joins können mehrere Tabellen (oder allgemein Datenmengen) via Vergleichsoperatoren zusammengefügt werden. Gehen wir Beispielhaft davon aus, dass sich in der Tabelle Vorgang alle Geschäftsvorgänge des Systems befinden. Eine Beschreibung der Tabelle liefert folgende Informationen:

Spaltenname:	**Struktur:**	**Inhalt:**
VorgangID	INT (Primary Key)	Fortlaufende ID der Geschäftsvorgänge, automatisch von DBMS generiert (Primary Key).
PartnerID	INT	PartnerID des Vorgangs.
ProduktID	INT	ProduktID des Vorgangs.
RechnungID	INT	RechnungID des Vorgangs.
Datum	DATE	Datum der Rechnungsstellung.

Die Tabelle Produkt wiederum hält alle relevanten Produktinformationen und wird mit folgenden Informationen beschrieben:

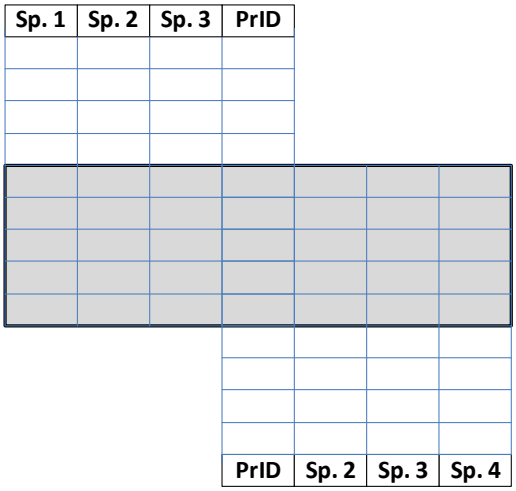
Spaltenname:	**Struktur:**	**Inhalt:**
ProduktID	INT (Primary Key)	Fortlaufende ID der Produkte, automatisch von DBMS generiert (Primary Key).
Bezeichnung	VARCHAR(30)	Produktbezeichnung.
Gewicht	DECIMAL(5,2)	Gewicht für Verpackungsberechnung.
Volumen	DECIMAL(5,2))	Volumen für Verpackungsberechnung.
Listenpreis	DECIMAL(8,2)	Offizieller Listenpreis.

Wenn man nun wissen möchte, zu welchen Vorgänge am 17.01.2010 eine Rechnungsstellung erfolgte und welche Produkte verkauft wurden, so müssen die Informationen der Produkttabelle und der Vorgangstabelle zusammengefügt werden. Dies geschieht mit dem folgenden Syntax:

```
SELECT * FROM Vorgang JOIN Produkt
```

```
   ON Vorgang.ProduktID = Produkt.ProduktID
WHERE Vorgang.Datum = '2010-01-17';
```

Der Befehl sieht eine Selektierung einer Datenmenge vor, welche aus einer Zusammenführung (JOIN) der Tabellen Vorgang und Produkt entstanden ist. Die Zusammenführung wurde unter der Bedingung gemacht, dass die Datensätze nur dann zusammengeführt werden, wenn die ProduktIDs der beiden Tabellen jeweils identisch sind. Grafisch kann dies wie Rechts dargestellt visualisiert werden. Die beiden Datenmengen aus den Tabellen werden so zusammengeführt, dass diejenigen Datensätze ausgegeben werden, für die die Bedingung Vorgang.ProduktID = Produkt.ProduktID zutrifft. Alle anderen werden nicht angezeigt. In der Ergebnismenge im MySQL Client ist zu erkennen, dass die Spalte ProduktID zweimal vorhanden ist – einmal von der Tabelle Vorgang und einmal von der Tabelle Produkt. Dies liegt daran, dass wir mit dem Befehl SELECT * alle in der abgefragten Datenmenge (also zwei Tabellen) existierenden Spalten auswählen.



Zu der oben genannten Notation gibt es noch weitere Alternativen. Eine ist die USING Klausel. Hierbei wird keine explizite Bedingung angegeben, sondern es wird angegeben, welche Spalten in beiden Tabellen gleich sein müssen. Dies gilt für die Wertinhalte, wie (gezwungener Maßen) auch für die Spaltennamen:

```
SELECT * FROM Vorgang JOIN Produkt
   USING(ProduktID)
WHERE Vorgang.Datum = '2010-01-17';
```

Bei gleichen Spaltennamen ist es ebenfalls Möglich, über den sogenannten NATURAL JOIN die Verknüpfung zu erstellen:

```
SELECT * FROM Vorgang NATURAL JOIN Produkt
WHERE Vorgang.Datum = '2010-01-17';
```

Anmerkung: Bei einem NATURAL JOIN ist Vorsicht geboten. Eine eventuelle nachträgliche Ergänzung von Feldern in beiden Tabellen kann zu sehr unvorhergesehenen Fehlergebnissen führen, ohne dass die Abfrage eine Fehlermeldung ausgibt!

Ein Join kann auch ohne das Schlüsselwort JOIN durchgeführt werden. Hierbei werden die Gleichheitsbedingungen explizit angegeben und gegebenenfalls mit AND verknüpft. Die beiden Tabellennamen werden nach dem FROM Schlüsselwort eingefügt und mit einem Komma getrennt.

```
SELECT * FROM Vorgang, Produkt
WHERE Vorgang.ProduktID = Produkt.ProduktID
  AND Vorgang.Datum = '2010-01-17';
```

Grundsätzlich sprechen wir bei den vorausgegangenen JOIN Statements von Inner Joins. Dies kann auch explizit angegeben werden:

```
SELECT * FROM Vorgang INNER JOIN Produkt
   USING(ProduktID)
WHERE Vorgang.Datum = '2010-01-17';
```

Wie sie sehen, gibt es in SQL verschiedene Wege, um das Gleiche zu erreichen. Bei Joins ist jedoch auch Vorsicht geboten! Sollten die Spalten, welche für die Identifikation der zusammenzufügenden Datensätze herangezogen werden nicht korrekt ausgewählt werden, bzw. ganz vergessen werden, so entstehen sogenannten Kreuzprodukte. Zur Erläuterung hier ein Beispiel. Die oberen Selects liefern alle die Vorgänge mitsamt Produkten, welche am 17.01.2010 verkauft

wurden. Wir erweitern die `WHERE` Klausel nun um eine weitere Bedingung – und zwar schränken wir die Abfrage auf Produkte mit dem Namen MiniMax2 ein:

```
SELECT * FROM Vorgang, Produkt
WHERE Vorgang.ProduktID = Produkt.ProduktID
  AND Vorgang.Datum = '2010-01-17'
  AND Bezeichnung = 'MiniMax2';
```

Die Datenbank liefert 2 Datensätze, für die die angegebenen Bedingungen zutreffen. Nun führen wir das Statement nochmal aus, „vergessen" jedoch die Joinbedingung:

```
SELECT * FROM Vorgang, Produkt
WHERE Vorgang.Datum = '2010-01-17'
  AND Bezeichnung = 'MiniMax2';
```

Wie sie in der Ergebnismenge sehen, erzeugt die Abfrage 11 Datensätze, bei denen die ProduktID aus der Vorgangstabelle auch andere Werte haben kann, als die ProduktID der Produkttabelle. Dies hat nun weitreichende Konsequenzen für fehlerhafte Abfragen! Eine Abfrage über zwei Tabellen ohne eine Joinbedingung gibt jede Kombination der existierenden Datensätze aus! Wir können dies mit folgenden drei Statements überprüfen:

```
SELECT COUNT(*) FROM Vorgang;
SELECT COUNT (*) FROM Produkt;
SELECT COUNT (*) FROM Vorgang, Produkt;
```

In der Tabelle Vorgang liegen 867 Datensätze. In der Tabelle Produkt liegen 20 Datensätze. Wenn nun beide Tabellen ausgewählt werden, ohne eine JOIN Bedingung zu formulieren, so erhalten Sie 17340 Datensätze, was nichts anderes ist als das Produkt von 20 * 867. Vor dem Hintergrund, dass auf Produktivsystemen mitunter Millionen von Datensätzen auf einer Tabelle liegen, können Sie sich vorstellen, dass eine fehlerhafte (oder nicht vorhandene) Joinbedingung zu sehr langen Datenbankprozessen führen kann.

Bei größeren (und vor allem verschachtelten) Select Abfragen ist es üblich, in den einzelnen Bedingungen nicht immer die Tabellennamen zu verwenden, sondern mit Aliasnamen zu arbeiten. Dies sind im Regelfall kürzere Zeichenketten, welche die Tabellen beschreiben um vor allem bei komplizierteren Join Bedingungen den Überblick nicht zu verlieren:

```
SELECT * FROM Vorgang v, Produkt p
WHERE v.ProduktID = p.ProduktID
  AND v.Datum = '2010-01-17'
  AND p.Bezeichnung = 'MiniMax2';
```

Die bisherigen Beispiele haben lediglich zwei Tabellen miteinander Verknüpft. Grundsätzlich funktioniert dies auch mit mehr als zwei Tabellen. Wenn bspw. zu der Abfrage nach MiniMax2 Produkten vom 17.01.2010 noch der Kunde mit angegeben werden soll, so kann dies mit den folgenden ergebnisgleichen Statements erreicht werden (zur Erleichterung bei der Schreibweise wurden in den Statements Aliasnamen verwendet):

```
SELECT * FROM
Vorgang v, Produkt p, Partner pa
WHERE v.ProduktID = p.ProduktID
  AND v.PartnerID = pa.PartnerID
  AND v.Datum = '2010-01-17'
  AND p.Bezeichnung = 'MiniMax2';
```

```
SELECT * FROM
Vorgang v JOIN (Produkt p, Partner pa)
  USING (ProduktID, PartnerID)
```

```
WHERE v.Datum = '2010-01-17'
  AND p.Bezeichnung = 'MiniMax2';
```

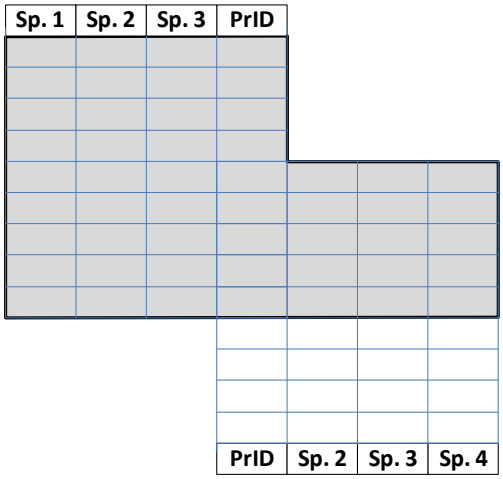
```
SELECT * FROM
Vorgang v NATURAL JOIN (Produkt p, Partner pa)
WHERE v.Datum = '2010-01-17'
  AND p.Bezeichnung = 'MiniMax2';
```

Die Schreibweise mit dem Schlüsselwort „ON“ ist bei drei Tabellen etwas komplizierter – hier wird oft eine Kombination mit der expliziten Schreibweise verwendet. Alternativ können Subselects verwendet werden (siehe weiter Unten).

```
SELECT * FROM
Vorgang v JOIN (Produkt p, Partner pa)
ON (v.ProduktID = p.ProduktID
AND v.PartnerID = pa.PartnerID)
WHERE v.Datum = '2010-01-17'
  AND p.Bezeichnung = 'MiniMax2';
```

MySQL nutzt bei diesen Joins die vorhandenen Indizes. Sollte bei der Optimierung festgestellt werden, dass MySQL die falschen Indexspalten nutzt, so kann dies mit den Begriffen `USE INDEX (Indexliste)` bzw. `IGNORE INDEX (Indexliste)` gesteuert werden.

Wenn es nun einen `INNER JOIN` gibt, so liegt es nahe, dass es auch einen `OUTER JOIN` gibt. Um dies zu verstehen, muss nochmal die Grafik herangezogen werden. Ein Inner Join (mit korrekter Joinbedingung) wirkt wie ein Filter auf die beiden verknüpften Datenmengen – nur diejenigen Datensätze werden ausgegeben, die auch in beiden Datenmengen laut Joinbedingung vorhanden sind. Es gibt nun Situationen, bei denen diese Filterfunktion nicht erwünscht ist. Es dürfen bspw. in der ersten Datenmenge keine Einträge im Ergebnisset fehlen. Hierzu verwendet man nun den `LEFT OUTER JOIN`. Dieser gibt nun alle Datensätze der ersten Datenmenge (bei `RIGHT OUTER JOIN` der zweiten Datenmenge) aus. Für diejenigen Datensätze, für die in der ersten (linken) Datenmenge kein Pendant in der zweiten (rechten) Datenmenge gefunden wurde, wird für die entsprechenden Spalten der Wert `null` gesetzt.



Doch wofür benötigt man derartige Funktionalitäten? Üblicherweise werden Outer Joins dafür verwendet, Datensätze zu finden, welche keine korrespondierenden in anderen Tabellen aufweisen. Wenn bspw. nun diejenigen Kunden gefunden werden sollen, welche noch keinerlei Vorgänge produziert haben, so ist dies mit folgendem Statement möglich:

```
SELECT pa.* FROM
Partner pa LEFT OUTER JOIN Vorgang v
ON (pa.PartnerID = v.PartnerID)
WHERE v.PartnerID IS NULL;
```

Es werden im obigen `SELECT` Statement also all diejenigen Partner gesucht, bei denen mittels eines `LEFT OUTER JOIN` die PartnerID in der Vorgangstabelle `null` ist. Dieses Statement kann auch ergebnisgleich mit einem `RIGHT OUTER JOIN` realisiert werden, wobei hier die Reihenfolge der Tabellen umgedreht werden muss:

```
SELECT pa.* FROM
Vorgang v RIGHT OUTER JOIN Partner pa
ON (pa.PartnerID = v.PartnerID)
WHERE v.PartnerID IS NULL;
```

Es gibt auch RDBMS Systeme, welche einen `FULL OUTER JOIN` beherrschen. Hier werden somit alle Datensätze von beiden Tabellen ausgegeben. Da dies in MySQL nicht vorgesehen ist, könnte bei Bedarf mittels `LEFT` und `RIGHT OUTER JOIN` und einer `UNION` ein derartiges Verhalten (mit einigen Einschränkungen) manuell erzeugt werden.

Zu Beachten ist noch, dass der Wert `NULL` nicht mittels einem Istgleich Zeichen „=“ abgefragt werden kann, sondern mit dem Statement `IS NULL`. Dies liegt daran, dass `NULL` kein eigentlicher Wert in der Datenbank ist, sondern lediglich ein Indikator, dass nichts vorhanden ist. Dieser muss entsprechend mit einer eigenen Funktionalität „`IS NULL`“ geprüft werden.

4.6.3 Löschstatement

Bis jetzt wurde nur darüber referiert, wie Daten in die Tabelle gelangen, und wie man sie ausliest. Im Zusammenhang mit dem `SELECT` Statement macht es jedoch auch Sinn darüber nachzudenken, wie man Daten wieder aus Tabellen entfernen kann. Da mitunter dedizierte Daten entfern werden müssen, liegt es nahe auch hier entsprechende Auswahlbedingungen für die zu löschenden Daten zu formulieren. Diese sind wiederum syntaxgleich zum `SELECT` Statement.

Für das dedizierte Löschen von Daten sieht SQL den Befehl `DELETE` vor. Dieser kann mit mehreren Tabellen, oder auch mit Einzeltabellen genutzt werden. Bei einzelnen Tabellen sieht der Syntax wie folgt aus:

```
DELETE FROM Partner
WHERE PartnerID = 1003;
```

Dieser Befehl löscht nun alle Einträge aus der Tabelle Partner, welche die PartnerID 1003 aufweisen. Sollte die `WHERE` Klausel vergessen werden, so löscht der Befehl alle Daten aus der Tabelle Partner! Dies kann gewünscht werden, jedoch wird aufgrund besserer Performance beim kompletten Löschen von Tabellen der Befehl `TRUNCATE` (also bspw. `TRUNCATE TABLE Partner;`) empfohlen.

Mitunter ist es notwendig, zusammenhängende Datensätze aus verschiedenen Tabellen zu löschen. Dies kann ebenfalls mit einem `DELETE` Statement und der entsprechenden Joinbedingung erreicht werden:

```
DELETE pa, v FROM Partner pa, Produkt p, Vorgang v
WHERE pa.PartnerID = v.PartnerID
  AND p.ProduktID  = v.ProduktID
  AND pa.PartnerID = 1048;
```

Das obige Statement löscht alle Einträge der Tabelle Partner und Vorgang, bei denen die PartnerID = 1048 ist. Sollte noch zusätzlich die Produkttabelle (mit Alias p) nach `DELETE` eingefügt werden, so werden auch sämtliche Produktinformationen gelöscht, welche der Partner mit der ID 1048 über die Vorgangstabelle in Verbindung gebracht wird:

```
DELETE pa, v, p FROM Partner pa, Produkt p, Vorgang v
WHERE pa.PartnerID = v.PartnerID
  AND p.ProduktID  = v.ProduktID
  AND pa.PartnerID = 1048;
```

Dies zeigt auch gleich die Brisanz des `DELETE` Befehls über mehrere Tabellen. Das Statement würde somit auch diejenigen Produkte löschen, die vielleicht für andere Kunden auch benötigt werden! Nur über den (zugegeben hier sehr konstruierten) Join zu Vorgang würden somit auch alle relevanten Produktdaten gelöscht. Dies ist im Regelfall jedoch nicht gewünscht. Seien Sie also bei der Nutzung von `DELETE` in Zusammenhang mit mehreren Tabellen extrem vorsichtig, da hier sehr schnell Dateninkonsistenzen entstehen können.

4.6.4 Union

Wenn man den Join als eine Zusammenfügung von Datenmengen in der Breite interpretieren möchte, so wäre die Union eine Zusammenfügung von Datenmengen in der Tiefe. Unions werden normalerweise genutzt, um Ergebnismengen von mehreren Abfragen in einer virtuellen Ergebnismenge zusammenzufügen. So liefert bspw. das folgende Statement eine Ansammlung aller Nachnamen und Produktbezeichnungen:

```
SELECT Bezeichnung String1 from Produkt
```

```
UNION
SELECT Nachname String1 from Partner;
```

MySQL ist relativ unempfindlich gegenüber unterschiedlichen Spalteneigenschaften der oberen und unteren Datenmenge. Trotzdem wird empfohlen, sowohl die Bezeichnungen, als auch die Datentypen in den zusammengefügten Datenmengen identisch zu halten.
Wichtig zu wissen ist bei MySQL noch, dass die Daten, welche in eine Union zusammengeführt werden distinct ausgewählt werden, also keine Duplikate ausgegeben werden.

4.6.5 Stringfunktionen

Mitunter ist es notwendig, Strings zu untersuchen. Hierzu stellt MySQL diverse Funktionen bereit. An dieser Stelle sind nur die wichtigsten vermerkt; weiterführende Informationen können aus dem MySQL Manual entnommen werden.

Oftmals ist es notwendig Strings aus verschiedenen Quellen anzugleichen, um sinnvolle Vergleiche durchzuführen. Hierzu bietet MySQL diverse Funktionalitäten. Die vielleicht wichtigste Kategorie sind die Trim Funktionen. `TRIM(str)` entfernt links und rechts des Strings alle Leerzeichen. `TRIM()` kann noch mit diversen Parametern justiert werden (nur Leerzeichen am Ende oder am Anfang – was auch durch `LTRIM()` und `RTRIM()` erreicht wird oder auch Ersetzung der Leerzeichen durch vorgegebene Ersatzcharacter.

```
SELECT TRIM(Bezeichnung) Bezeichnung FROM Produkt;
```

Auch die Umwandlung aller Buchstaben in Großbuchstaben oder Kleinbuchstaben dient der einfacheren Vergleichbarkeit von Strings. Hierzu stehen die Funktionen `LOWER(str)` und `UPPER(str)` zur Verfügung.

```
SELECT UPPER(Bezeichnung) Bezeichnung FROM Produkt;
```

Mit Hilfe der `LOCATE(substr,str,pos)` Funktion kann ein Substring in einem String ab einer bestimmten Position bestimmt werden. Der Rückgabewert ist der Positionsindex. Ohne der Positionsangabe wird ab der ersten Position gesucht, womit die Funktion identisch mit `INSTR(str, substr)` ist.

```
SELECT * FROM Produkt WHERE INSTR(Bezeichnung, 'master') > 0;
```

Eine weitere Möglichkeit ist es, mit Hilfe von „LIKE“ oder „NOT LIKE“ zu arbeiten. Hierbei kann man mit den Wildcards "%" und "_" arbeiten. "%" steht für eine beliebige Anzahl an beliebigen Zeichen und "_" für ein beliebiges Zeichen. Vor allem bei der Nutzung von "%" ist Vorsicht geboten, da man hier schnell zu viele Treffer bekommt. Hier zwei kurze Beispiele für die Nutzung von LIKE:

Alle Produkte, welche mit der Bezeichnung „M“ beginnen findet man mit:

```
SELECT * FROM Produkt WHERE Bezeichnung LIKE ("M%");
```

Alle Produkte, welche „Maxibooster “ gefolgt von einem beliebigen Zeichen heißen, findet man mit:

```
SELECT * FROM Produkt WHERE Bezeichnung LIKE ("Maxibooster _");
```

Die umfangreichsten Möglichkeiten für Stringvergleiche bietet jedoch die Nutzung von Regulären Ausdrücken. So können mit der Operation `REGEXP` Reguläre Ausdrücke auf Stringwerte angewendet werden. `REGEXP` liefert 0 für keine Übereinstimmung bzw. 1 für Übereinstimmung zurück. Für den Umfang der in MySQL umgesetzten Regulären Ausdrücke konsultieren Sie bitte das Manual.

```
SELECT * FROM Produkt WHERE Bezeichnung REGEXP'.*master.*' = 1;
```

Die Länge eines Strings wird mit `LENGTH(str)` ausgegeben:

```
SELECT LENGTH(Bezeichnung) Stringlaenge, Bezeichnung FROM Produkt
ORDER by Stringlaenge;
```

Dies kann zusammen mit `INSTR(str, substr)` für Substringfunktionalitäten wie `LEFT(str, len)`, `RIGHT(str, len)` bzw. `SUBSTRING(str, pos, len)` sinnvoll eingesetzt werden. `SUBSTR(str, pos, len)` liefert einen Substring des Strings `str`, der ab der Position `pos` mit der Länge `len` gebildet wird:

```
SELECT SUBSTRING(Bezeichnung, 2, 4) Substring, Bezeichnung FROM Produkt;
```

`LEFT(str, len)` ist gleichbedeutend mit `SUBSTRING(str, 1, len)`. `RIGHT(str, len)` ist gleichbedeutend mit `SUBSTRING(str, LENGTH(str) – len + 1, len)`.

Es können mit `CONCAT(str1,str2,...)` auch einzelne Strings zu einem neuen String zusammengefügt werden:

```
SELECT CONCAT(Vorname, ' ', Nachname) FROM Partner;
```

Dies ist z.B. dann nützlich, wenn in einer Tabelle der Vorname und Nachname getrennt durch ein Leerzeichen in einer Spalte liegen, in einer zweiten Tabelle Vorname und Nachname je in einer Eigenen Spalte vorliegen und beide Verknüpft werden müssen.

Schließlich gibt es noch Funktionen zur Stringmanipulationen, welche vornehmlich bei Outputfile Erzeugungen verwendet werden. Sollten Strings (aber auch andere Datentypen) mit Quotierungszeichen ausgegeben werden, so erledigt dies der Befehl `QUOTE(str)`:

```
SELECT QUOTE(Vorname) FROM Partner;
```

Eine weitere Hilfestellung kann der Befehl `REPLACE(str,from_str,to_str)` sein, wenn man bestimmte Zeichen in der Ergebnismenge escapen will. Der Befehl ersetzt im String `str` jedes Vorkommen `from_str` mit dem Ersatzsstring `to_str`:

```
SELECT REPLACE(Nachname, 'ß', 'ss'), Nachname FROM Partner;
```

Für Ausgabeformate im Fixed Length Format bietet sich der `LPAD(str,len,padstr)` bzw. der `RPAD(str,len,padstr)` Befehl an. Hier wird eine fixe Länge eines Strings angegeben. Sollte der String `str` kürzer sein, so wird er mit `padstr` aufgefüllt. Sollte er länger sein, so wird er abgeschnitten, wobei `LPAD` von links auffüllt und `RPAD` von rechts. Abgeschnitten wird jedoch immer von rechts.

```
SELECT RPAD(Nachname, 7, '#'), Nachname FROM Partner;
```

Schließlich soll zum Schluss an dieser Stelle noch der Verschlüsselungsbefehl angemerkt werden – auch wenn es hierzu schon einige Exploits gibt und die Literatur das Ausweichen auf andere Algorithmen nahelegt. Für kurzfristige Lösungen ist das in MySQL implementierte Verfahren auf jeden Fall sinnvoll nutzbar. Wenn auf Testsystemen für den Abnahmetest Produktivdaten eingesetzt werden müssen, diese aber rechtlich von dem Testteam nicht eingesehen werden dürfen, so müssen die Testdaten bei der Überleitung von der Produktivdatenbank auf die Testdatenbank verschlüsselt werden. Hierfür bietet MySQL den Befehl `AES_ENCRYPT(str, password)` an:

```
SELECT Vorname, AES_ENCRYPT(Vorname, 'irgendeinpasswort')
FROM Partner;
```

Was man theoretisch mit diesem Befehl auch bewerkstelligen kann ist, Userpasswörter einer Webanwendung verschlüsselt ablegen, um keine klartextlichen Passwörter auf der Datenbank ablegen zu müssen. Hier sei jedoch vermerkt, dass das Verschlüsselungspasswort der `AES_ENCRYPT` Funktion auf dem System hinterlegt werden muss – im Zweifelsfall als Systemvariablenwert. Da dies wiederum ein Sicherheitsrisiko darstellt, bieten Skriptengines wie PHP eigene Methoden an, um Passwörter verschlüsselt abzulegen.

4.6.6 Expressions und Funktionen

Neben den Stringfunktionen gibt es noch weitere Funktionen innerhalb MySQL, welche das Leben sehr erleichtern können. Unterschieden werden können diese zwischen Funktionen mit dem Fokus auf Zahlenmanipulationen, solche die innerhalb von `GROUP BY` Datenmengen verwendet werden können und dem großen Bereich der Datumsfunktionen (`GROUP BY` siehe Kapitel 4.6.9). Hier werden (wieder einmal) nur die wichtigsten Befehle gestreift – Detailinformationen liefert das MySQL Referenzhandbuch.

Zuerst kommen die Grundrechenarten. MySQL beherrscht alle notwendigen Operatoren, um rechnen zu können: `*, /, %, +, -`, wobei der Moduloperator `%` auch als `MOD(N,M)`, bzw. `N MOD M` genutzt werden kann.

```
SELECT (Gewicht / 1000) Grammgewicht FROM Produkt;
```

Die nächsten wichtigen Operatoren sind die Vergleichsoperatoren. Auch hier ist die Nutzung wie in anderen Programmiersprachen auch über `<, <=, >, >=, <>` (alternativ `!=`).

```
SELECT * FROM Produkt WHERE Gewicht > 1.0;
```

Schließlich gibt es noch die Logikoperatoren, welche im Gegensatz zu den gängigen Programmiersprachen üblicherweise klartextlich angegeben werden: `OR, XOR, AND, NOT`.

```
SELECT * FROM Produkt WHERE Gewicht >= 1.0 AND Gewicht <= 100;
```

Da Bereichsauswahlmuster relativ häufig vorkommen, gibt es für das obige Select Statement noch eine eingängigere Schreibweise:

```
SELECT * FROM Produkt WHERE Gewicht BETWEEN 1.0 AND 100;
```

Die nächsten Operatoren dienen der mathematischen Manipulation. Ein Blick auf das Referenzhandbuch von MySQL zeigt, dass hier sehr viele Möglichkeiten existieren – die wichtigsten sind die Befehle, welche die Zahlenwerte grundsätzlich manipulieren:

Der Absolutbetrag wird mit `ABS(x)` ausgelesen, wobei das Vorzeichen `mit SIGN(X)` wiederum extrahiert werden kann.

```
SELECT CONCAT(Umsatzbetrag, ' ist ',
SIGN(Umsatzbetrag), ' mal ', ABS(Umsatzbetrag))
FROM Kontenumsatz;
```

Runden wird mit dem Befehl `ROUND(X,D)` realisiert, wobei `X` der zu rundende Wert und `D` die Anzahl der Nachkommastellen angibt:

```
SELECT Umsatzbetrag, ROUND(Umsatzbetrag, 1) FROM Kontenumsatz;
```

Ohne Angabe der Nachkommastellen wird auf ganze Zahlen gerundet (ROUND(x) ist also gleichbedeutend mit `ROUND(x,0)`). Sollte lediglich der Wert Abgeschnitten werden, so wird dies mit `TRUNCATE(X,D)` erreicht.

Sollten die gerundeten Zahlen in Tausendergruppen ausgegeben werden, so muss `FORMAT(X,D)` verwendet werden. Hier wird ebenfalls gerundet – aber Achtung: das Ergebnis hat den Datentyp String.

Das Gruppieren von Datenmengen ist eine sehr wichtige und vielgenutzte Funktionalität. Dies hängt nicht zuletzt damit zusammen, dass die RDBMS Systeme hier noch eine Vielzahl von Zusatzfunktionen anbieten, welche das Interpretieren von Datenbankinhalten sehr erleichtern – die Aggregatsfunktionen. Da dies ein so zentrales Thema ist, wurde dem Gruppieren ein eigenes Kapitel 4.6.9 und den Aggregatsfunktionen ein eigenes Kapitel 4.6.10 gewidmet.

Die letzte hier besprochene Gruppe von Befehlen rankt sich um Datumsrechnung. Dies ist ein extrem weites Feld und es wird dringend geraten, dass wenn für Abfragen Datumsrechnungen benötigt werden, das Referenzhandbuch zum Thema Datumsfunktionen herangezogen wird, um zu sehen, ob für das aktuelle Problem eine geeignete Lösung angeboten wird. Die Wichtigsten Funktionen sind wieder Teil dieses Dokumentes.

Für das Nachvollziehen von Usereingaben in Webapplikationen wird gerne ein Zeitstempel abgelegt, wann bspw. ein Gästebucheintrag vorgenommen wurde. Dies kann mit der Funktion `NOW()` realisiert werden. Er liefert das Datum plus Zeitstempel. Sollte nur das Datum von Interesse sein, bietet sich der Befehl `CURDATE()` an. Ist nur die Zeit von Interesse, so nutzt man `CURTIME(), bzw NOW()`:

```
SELECT NOW();
```

Diese Befehle werden üblicherweise direkt in den `INSERT` Befehl mit integriert.

Für Zeitrechnungen gibt es eine Vielzahl von Befehlen. An dieser Stelle sollen nur die beiden Befehle `DATE_ADD(date,INTERVAL expr type)` und `DATE_SUB(date,INTERVAL expr type)` besprochen werden. Die Funktionen dienen dazu, zu einem Datum date einen Wert eines bestimmten Intervalls hinzuzuaddieren, oder abzuziehen. Hierbei ist die Angabe der Art des Intervalls notwendig. Anbei die in MySQL umgesetzten Intervallarten:

***type* Wert**	**Erwartetes *expr*-Format**	**Beispiel**
MICROSECOND	Mikrosekunden	100
SECOND	Sekunden	40
MINUTE	Minuten	15
HOUR	Stunden	2
DAY	Tage	14
WEEK	Wochen	3
MONTH	Monate	3
QUARTER	Quartale	1
YEAR	Jahre	13
SECOND_MICROSECOND	Sekunden.Mikrosekunden	40.100
MINUTE_MICROSECOND	Minuten.Mikrosekunden	15.100
MINUTE_SECOND	Minuten.Sekunden	15.40
HOUR_MICROSECOND	Stunden.Mikrosekunden	2.100
HOUR_SECOND	Stunden.Sekunden	2.40
HOUR_MINUTE	Stunden.Minuten	2.15
DAY_MICROSECOND	Tage.Mikrosekunden	14.100
DAY_SECOND	Tage.Sekunden	14.40
DAY_MINUTE	Tage.Minuten	14.15
DAY_HOUR	Tage.Stunden	14.2
YEAR_MONTH	Jahre.Monate	13.3

Sollte nun bspw. zu einem Datumswert zwei Wochen hinzuaddiert werden, sieht der Befehl wie folgt aus:

```
SELECT UmsatzID, Datum, DATE_ADD(Datum, INTERVAL 2 WEEK) FROM Kontenumsatz WHERE
Kontonummer_Partner = 410882995;
```

Oft ist es notwendig, Zeiträume zwischen zwei Datumswerten zu bestimmen. `DATEDIFF(expr,expr2)` liefert die Differenz zwischen zwei Datumswerten in Tagen:

```
SELECT UmsatzID, Datum, DATEDIFF(CURDATE(), Datum) FROM Kontenumsatz WHERE Kon-
tonummer_Partner = 410882995;
```

Für weiterführende Berechnungen kann auch der Tag des `Monats (DAYOFMONTH(date))`, Tag der Woche `(DAYOFWEEK(date))`, Tag im Jahr (`DAYOFYEAR(date)`) oder Kalenderwoche (`WEEKOFYEAR(date)`) ausgegeben werden.

```
SELECT UmsatzID, Datum,
DAYOFMONTH (Datum),
DAYOFWEEK (Datum),
DAYOFYEAR (Datum),
WEEKOFYEAR (Datum)
FROM Kontenumsatz WHERE Kontonummer_Partner = 410882995;
```

Weiterhin können Sie mit YEAR(date) das Jahr des Datums und mit Month(date) das Monat des Datums extrahieren. Mitunter ist das SQL Datumsformat für die Ausgabe nicht geeignet und muss umformatiert werden. Hierfür bietet MySQL die Funktion `DATE_FORMAT(date,format)` an, wobei hier das Datum date mit Hilfe der format – Angabe umformatiert wird.

Die Möglichkeiten von format sind im Folgenden Dargestellt:

Konfigurations-angabe	Beschreibung
%a	Abgekürzter Name des Wochentags (Sun ... Sat)
%b	Abgekürzter Name des Monats (Jan ... Dec)
%c	Monat, numerisch (0 ... 12)
%D	Tag im Monat mit englischem Suffix (0th, 1st, 2nd, 3rd, …)
%d	Tag im Monat, numerisch (00 ... 31)
%e	Tag im Monat, numerisch (0 ... 31)
%f	Mikrosekunden (000000 ... 999999)
%H	Stunde (00 ... 23)
%h	Stunde (01 ... 12)
%I	Stunde (01 ... 12)
%i	Minuten, numerisch (00 ... 59)
%j	Tag im Jahr (001 ... 366)
%k	Stunde (0 ... 23)
%l	Stunde (1 ... 12)
%M	Monatsname (January ... December)
%m	Monat, numerisch (00 ... 12)
%p	AM oder PM
%r	Uhrzeit im 12-Stunden-Format (hh:mm:ss gefolgt von AM oder PM)
%S	Sekunden (00 ... 59)
%s	Sekunden (00 ... 59)
%T	Uhrzeit im 24-Stunden-Format (hh:mm:ss)
%U	Woche (00 ... 53), wobei Sonntag der erste Tag der Woche ist
%u	Woche (00 ... 53), wobei Montag der erste Tag der Woche ist
%V	Woche (01 ... 53), wobei Sonntag der erste Tag der Woche ist; wird mit %X verwendet
%v	Woche (01 ... 53), wobei Montag der erste Tag der Woche ist; wird mit %x verwendet
%W	Name des Wochentags (Sunday ... Saturday)
%w	Tag in der Woche (0=Sonntag ... 6=Sonnabend)
%X	Jahr der Woche, wobei Sonntag der erste Tag der Woche ist, numerisch, vierstellig; wird mit %V verwendet
%x	Jahr der Woche, wobei Montag der erste Tag der Woche ist, numerisch, vierstellig; wird mit %v verwendet
%Y	Jahr, numerisch, vierstellig
%y	Jahr, numerisch, zweistellig
%%	Literales '%'-Zeichen
%*x*	*x*, steht für jedes nicht oben aufgeführte '*x*'

Ein Select Befehl mit Datumsumformatierung könnte somit wie folgt aussehen:

```
SELECT UmsatzID, Datum, DATE_FORMAT(Datum, '%a, der %d.%m.%Y')
FROM Kontenumsatz;
```

Zum Schluss noch der Hinweis, dass der Formatierungsbefehl als Ausgabetyp kein Datum mehr sein kann, da das Datumsformat in MySQL als YYYY-MM-TT definiert ist. Insofern kann der Ausgabetyp von DATE_FORMAT nur String sein. Daran schließt sich die Frage an, wie kann man aus einem String wiederum ein Date Datentyp machen. Dies erledigt der Befehl `STR_TO_DATE(str,format)`:

```
SELECT STR_TO_DATE('26.01.2010', '%d.%m.%Y');
```

4.6.7 Typenkonvertierungen

Wir haben gesehen, dass auch bei SQL die Beachtung von Datentypen wichtig ist. Insofern muss SQL auch Möglichkeiten zur Verfügung stellen, Daten von einem Datentyp in einen anderen umzuwandeln. Wie bei anderen Programmiersprachen auch, versucht MySQL bei fehlender Typecastangabe einen impliziten Typecast durchzuführen – was mitunter zu Problemen führen kann. Explizit wird der Typecast mit dem Befehl `CAST(expr AS type)` durchgeführt, wobei type folgende Ausprägungen annehmen kann:

- `BINARY[(N)]`

- CHAR[(N)]
- DATE
- DATETIME
- DECIMAL
- SIGNED [INTEGER]
- TIME
- UNSIGNED [INTEGER]

Darüber hinaus gibt es noch die bereits besprochenen Befehle `STR_TO_DATE(str,format)` und `DATE_FORMAT(date,format)`.

Ein wichtiger Befehl – der zwar streng genommen kein Typecastbefehl ist, jedoch in diese Gruppe einordenbar ist – stellt `COALESCE(value, …)` dar. Der Befehl erwartet eine Liste an Werten und gibt den ersten Wert ungleich null zurück. Der null Wert ist bei diversen Operationen ein Sonderfall, der entsprechend ausgeklammert werden müsste. Mit Hilfe des `COALESCE` Befehls kann dem `null` Wert ein anderer Wert zugeordnet werden:

```
SELECT COALESCE(null, 1), COALESCE(2,1);
```

4.6.8 Ablaufsteuerung

Für komplexere Funktionalitäten bietet SQL noch weiterreichende Ablaufsteuerungsfunktionen an. Diese sind nicht mit Stored Procedures zu verwechseln!

Zuerst sei der Befehl `CASE` (auch Mehrfachauswahl) genannt. Dieser wird in zwei verschiedenen Arten genutzt. Die den klassischen Programmiersprachen ähnlichste Nutzung ist die `CASE` Abfrage auf ein dediziertes Feld, welches zu einem Ausgabewert führt. Wie im Beispiel unten zu sehen, prüft die `CASE` Anweisung, ob das Feld Status die Werte FK, WK, MA, PK, LF oder keines der genannten aufweist. Wenn ein entsprechender Treffer gefunden wurde, so wird der Wert nach `THEN` ausgegeben.

```
SELECT Vorname, Nachname,
CASE Status
  WHEN 'FK' THEN 'Firmenkunde'
  WHEN 'WK' THEN 'Wunschkunde'
  WHEN 'MA' THEN 'Mitarbeiter'
  WHEN 'PK' THEN 'Privatkunde'
  WHEN 'LF' THEN 'Lieferant'
  ELSE 'Unbekannt'
END
FROM Partner;
```

In der zweiten Form wird `CASE` als eine Aneinanderreihung von Bedingungen interpretiert. Wenn eine Bedingung erfüllt ist (Suchreihenfolge ist von oben nach unten), dann wird der Wert nach `THEN` ausgegeben.

Der `ELSE` Zweig fängt den Fall, ab, dass keine Bedingung erfüllt sein könnte.

```
SELECT Kontonummer Partner, Umsatzbetrag,
CASE
  WHEN Umsatzbetrag > 0 THEN 'Ueberweisung'
  WHEN Umsatzbetrag < 0 THEN 'Abbuchung'
  ELSE 'Keine Transaktion'
END
FROM Kontenumsatz;
```

Der `CASE` Befehl wird immer mit `END` abgeschlossen. In den einzelnen Abschnitten müssen nicht, wie im Beispiel Angegeben immer Konstantwerte stehen. Es gibt auch die Möglichkeit Spaltenwerte einzusetzen:

```
SELECT Kontonummer_Partner, Kontonummer, Umsatzbetrag,
```

```
CASE
  WHEN Umsatzbetrag > 0 THEN Kontonummer_Partner
  WHEN Umsatzbetrag < 0 THEN Kontonummer
  ELSE 'Keine Transaktion'
END
FROM Kontenumsatz;
```

Der zweite Befehl in dieser Kategorie ist der `IF` Befehl - dieVerzweigung. Dieser funktioniert relativ intuitiv:

```
SELECT Umsatzbetrag, Kontonummer_Partner, Kontonummer,
IF (Umsatzbetrag > 0, Kontonummer_Partner, Kontonummer)
FROM Kontenumsatz;
```

`IF` Befehle können auch geschachtelt werden:

```
SELECT Umsatzbetrag, Kontonummer_Partner, Kontonummer,
IF (Umsatzbetrag > 0, Kontonummer_Partner, IF (Umsatzbetrag < 0, Kontonummer,
'Keine Transaktion'))
FROM Kontenumsatz;
```

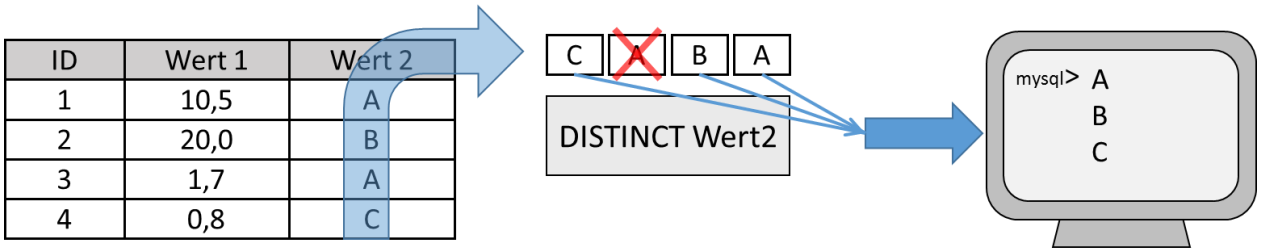
Somit erledigt das oben genannte IF Statement das gleiche, wie der letzte CASE Befehl.

4.6.9 Datengruppierung

Eine sehr wichtige Funktion in SQL Selects ist `GROUP BY`. Dieser bewirkt, dass alle selektierten Daten in Gruppen zusammengefasst werden und lediglich ein Datensatz pro Gruppe ausgegeben wird. Folgende Abfrage gruppiert alle Datensätze der Tabelle Partner entsprechend ihres Statuswertes und gibt je Gruppe einen Statuswert aus:

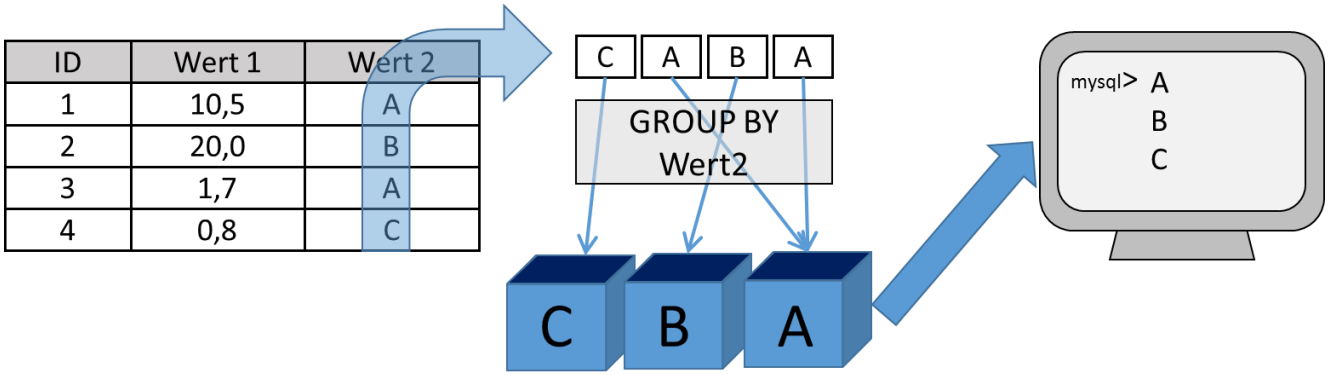
```
SELECT Status FROM Partner GROUP BY Status;
```

Das Ergebnis ist identisch mit einem DISTINCT Select – es werden alle Kundengruppen (Firmenkunden, Privatkunden etc.). Die Funktionalität dahinter ist jedoch eine grundlegend andere! Hier die Vorgehensweise bei „DISTINCT“:



Es werden also alle selektierten Werte der angegebenen Spalte (hier „Wert2“) aus der Tabelle gelesen, welche bis dato noch nicht in der Ergebnismenge sind. Alle anderen werden verworfen. Hier im Beispiel taucht der Wert „A“ zweimal auf. Insofern wird der zweite nicht berücksichtigt und aus der Ergebnismenge entfernt.

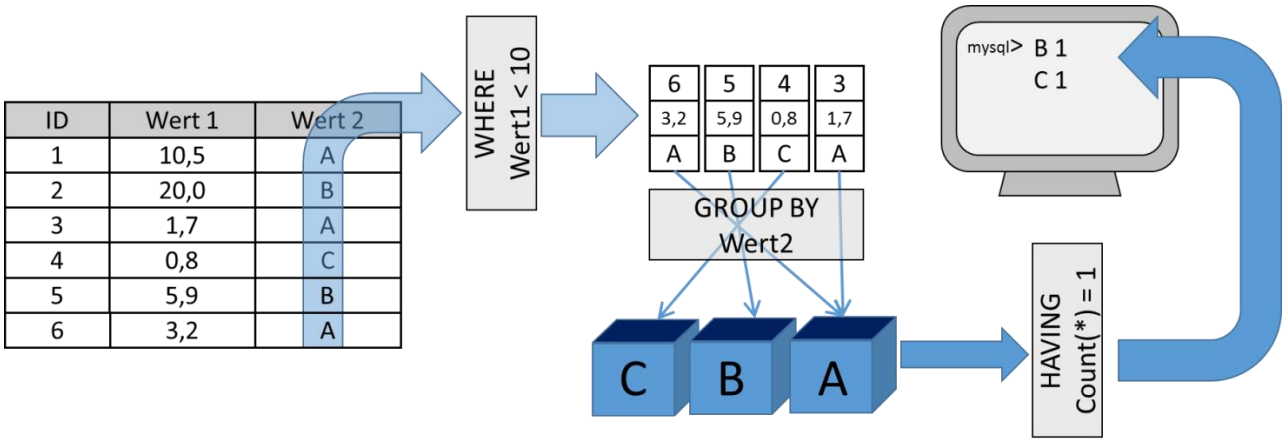
Führen wir die gleiche Abfrage mit „GROUP BY“ und der entsprechenden Spalte durch, so sehen wir am Bildschirm zwar das gleiche Ergebnis, jedoch erfolgt die Ermittlung über eine Gruppenbildung:



Auch hier werden alle Datensätze von der Tabelle gelesen. Der Unterschied besteht darin, dass die Ergebnismenge nicht mehr auf Datensatzlevel basiert, sondern auf einem Gruppenlevel. Für jeden unterschiedlichen Wert der Gruppierten Spalte (hier A, B und C) wird ein Gruppenbehälter erzeugt. Diese nehmen jeden Datensatz auf, der zu der Gruppe gehört. In dem Beispiel liegen nun im Gruppenbehälter „A“ zwei Datensätze, in den Gruppenbehältern „B“ und „C“ jeweils nur einer. Auf dem Bildschirm werden nun nicht mehr einzelne Datensätze angezeigt, sondern nur noch die Gruppenbehälter. Wichtig ist hier noch zu erwähnen, dass die Anzeige von nicht gruppierten Werten (also bspw. ID) keinen Sinn mehr macht, weil wir die Granularität der Anzeige auf Ebene der Gruppen haben. Die Anzeige des Feldes „ID“ von Gruppe „A“ ist sinnlos, weil wir dort ja zwei Datensätze (und somit zwei IDs haben).

Wir können zwar aufgrund der Gruppenbehälter nicht mehr auf den einzelnen Datensatz zugreifen, der große Vorteil dieser Vorgehensweise ist jedoch, dass wir auf die Gruppeneigenschaften Zugriff haben. Beispielsweise können wir nun die Anzahl der Datensätze in den Gruppenbehältern mit count(*) zählen. Grundsätzlich funktionieren alle Aggregatsfunktionen auf Gruppenebene (siehe nächstes Kapitel).

Eine wichtige Funktion des GROUP BY Befehls ist die „HAVING“ Klausel. Diese filtert die Gruppenbehälter entsprechend ihrer Eigenschaften aus. Dies ist nicht zu verwechseln mit der eigentlichen Filterfunktion „WHERE“, welche die Eingangsdaten filtert.
Wenn wir nur Datensätze berücksichtigen wollen, für die bspw. der Wert1 kleiner als 10 sein soll, so formulieren wir dies mit „WHERE Wert1 < 10“. Wenn wir darüber hinaus nur die Behälter haben wollen, in denen sich nur ein Datensatz befindet, so schreiben wir „HAVING count(*) = 1“.



Die Für den Fall, dass die Tabelle den Namen Tab1 trägt, würde die Abfrage also wie folgt lauten:

```
SELECT Wert2 FROM Tab1 WHERE Wert1 < 10 GROUP BY Wert2 HAVING COUNT(*) = 1.
```

Um GROUP BY sinnvoll nutzen zu können, sollte das nächste Kapitel ebenfalls gelesen werden – dort sind auch konkrete Beispiele auf der Übungsdatenbank angegeben.

4.6.10 Aggregatsfunktionen

Um den eigentlichen Nutzen der Gruppierung aus dem vorausgegangenen Kapitel zu verstehen, muss nun eine Ergänzung gemacht werden. Die Gruppe, von denen nun pro Gruppe immer nur ein Datensatz angezeigt wird, beinhaltet ja im Hintergrund immer noch alle Datensätze welche der Gruppe zugehörig sind – sie werden nur nicht angezeigt. Aus diesem Grund macht es keinen Sinn Attribute anzeigen zu lassen, welche für die einzelnen Datensätze innerhalb der Gruppe unterschiedliche Ausprägungen aufweisen. Wenn bspw. der Vorname ausgegeben werden soll, so wird ein zufälliger

Datensatz aus der Gruppe für die Anzeige des Vornamens ausgewählt. Dies ist nicht sinnvoll nutzbar. Es ist sinnlos zu fragen, wie ist der Vorname der Gruppe der Firmenkunden. Sinnvoll ist es jedoch, Eigenschaften der Gruppe auszuwählen. Eine solche Eigenschaft kann zum Beispiel die Anzahl der der Datensätze pro Gruppen sein (also, wie viele Firmenkunden, Privatkunden etc. gibt es):

```
SELECT Status, COUNT(*) FROM Partner GROUP BY Status;
```

Es ist nun noch möglich, innerhalb der Gruppen nur unterschiedliche Ausprägungen eines Datensatzes (oder auch mehrerer Datensätze) zu zählen – hier hilft der Zusatz DISTINCT:

```
SELECT Kontonummer, COUNT(DISTINCT Kontonummer_partner)
FROM Kontenumsatz GROUP BY Kontonummer;
```

Folgendes Beispiel nutzt zwar `GROUP BY`, erzeugt jedoch nicht brauchbare Daten:

```
SELECT * FROM Kontenumsatz GROUP BY Kontonummer;
```

Wenn Sie diesen Befehl ausführen, so sehen Sie, dass lediglich zwei Datensätze zurückgeliefert werden. Dies liegt daran, dass in der Tabelle in der Spalte Kontonummer auch nur zwei verschiedene Werte zu finden sind (213323443 und 213323450). Um nun sinnvolle Informationen zu erlangen könnte nun bspw. eine Aufsummierung des Umsatzbetrages sein:

```
SELECT Kontonummer, SUM(Umsatzbetrag) SummeUmsatz
FROM Kontenumsatz
GROUP BY Kontonummer;
```

Dieser Befehl summiert somit pro Kontonummer die Umsatzbeträge und gibt sie zusammen mit der Kontonummer aus. Es können bei der Auswahl der entsprechenden Datensätze auch Bedingungen angegeben werden:

```
SELECT Kontonummer, SUM(Umsatzbetrag) SummeUmsatz
FROM Kontenumsatz
WHERE Umsatzbetrag > 0
GROUP BY Kontonummer;
```

Hier werden also nur die positiven Umsätze pro Kontonummer aufsummiert. `GROUP BY` erlaubt auch Gruppierungen über mehrere Spalten. So kann bspw. über Kontonummer und Kontonummer_Partner gruppiert werden:

```
SELECT Kontonummer, Kontonummer_Partner, SUM(Umsatzbetrag) SummeUmsatz
FROM Kontenumsatz
GROUP BY Kontonummer, Kontonummer_Partner;
```

Sollten noch bestimmte Einschränkungen auf die `GROUP BY` Ergebnismenge angewendet werden, so ist dies mit dem `HAVING` Zusatz möglich:

```
SELECT Kontonummer, Kontonummer_Partner, SUM(Umsatzbetrag) SummeUmsatz
FROM Kontenumsatz
GROUP BY Kontonummer, Kontonummer_Partner
HAVING SummeUmsatz > 0;
```

Hier werden nur diejenigen Datensätze an den Client weitergegeben, welche einen summierten Umsatzbetrag größer als 0 aufweisen. Details zu „HAVING“ sind im vorausgegangenen Kapitel.

Ein weiterer Befehl, welcher auf Gruppenebene funktioniert ist der Befehl `COUNT(*)`, bzw. `COUNT(col)`, wobei *col* für eine beliebige Spalte steht. Hier werden pro Gruppe die Anzahl der Zeilen (bei `COUNT(*)`) bzw. bei `COUNT(col)` die Anzahl der Werte ungleich NULL angezeigt.

Weitere Funktionen, welche auf Gruppenebene Funktionieren sind die `MIN(expr)` bzw. `MAX(expr)` und die `AVG(expr)` Funktion. Diese liefern pro Gruppe den minimalen, maximalen bzw. durchschnittlichen Wert pro Gruppe (siehe Oben).

```
SELECT Kontonummer, MIN(Umsatzbetrag) SummeUmsatz
FROM Kontenumsatz
GROUP BY Kontonummer;
```

4.6.11 Subqueries

Unterabfragen (engl. Subqueries) können in bestimmter Form in der `WHERE` Klausel auftreten. So kann z.B. mit Hilfe einer Subquery eine Eigenschaft für eine Bedingung ermittelt:

```
SELECT k.* FROM Kontenumsatz k
WHERE Umsatzbetrag = (SELECT MAX(Umsatzbetrag) FROM Kontenumsatz);
```

Diese Form von Unterabfragen kann noch erweitert werden, indem die Auswahl der Unterabfrage auch mehrere Datensätze haben kann:

```
SELECT k.* FROM Kontenumsatz k
WHERE Umsatzbetrag = ANY (SELECT Listenpreis FROM Produkt);
```

Das Schlüsselwort `ANY` kann nach jedem Vergleichsoperand eingesetzt werden (=, <, >, <>, <=, >=). `ANY` ist jedoch nicht zu verwechseln mit `ALL`. `ALL` liefert nur dann ein true zurück, wenn die Bedingung für alle Ergebnisse der Unterabfrage gültig ist!

Subqueries werden häufig in Zusammenhang mit Aggregationsbefehlen verwendet. Hierbei wird mittels einem `SELECT` Befehl eine Datenmenge erzeugt, auf die wiederum ein `SELECT` ausgeführt werden kann.

```
SELECT * FROM
  (SELECT Kontonummer_Partner, SUM(Umsatzbetrag) SummeUmsatz
   FROM Kontenumsatz
   WHERE Umsatzbetrag > 0
   GROUP BY Kontonummer_Partner
  ) As Temp
WHERE SummeUmsatz > 6000;
```

Üblicherweise muss das innere Select (die Subquery) mit einem eigenen Namen versehen werden, um die Informationen außerhalb der Subquery referenzieren zu können. Das Oben genannte Statement liefert bspw. alle Einträge von Kundenkonten, welche einen summierten positiven Umsatzbetrag von mehr als 6000 aufweisen.

Subqueries können auch in Joins eingehen – zur Erinnerung: SQL bezieht sich auf Datenmengen, welche Tabellen sein können, aber nicht sein müssen!

Im Folgenden Select wird zu der Partnertabelle der Vor- und Nachname des zugehörigen Sachbearbeiters zugeordnet. Nachdem jedoch nicht alle Personen aus der Partnertabelle in den Join eingehen sollen, so wird zuerst ein Subselect durchgeführt, welches ausschließlich die Mitarbeiter auswählt. Diese Submenge geht in den Join ein.

```
SELECT ma.VornameSachb, ma.NachnameSachb, p.* FROM
  Partner p,
  (SELECT p.PartnerID, p.Vorname AS VornameSachb, p.Nachname AS NachnameSachb
    FROM Partner p
    WHERE p.Status = 'MA'
  ) ma
WHERE p.SachbearbeiterID = ma.PartnerID;
```

Eine weitere Art, Subselects zu verwenden ist die Klausel `IN` oder `NOT IN`.

```
SELECT p.* FROM
  Partner p
WHERE p.PartnerID NOT IN
  (SELECT DISTINCT v.PartnerID
   FROM Vorgang v);
```

Hierbei wird die (nicht)Existenz eines Wertes in einer Datenmenge geprüft. Wichtig bei der zu prüfenden Datenmenge ist, dass sie einspaltig ist. Wenn das Subselect bspw. wie folgt lauten würde: `SELECT DISTINCT v.PartnerID, v.ProduktID`
würden zwei Spalten vorhanden sein (PartnerID und ProduktID), womit die `NOT IN` Klausel nicht mehr funktioniert.

Die relevante Datenmenge kann auch explizit als Konstante Menge angegeben werden:

```
SELECT p.* FROM
  Partner p
WHERE p.Status IN
  ('FK', 'WK');
```

5 Stored Procedures / Functions

5.1 Allgemeines

Relationale Datenbanksysteme sind im Regelfall in einer normalisierten Form aufgebaut, um Dateninkonsistenzen und Datenredundanzen möglichst zu vermeiden. Oftmals führt dies jedoch zu unübersichtlichen Datenbanken, welche sich damit aus Nutzersicht als nur bedingt brauchbar darstellen. Es sind mitunter viele Joins notwendig, um an die gewünschten Daten zu kommen. Für immer wiederkehrende Zusammenhänge macht es somit Sinn, sich die Arbeit der komplizierten Joins nur einmal zu machen und sie wiederverwertbar in dem RDBMS zu speichern. Hierfür werden üblicherweise zwei Konzepte verfolgt:

- Stored Procedures/Functions
- Views (wird in einem späteren Kapitel erläutert)

Stored Procedures bzw. Stored Functions (gespeicherte Prozeduren und Funktionen) sind Programme mit Anweisungen ähnlich einer prozeduralen Programmiersprache, wobei die einzelnen RDBMS Hersteller hier auf z.T. proprietäre Programmiersprachen zurückgreifen. Wie so vieles haben Stored Procedures sowohl Vor- als auch Nachteile. Als Vorteile währe zu nennen:

- Sie bieten im Regelfall eine hohe Performance, da sie vorkompilliert sind.
- Da die Logik auf der Datenbank umgesetzt werden kann, reduziert sich oftmals der Datenverkehr zwischen Datenbank und Applikation.
- Datenbankrelevante Funktionalitäten werden dort zentralisiert, wo sie hingehören – auf die Datenbank.
- Die Berechtigungen von Stored Procedures können individuell eingestellt werden.

Nachteilig wirkt sich folgendes aus:

- Der RAM – Verbrauch steigt mit Stored Procedures merklich an.
- Die Entwicklung ist mitunter kompliziert, da der Befehlsumfang eher beschränkt ist.
- Ein einfach zu handhabendes Debugging ist nicht vorgesehen.

Grundsätzlich muss situationsabhängig entschieden werden, ob sich der Aufwand für die Entwicklung von Stored Procedures lohnt.

Wichtig ist noch, dass die Stored Procedures und Functions von den einzelnen RDBMS Herstellern anders implementiert werden und wir hier ausschließlich auf die MySQL Notation eingehen. Dies ist zwar im Vergleich zu anderen Herstellern relativ einfach gehalten (bspw. sind keine rekursiven Aufrufe möglich), aber für das Verständnis soll es ausreichend sein.

5.2 Stored Procedures

Stored Procedures zeichnen sich dadurch aus, dass sie komplette Funktionalitäten kapseln und über Parameter Daten empfangen, bzw. auch zurückgeben können. Ein einfaches Beispiel für eine Stored Procedure für MYSQL sieht wie folgt aus:

```
CREATE PROCEDURE GetAllProducts()
 BEGIN
   SELECT * FROM produkt;
 END ;
```

Das Problem hier ist, dass bei der Konsoleneingabe der Interpreter das Semikolon als das Ende des Statements ansieht und den Befehl abschickt, bevor das END; erreicht wurde. Um dies zu umgehen verändert man temporär den Delimiter vom Semikolon auf ein anderes Zeichen (üblicherweise $$):

```
DELIMITER $$
```

Danach erfolgt die Stored Porcedure und am Ende muss entsprechend der Delimiter wieder zurück auf das Semikolon geführt werden. Der Code sieht somit wie folgt aus:

```
DELIMITER $$
```

```
CREATE PROCEDURE GetAllProducts()
 BEGIN
   SELECT * FROM produkt;
 END $$
DELIMITER ;
```

Auch können mit Stored Procedures kompliziertere Funktionalitäten durchgeführt werden, welche nicht in den Standardfunktionen des RDBMS vorhanden sind, wie folgendes Beispiel zeigt[5]:

```
DELIMITER $$
 CREATE PROCEDURE INSERT_SPACER(INOUT str VARCHAR(500))
 BEGIN
   DECLARE i INT DEFAULT 1;
   DECLARE outstr VARCHAR(1000) DEFAULT '';
   WHILE i <= CHAR LENGTH(str) DO
     SET outstr = CONCAT(outstr, SUBSTRING(str, i, 1), ' ');
     SET i = i + 1;
   END WHILE;
   SET str = outstr;
 END$$
 DELIMITER ;
```

Aufgerufen werden diese Prozeduren wie folgt:

```
CALL GetAllProducts();
```

bzw.:

```
SET @str = 'Ich lerne SQL';
CALL INSERT_SPACER(@str);
SELECT @str;
```

5.3 Stored Functions

Wenn die Funktionalität im Rahmen eines normalen SQL Statements genutzt werden soll, gibt es auch die Möglichkeit anstatt einer Procedure eine Function zu deklarieren. Diese liefern ein Ergebnis zurück – wodurch sich die Notwendigkeit des „RETURN“ Statements ergibt:

```
DELIMITER $$
 CREATE FUNCTION INSERT_SPACER2(str VARCHAR(500)) RETURNS VARCHAR(1000)
 BEGIN
   DECLARE i INT DEFAULT 1;
   DECLARE outstr VARCHAR(1000) DEFAULT '';
   WHILE i <= CHAR_LENGTH(str) DO
     SET outstr = CONCAT(outstr, SUBSTRING(str, i, 1), ' ');
     SET i = i + 1;
   END WHILE;
   RETURN outstr;
 END$$
 DELIMITER ;
```

[5] http://www.mysqltutorial.org

Der Aufruf der Funktion innerhalb eines Select Statements sieht dann entsprechend einfach aus:

```
SELECT INSERT_SPACER2(Vorname) FROM Partner;
```

Bei Funktionen müssen Sie darauf achten, dass Sie bei der Benamung keine Konflikte mit Standardfunktionen erzeugen.

5.4 Anzeige / Löschen von Procedures und Functions

Eine Liste aller Procedures erhalten Sie mit folgendem Statement:

```
SHOW PROCEDURE STATUS;
```

Für Functions gilt ensprechendes:

```
SHOW FUNCTION STATUS;
```

Die Struktur einer Procedure (bzw. Function) erhalten Sie mit folgendem Befehl:

```
SHOW CREATE PROCEDURE INSERT_SPACER;
SHOW CREATE FUNCTION INSERT_SPACER2;
```

Gelöscht wird eine Procedure (bzw. Function) mittels dem DROP Statement:

```
DROP PROCEDURE INSERT_SPACER;
DROP FUNCTION INSERT_SPACER2;
```

5.5 Parameter

Parameter dienen dazu, Werte dynamisch an die Stored Procedures bzw. an die Stored Functions zu übergeben. Die Parameter stehen immer in Klammern nach dem Procedure/Function Namen. Nach dem Parameternamen folgt ein Datentyp, welcher 1:1 die Datentypen sein dürfen, wie sie für die Tabellenerstellung erlaubt sind. Beim Syntax ist allerdings die Nutzung bei Stored Procedures und Stored Functions zu unterscheiden:

Stored Functions: Hier werden die Parameter direkt mit Datentyp angegeben. Sie dienen ausschließlich dazu, Werte in die Function hinein zu tragen. Der einzige Rückgabewert steht nach der Parameterliste in der Form „RETURNS *Datentyp*“, wobei der Datentyp wieder ein beliebiger Datenbank-Datentyp sein darf.

Stored Procedures: Bei Stored Procedures können die Parameter Werte in die Procedure hineintragen („IN“), Werte zurückgeben („OUT“) oder beides („INOUT“). Diese drei Schlüsselwörter stehen vor dem eigentlichen Parameter. Da über die Parameterliste Werte auch zurückgegeben werden können, hat die Stored Procedure keine „RETURN“ Anweisung.

5.5.1 IN Parameter

Wenn Werte in eine Stored Procedure übergeben werden müssen, so wird dies über den „IN“ Parameter erledigt. Sie dienen ausschließlich zur Parameterübergabe und eine Veränderung des Wertes wird nicht von den aufrufenden Elementen registriert.

```
DELIMITER $$
CREATE PROCEDURE GetPartner(IN myPartnerID INT)
  BEGIN
    SELECT *
    FROM Partner
    WHERE PartnerID = myPartnerID;
  END $$
DELIMITER ;
```

```
CALL GetPartner(1144);
```

Das Verhalten entspricht also einem „Call by Value“.

5.5.2 OUT Parameter

Für den Fall, dass Werte von einer Procedure zurückgegeben werden müssen, bietet MySQL einen „OUT“ Parameter an. Dieser wird tatsächlich ausschließlich für die Wertrückgabe genutzt – was im Vergleich bei Programmiersprachen wie bzw. „Java“ so nicht vorgesehen ist.

```
DELIMITER $$
CREATE PROCEDURE NoOfMa(
     IN myMaPartnerID INT, OUT numberOfMa INT)
BEGIN
   SELECT count(*)
   INTO numberOfMa
   FROM Partner
   WHERE SachbearbeiterID = myMaPartnerID;
END$$
DELIMITER ;

CALL NoOfMa(1163,@numberOfMa);
SELECT @numberOfMa;
```

Um nun in der aufrufenden Struktur mit der Variable arbeiten zu können, wird beim Prozeduraufruf ein „@“ vorangestellt, was den Wert innerhalb der Session nutzbar macht. Das Verhalten ist also vergleichbar mit Session-Variablen (siehe Unten). Über ein einfaches SELECT auf die Variable kann der Wert ausgelesen werden.

Innerhalb der Prozedur können Werte mit „SELECT INTO“ oder mit einem „SET“ belegt werden. Details werden weiter Unten im Kapitel „Variablen“ Behandelt.

5.5.3 INOUT Parameter

Sind „IN“ Variablen einem „Call by Value” gleichzusetzen, sind „INOUT“ Variablen als „Call by Reference“ zu interpretieren. Hier können Werte übergeben, verändert und wieder zurückgegeben werden.

```
DELIMITER $$
CREATE PROCEDURE doCountUp(INOUT myCount INT,
     IN direction boolean)
   BEGIN
     IF direction = TRUE THEN
       SET myCount = myCount + 1;
     ELSE
       SET myCount = myCount - 1;
     END IF;
   END$$
DELIMITER ;

SET @myCount = 1;
CALL doCountUp(@myCount, TRUE);
CALL doCountUp(@myCount, TRUE);
CALL doCountUp(@myCount, FALSE);
SELECT @myCount;
```

Um dies zu ermöglichen muss eine Session-Variable erstellt und initialisiert werden (SET @myCount = 1). Diese kann im Aufruf übergeben und innerhalb der Prozedur verändert werden. Beim anschließenden SELECT auf die Variable wird der veränderte Wert ausgegeben.

5.6 Variablen

Variablen dienen zum temporären Abspeichern von Informationen. Hierbei unterscheiden wir zwischen drei verschiedenen Typen:

- „Parameter“ wie sie in den vorangegangenen Kapiteln beschrieben wurden
- Lokale Variablen
- Session-Variablen

Grundsätzlich gilt, dass innerhalb der Procedures und Functions Werte über zwei Arten belegt werden können:
„SET“ Statement. Hiermit wird in eine Variable der Wert rechts neben dem Istgleichzeichen übernommen:

```
SET myVariable = 20;
```

Der zweite Weg ist über ein „SELECT INTO“:

```
SELECT count(*)INTO numberOfPa FROM Partner;
```

Die Werte werden wieder ausgelesen, indem sie entweder in Ausdrücken verwendet werden:

```
… WHERE SachbearbeiterID = myMaPartnerID;
```

oder indem der Wert direkt selektiert wird:

```
SELECT myVariable;
```

wodurch der Wert auf der Konsole erscheint.

5.6.1 Lokale Variablen

Lokale Variablen werden innerhalb von Stored Procedures und Stored Functions deklariert, wobei sie hierbei auch einen Defaultwert (also Initialisierungswert) erhalten.

```
DECLARE myVariable INT DEFAULT 0;
```

Die Gültigkeit reicht hierbei bis zum „END“ Statement der einschießenden Struktur. Wenn die Variablen also am Anfang einer Stored Procedure deklariert werden, gelten sie bis zum „END“ Statement der Stored Procedure. Werden sie innerhalb Loops oder IF- Anweisungen deklariert, dann bis zum „END“ Statement dieser Strukturen. Sobald die Variable nicht mehr gültig ist, kann der Name neu vergeben werden.

```
DELIMITER $$
  CREATE PROCEDURE SetMyVariable()
    BEGIN
      DECLARE myVariable INT DEFAULT 0;
      SELECT myVariable;
      SET myVariable = 20;
      SELECT myVariable;
    END $$
DELIMITER ;

CALL SetMyVariable();
```

5.6.2 Session-Variablen

Im Gegensatz zu lokalen Variablen gelten Session-Variablen während der gesamten Session des Clients. Gerade bei Applikationen, welche sich ggf. Connections teilen (Stichwort „pooling“) ist hier darauf zu achten, dass die Belegung der Werte mit Bedacht erfolgt, da sonst ein Sicherheitsrisiko entstehen könnte. Weiterhin werden Session-Variablen nicht deklariert, sondern lediglich „gesetzt“. Man erkennt Session-Variablen am vorangestellten „@“- Zeichen.

```
SET @mySessionVariable = 5;

DELIMITER $$
   CREATE PROCEDURE ChangeMySessionVariable()
     BEGIN
       SELECT @mySessionVariable;
       SET @mySessionVariable = 20;
       SELECT @mySessionVariable;
     END $$
DELIMITER ;

CALL ChangeMySessionVariable();

SELECT @mySessionVariable;
```

5.7 Logik

Logikelemente dienen zur Ablaufsteuerung. Sie entsprechen dem Syntax der Ablaufsteuerung der SQL Befehle aus den vorausgegangenen Kapiteln für SQL. Bei Stored Procedures/Functions kommt diesen Elementen eine besondere Bedeutung zu, da sie elementar für prozedurale Programmierung sind.

5.7.1 IF Statement

Das einfachste Element zur Ablaufsteuerung ist die Verzweigung. Hier wird mit „IF“ eine Prüfung durchgeführt, welche entweder wahr, oder falsch sein kann. Insofern ist es hier auch möglich mit boolschen Operatoren wie „AND“ bzw. „OR“ und entsprechender Klammerung zu arbeiten.

```
DELIMITER $$
    CREATE FUNCTION ExceedsLimit(myProductID INT)
RETURNS BOOLEAN
     BEGIN
       DECLARE myWeight INT DEFAULT 0;
       SELECT Gewicht into myWeight FROM Produkt
       WHERE ProduktID = myProductID;
       IF myWeight > 100 THEN
          RETURN TRUE;
       ELSE
          RETURN FALSE;
       END IF;
     END $$
DELIMITER ;

SELECT ExceedsLimit(206704);
SELECT ExceedsLimit(505600);
```

Der “falsch” Zweig wird mit “ELSE” eingeleitet. Das gesamte Statement wird mit „END IF;“ beendet. Es ist auch möglich mit „ELSE IF…“ zu arbeiten, um die Verschachtelung bei komplexeren Abfragen zu minimieren.

5.7.2 Case - Statement

Eine alternative zum “IF” Statement ist das “CASE”. Wie auch in anderen Programmiersprachen können hier verschiedene Fälle abgeprüft werden. Jeder Fall wird hier mit „WHEN“ eingeleitet. Anders als bspw. in Java unterscheiden wir hier zwei verschiedene Varianten des „CASE“ Statements. Steht nach dem „CASE“ Schlüsselwort eine Variable, so wird der Inhalt der Variable ausgewertet. Dies erfolgt, indem die Werte zwischen „WHEN“ und „THEN“ mit dem Inhalt der geprüften Variablen verglichen werden. Dies entspricht dem Standardverhalten von Switch/Case Anweisungen. Der Defaultfall wird über „ELSE“ abgehandelt.

```
DELIMITER $$
   CREATE FUNCTION GetType(myProductID INT)
RETURNS VARCHAR(6)
     BEGIN
       DECLARE myWeight INT DEFAULT 0;
       SELECT Gewicht into myWeight FROM Produkt
         WHERE ProduktID = myProductID;
       CASE myWeight
         WHEN 0.0 THEN RETURN 'Nix';
         ELSE RETURN 'Was';
       END CASE;
     END $$
DELIMITER ;

SELECT GetType(206704);
SELECT GetType(602788);
```

Eine zweite Variante des “CASE” Statements wird realisiert, indem nach dem Schlüsselwort “CASE” keine Variable angegeben wird. Hier wird zwischen „WHEN“ und „THEN“ ein boolscher Ausdruck eingesetzt, welcher entsprechend ausgewertet wird. Dies entspricht im Prinzip einer IF / ELSE IF Anweisung:

```
DELIMITER $$
   CREATE FUNCTION GetType(myProductID INT)
RETURNS VARCHAR(6)
     BEGIN
       DECLARE myWeight INT DEFAULT 0;
       SELECT Gewicht into myWeight FROM Produkt
         WHERE ProduktID = myProductID;
       CASE
         WHEN myWeight < 50 THEN RETURN 'Leicht';
         WHEN myWeight > 100 THEN RETURN 'Schwer';
         ELSE RETURN 'Mittel';
       END CASE;
     END $$
DELIMITER ;

SELECT GetType(206704);
SELECT GetType(602788);
```

5.8 Schleifen

Schleifen sind ein Hautpunterscheidungsmerkmal zwischen einfachem SQL und Stored Procedures/Functions. Das iterative durcharbeiten von Werten ist bei Standard-SQL nur begrenzt möglich. Hier können die Procedures/Functions

also ihre Stärken zeigen. Wie in vielen Programmiersprachen auch, gibt es hier kopf- bzw. fußgesteuerte Schleifen. Weiterhin existiert noch eine allgemeine Form, welche recht flexibel einsetzbar ist.

5.8.1 Kopfgesteuerte Schleife

Wie der Name schon sagt, erfolgt die Steuerung der Schleife am Anfang – also die Prüfung, ob die Schleife wiederholt werden muss. Das entsprechende Schlüsselwort ist „WHILE“, gefolgt von einem boolschen Ausdruck. Der Rumpf der Schleife befindet sich dann zwischen dem „DO“ und dem „END WHILE;“.

```
DELIMITER $$
CREATE PROCEDURE PrintX(IN myNoOfX INT)
  BEGIN
    DECLARE str  VARCHAR(255);
    SET str =  '';
    WHILE myNoOfX > 0 DO
      SET str = CONCAT(str,'X');
      SET myNoOfX = myNoOfX - 1;
    END WHILE;
    SELECT str;
  END$$
DELIMITER ;

CALL PrintX(3);
```

5.8.2 Fußgesteuerte Schleife

Das Gegenstück zur kopfgesteuerten Schleife ist die fußgesteuerte Schleife. Diese beginnt mit dem Schlüsselwort „REPEAT“ und endet mit „END REPEAT;“. Vor diesem Befehl muss jedoch die Wiederholbedingung stehen, welche mit „UNTIL“ eingeleitet wird:

```
DELIMITER $$
CREATE PROCEDURE PrintY(IN myNoOfY INT)
  BEGIN
    DECLARE str  VARCHAR(255);
    SET str =  '';
    REPEAT
      SET str = CONCAT(str,'Y');
      SET myNoOfY = myNoOfY - 1;
    UNTIL myNoOfY <= 0
    END REPEAT;
    SELECT str;
  END$$
DELIMITER ;

CALL PrintY(3);
```

5.8.3 Allgemeine Schleife

Eine allgemeine Schleife ist vom Prinzip her eine Endlosschleife, welche dediziert abgebrochen werden kann. Die Schleife benötigt einen Namen, welcher wie folgt festgelegt wird:

```
my_loop_name: LOOP
```

Alle Kontrollbefehle müssen den Namen mitliefern. Wir unterscheiden hier zwei Kontrollbefehle – das „ITERATE“ und „LEAVE“.

„ITERATE“ bewirkt, dass die Schleife sofort zum Anfang springt, also den Teil unterhalb des „ITERATE“ auslässt. Beim Aufruf muss wie erwähnt, der Schleifenname mitgeliefert werden:

```
ITERATE  my_loop_name;
```

Dies ist deshalb notwendig, weil bei verschachtelten Schleifen klar sein muss, welche gemeint ist. Gleiches gilt für „LEAVE“, was die Schleife beendet:

```
LEAVE my_loop_name;
```

Eine komplette Schleife würde somit wie folgt aussehen:

```
DELIMITER $$
CREATE PROCEDURE PrintZ(IN myNoOfZ INT)
  BEGIN
    DECLARE str  VARCHAR(25) DEFAULT '';
    my_loop_name: LOOP
      SET myNoOfZ = myNoOfZ - 1;
      IF (myNoOfZ mod 4) THEN SET str = CONCAT(str,'-');
        ITERATE  my_loop_name;
      END IF;
      SET str = CONCAT(str,'Z');
      IF (myNoOfZ <= 0) THEN LEAVE my_loop_name;
      END IF;
    END LOOP;
    SELECT str;
  END$$
DELIMITER ;
CALL PrintZ(6);
```

5.9 Cursor

Ein sehr wichtiges Element für Stored Procedures/Functions ist der Cursor. Ein Cursor ist eine Art Zeiger, welcher auf den aktuellen Datensatz einer ganzen Liste von Datensätzen zeigt. MySQL Cursors weisen folgende Eigenschaften auf:

- Read-Only – also sie können nur gelesen, nicht geschrieben werden.
- Nicht scrollable – sie müssen also sequenziell von oben nach unten abgearbeitet werden. Der Cursor kann nicht manuell auf eine bestimmte Position geschoben werden.
- „Asensive“, was soviel bedeutet, dass der Cursor auf die tatsächlichen Daten auf der Datenbank zeigt. „Insensive“ Cursors würden auf eine temporäre Kopie verweisen. Die Konsequenz ist, dass bei einer temporären Kopie der Datenbestand sich während der Verarbeitung nicht ändern wird. Bei MySQL hat man zugunsten der Performance auf diese Kopie verzichtet, wodurch wir im Zweifelsfall zusätzliche Schritte unternehmen müssen, um Dateninkonsistenzen zu vermeiden.

Um einen Cursor zu nutzen, muss er zuerst deklariert werden. Dies erfolgt nach allen Variablendeklarationen und vor den Wertzuweisungen. Wird dieser Forderung nicht entsprochen, so wird MySQL sehr wahrscheinlich eine Fehlermeldung beim Erstellen der Stored Procedure ausgeben. Der Deklarationssyntax sieht wie folgt aus:

```
  DECLARE nameCursor CURSOR FOR
    SELECT Nachname FROM Partner WHERE
      SachbearbeiterID = myMaPartnerID;
```

Der Cursor erhält einen Namen und eine Definition. Diese ist ein SELECT Statement, welches in unserem Beispiel eine Liste mit einer Spalte ausgibt. Sie wird dem Cursor zugewiesen.

Der nächste Schritt ist einen Handler zu definieren, welcher beim Erreichen des Datenendes auslöst. Handler sind ein weiterer, umfangreicher Bestandteil von Stored Procedures/Functions, auf den mit Hinblick auf unseren Fokus hier nicht weiter eingegangen wird. Wir erstellen an dieser Stelle lediglich einen Handler, welcher eine Variable auf „1“ setzt, sobald der Cursor zum Ende gelangt ist:

```
  DECLARE CONTINUE HANDLER FOR
    NOT FOUND SET isDone = 1;
```

Anmerkung: Bei verschachtelten Cursors muss diese Stelle etwas aufwändiger gestaltet werden – wobei auch dies außerhalb unseres Fokusbereiches liegt.

Nun muss das Statement gestartet werden, so dass der Cursor aktiv wird:

```
  OPEN nameCursor;
```

Die Werte werden mittels „FETCH“ in die Variablen geschrieben. Da wir nur eine Spalte in unserem Statement verarbeiten, wird auch nur eine Variable benötigt:

```
    FETCH nameCursor INTO myName;
```

Hätten wir ein SELECT mit mehr als einer Spalte, so müssten nach dem „INTO“ mehrere Variablen kommasepariert stehen. Wichtig ist nun noch zu verstehen, dass wir die Verarbeitung unserer Datenliste in einem LOOP realisieren müssen, da wir pro FETCH nur einen Zeile unserer Datenmenge erhalten. An dieser Stelle setzt nun auch unser Handler an, welcher die Variable „isDone“ auf 1 setzt, sobald wir mit dem Auslesen aller Datensäte fertig sind. Dies wiederum beendet unsere Schleife:

```
DELIMITER $$
CREATE PROCEDURE buildList (IN myMaPartnerID INT)
BEGIN
  DECLARE isDone INT(1) DEFAULT 0;
  DECLARE myName VARCHAR(30) DEFAULT '';
  DECLARE nameCursor CURSOR FOR
    SELECT Nachname FROM Partner WHERE
      SachbearbeiterID = myMaPartnerID;
  DECLARE CONTINUE HANDLER FOR
    NOT FOUND SET isDone = 1;
  OPEN nameCursor;
  SET @myList = "";
  get_names: LOOP
    FETCH nameCursor INTO myName;
    IF isDone = 1 THEN LEAVE get_names; END IF;
    SET @myList = CONCAT(myName,";", @myList);
  END LOOP get_names;
 CLOSE nameCursor ;
END$$
DELIMITER ;

CALL buildList(1163);
SELECT @myList;
```

6 Views

Die gängigste Methode Abfragelogik zu kapseln sind Views. Das grundsätzliche Handling von Views bei den verschiedenen RDBMS ist vergleichbar, da Views im Wesentlichen auf SQL Befehlen aufbauen.

6.1 Grundgedanke

Eine View ist nichts anderes als ein Select Befehl, welcher innerhalb der Datenbank gespeichert wird und somit wiederverwendbar ist. Dies birgt folgende Vorteile:

- Sichten auf Daten können für den Anwender gekapselt und in der Tiefe angepasst werden
- Die einzelnen Kapselungen können wiederum in anderen Selects und somit auch Views verwendet werden
- Die Komplexität von normalisierten Tabellen kann mittels Views wieder denormalisiert und nutzerfreundlich dargestellt werden
- Der Query-Optimizer muss lediglich bei der Erzeugung der View optimieren, bei der Verwendung der View kann auf diese Optimierung einfach zugegriffen werden[6].

Ein Beispiel:

```
SELECT p.Vorname, p.Nachname, v.*
       FROM Vorgang v, Partner p
       WHERE v.PartnerID = p.PartnerID;
```

kann als View (hier mit dem Namen *Demoview*) erzeugt werden:

```
CREATE VIEW Demoview AS
       SELECT p.Vorname, p.Nachname, v.*
       FROM Vorgang v, Partner p
       WHERE v.PartnerID = p.PartnerID;
```

Der Zugriff erfolgt danach über ein einfaches SELECT Statement:

```
SELECT * FROM Demoview;
```

Das untere SELECT ist vom Ergebnis her somit identisch mit dem oberen Select.

6.2 Syntax in MySQL[7]

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = {user | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

CREATE	Erzeugt ein neues Datenbankobjekt (hier eine neue View)
REPLACE	Ersetzt eine existierende View durch eine neue, gleichen Namens
ALGORITHM	Optionale Klausel für die Ausführung der View innerhalb einer Anweisung: *MERGE*: Hier wird die Anweisung der View mit der Anweisung, welche die View aufruft verschmolzen, um eventuelle weitere Optimierungen durchführen zu können. *TEMPTABLE*: Die Anweisungsergebnisse der View werden in eine temporäre Tabelle geschrieben, worauf anschließend die Anweisung, welche die View nutzt, zugreift.

[6] Je nach Queryoptimizer kann dies jedoch trotzdem bei Laufzeit nochmals notwendig sein.
[7] Siehe http://www.mysql.com/

	UNDEFINED: MySQL entscheidet selbst zwischen MERGE und TEMPTABLE, wobei im Zweifelsfall MERGE vorgezogen wird, da es im Regelfall effizienter ist.
DEFINER	Optionale Klausel, um den Erzeuger der View zu setzen. Nur mit dem SUPER –Recht kann ein Konto (*'user_name'@'host_name'*) angegeben werden, welches nicht das eigene ist, sonst darf eben nur das eigene explizit oder mit CURRENT_*USER* angegeben werden.
SQL SECURITY	Optionale Klausel mit der angegeben wird, mit welchem Recht die View auf die Ressourcen zugreift. Standard bei Nichtangabe ab MySQL 5.1.2 ist *DEFINER*. Dies bedeutet, dass wenn User A die View erzeugt und User B die View nutzt, so kann er damit via View auf Tabellen zugreifen, auf die eigentlich nur User A die Zugriffsrechte besitzt. Der Wert *INVOKER* würde dazu führen, dass nur die Userrechte von User B greifen.
VIEW	Gibt an, dass eine View erzeugt werden soll. Nach VIEW folgt der (eindeutige) Viewname und eine optionale Spaltenliste (column_list) – siehe Unten wg. Eindeutigkeit der Spaltennamen.
AS	Gibt an, dass nun die eigentliche Anweisung der View beginnt. Dies kann eine beliebige MySQL konforme Select Anweisung sein[8].
WITH	Optionale Klausel, welche die Überprüfungstiefe bei der Ausführung festlegt. Dies kann *LOCAL CHECK OPTION* sein, wobei somit nur die eigene View geprüft wird, oder *CASCADED CHECK OPTION*, bei dem die Prüfung auch auf die Views ausgeweitet wird, welche von der gerade definierten View genutzt werden (siehe hier auch MySQL Manual)

Hierbei ist folgendes zu beachten:
Da Views in der Handhabung im SELECT wie Tabellen agieren, dürfen keine Spaltennamen doppelt vorkommen. Dies bedeutet, dass wenn in zwei Tabellen jeweils eine Spalte mit gleichem Namen existiert, wird folgendes Statement fehlschlagen:

```
CREATE VIEW Demoview2 AS
        SELECT *
        FROM Vorgang v, Produkt p
        WHERE v.ProduktID = p.ProduktID;
```

Um dies zu umgehen, müssen die einzelnen Spalten – zumindest von einer Tabelle – explizit benamt werden:

```
CREATE VIEW Demoview2 AS
        SELECT v.VorgangID, v.PartnerID, v.ProduktID AS ProduktID2,
                v.RechnungID, v.Datum, p.*
        FROM Vorgang v, Produkt p
        WHERE v.ProduktID = p.ProduktID;
```

6.3 Update über Views

Wie der Name schon sagt, sind Views in erster Linie dazu gedacht, Sichten über Daten zu erstellen. Dies bedeutet, dass im Zweifelsfall über Views keine Updates gemacht werden können. MySQL erlaubt zwar Updates über Views, jedoch mit diversen Einschränkungen.

Grundsätzlich gilt, dass man bei Updates über Views in Gefahr läuft, Dateninkonsistenzen zu erzeugen und es wird von der Nutzung dieser Möglichkeit abgeraten.

6.4 Anzeige und Löschen von Views

Eine Liste aller Views erhalten Sie, wenn Sie auf die MySQL Systemtabellen zurückgreifen:

```
SELECT table_name FROM information_schema.views;
```

Die Struktur eines Views erhalten Sie mit folgendem Befehl:

```
SHOW CREATE VIEW Demoview2;
```

[8] In MySQL ist dies jedoch nur mit diversen Einschränkungen möglich (http://dev.mysql.com/doc/refman/5.1/de/view-restrictions.html)

Gelöscht wird ein View mittels dem DROP Statement:

```
DROP VIEW Demoview2;
```

7 Trigger

Ein weiteres Konstrukt für einen effizienten Umgang mit Datenbanken sind sogenannte Trigger. Diese sind in MySQL zwar nur rudimentär implementiert, jedoch sind sie trotz des im Vergleich zu anderen RDBMS limitierten Funktionsumfanges auch bei MySQL ein sehr elegantes Mittel Daten konsistent zu halten. Prinzipiell sind die Funktionalitäten von Views ähnlich wie Stored Procedures aufgebaut, wenngleich sie aber keine Parameter besitzen, sondern basierend auf geänderten Datenwerten agieren.

7.1 Grundgedanke

Ein Trigger „hört" auf Datenmanipulationen (INSERT, UPDATE, DELETE) in Tabellen und wird beim Auftreten einer solchen Manipulation ausgeführt. Die Aktion eines Triggers ist wiederum eine SQL Anweisung – im Regelfall eine Anweisung, welche Daten in Tabellen einfügt oder verändert. Dies erklärt somit auch die Bedeutung von Triggern:

- Trigger können automatisiert Informationen über Datenbankaktivitäten generieren (Bspw. Eintrag in eine eigene Tabelle A, wenn die Anzahl der neuen Datensätze in Tabelle B über eine gewisse Grenze steigt).
- Mit Hilfe von Triggern können Daten konsistent gehalten werden (Bspw. wenn ein Kunde ab einem gewissen Umsatzbetrag als VIP Kunde geführt wird, kann ein Trigger diesen Update vornehmen)

Ein Trigger ist im Wesentlichen durch Standard SQL Befehle aufgebaut. Hier ein Beispiel[9]:

```
CREATE TRIGGER testref BEFORE INSERT ON test1
  FOR EACH ROW BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3 WHERE a3 = NEW.a1;
    UPDATE test4 SET b4 = b4 + 1 WHERE a4 = NEW.a1;
  END;
```

Ein Beispiel, welches auf unserer Übungsdatenbank läuft könnte wie folgt aussehen:

```
DELIMITER $$
CREATE TRIGGER CheckWunschkunde AFTER INSERT ON Vorgang
  FOR EACH ROW BEGIN
   IF (SELECT Status FROM Partner WHERE PartnerID = NEW.PartnerID) = 'WK' THEN
     UPDATE Partner SET Status = 'PK';
   END IF;
  END$$
DELIMITER ;
```

Das Statment prüft nach dem Eintragen eines neuen Datensatzes in die Vorgangstabelle, ob im dazugehörigen Partner noch ‚WK' als Wunschkunde steht. Da dieser Kunde nun etwas gekauft hat ist er kein Wunschkunde mehr, sondern mindestens ein Privatkunde. Insofern ändert das Statement den Status von ‚WK' auf ‚PK'. Prüfen können wir das Statment, wenn wir zuerst den Status des Kunden mit der ID 1081 prüfen:

```
SELECT Status FROM Partner WHERE PartnerID = 1081;
```

Anschließend fügen wir einen neuen Datensatz in Vorgang ein:

```
INSERT INTO Vorgang
   SET PartnerID = 1081, ProduktID = 833356,
   RechnungID = 92345, Datum = '2011-03-12';
```

[9] http://dev.mysql.com/doc/refman/5.1/de/create-trigger.html

Wenn wir den Status nun wieder prüfen sehen wir, dass er sich nun auf ‘PK’ geändert hat.

7.2 Syntax in MySQL[10]

```
CREATE
    [DEFINER = {user | CURRENT_USER }]
    TRIGGER trigger_name trigger_time trigger_event
    ON tbl_name FOR EACH ROW trigger_stmt
```

CREATE	Erzeugt ein neues Datenbankobjekt (hier einen neuen Trigger)
DEFINER	Optionale Klausel, um den Erzeuger des triggers zu setzen. Nur mit dem SUPER –Recht kann ein Konto (*'user_name'@'host_name'*) angegeben werden, welches nicht das eigene ist, sonst darf eben nur das eigene explizit oder mit CURRENT_USER angegeben werden. Standardmäßig ist *CURRENT_USER* gesetzt. Die Ausführung des Triggers wird mit den Berechtigungen des gesetzten DEFINERS durchgeführt.
TRIGGER	Gibt an, dass ein Trigger erzeugt werden soll, gefolgt von dem eindeutigen Triggernamen und der Regelung der Auslösung. Hier sind zwei Informationen notwendig: Triggerzeitpunkt: Dieser kann *BEVORE* oder *AFTER* sein, was bedeutet, dass die Triggeraktion vor oder nach dem auslösenden Ereignis ausgeführt wird. Triggerevent, oder Trigger Ereignis: Dies kann *INSERT* für alle INSERT, LOAD DATA oder REPLACE Anweisungen, es kann *UPDATE* für alle updates oder *DELETE* für alle DELETE oder REPLACE Anweisungen sein.
ON	Gibt die Tabelle an, auf der nach dem eingestellten Trigger Ereignis gehört wird.
FOR EACH ROW	Weist auf die Aktion hin, welche für jede entsprechend des Trigger Ereignis eingefügte, veränderte oder gelöschte Zeile durchgeführt wird. Auf Inhalte dieser Zeile kann man mit entweder OLD.Spaltenname oder NEW.Spaltenname zugreifen. Hierbei zeigt OLD auf den Datensatz vor der Änderung oder Löschung und NEW auf den Datensatz nach der Änderung oder Einfügung.

Wenn Sie die existierenden Trigger anzeigen möchten, so können Sie dies mit folgendem Code tun:

```
SHOW TRIGGERS;
```

Mit folgendem Statement wird die Struktur eines einzelnen Triggers angezeigt:

```
SHOW CREATE TRIGGER CheckWunschkunde;
```

Gelöscht wird ein Trigger mittels dem DROP Statement:

```
DROP TRIGGER CheckWunschkunde;
```

8 User auf der Datenbank

MySQL bringt eine eigene Userverwaltung mit. User können sich gegen eine Datenbank verbinden und Aktionen ausführen. Dies ist wichtig zu verstehen, da bspw. in einem Onlineportal der Datenbankuser der Applicationserver, oder das PHP Script ist und nicht die User, welche sich über das Onlineportal einloggen!
Der erste User, mit dem wir zu tun hatten, war der „root“ User. Dieser darf alles und hat sämtliche Rechte. Aus Sicherheitsgründen sollten die einzelnen Applikationen aber niemals über den „root“ User gegen die Datenbank gehen, da bei eventuellen Sicherheitslücken in der Serversoftware eventuelle Angreifer via der Serversoftware somit als „root“ User großen Schaden anrichten könnten. Insofern sollten für die einzelnen Aktionen auch eigene User angelegt werden.

8.1 Erzeugen von Usern

Ein User wird mit dem „CREATE USER“ Statement angelegt:

```
CREATE USER 'User1'@'localhost' IDENTIFIED BY 'secretpwd';
```

[10] Siehe http://www.mysql.com/

Hier wird ein User mit dem Login “User1” angelegt. Dieser darf sich über den localhost (also 127.0.0.1) am System anmelden. Er muss sich mit dem Passowrt „secretpwd“ authentifizieren. Neben dem localhost können noch dedizierte IP Adressen angegeben werden. Das „%“ Zeichen gilt in diesem Zusammenhang als Wildcard.

Anmerkung: Auf Windows Systemen gilt das „%“ nur für nicht localhost Adressen.

Gerade der Umgang mit den Passwörtern kann noch sehr viel genauer spezifiziert werden. Hier sei aber an das MySQL Manual verwiesen.

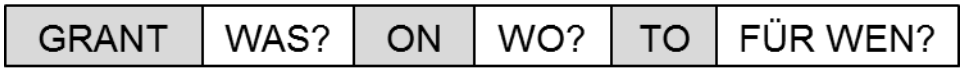
8.2 Einräumen von Rechten

Ist ein User angelegt, so können nun die Zugriffsrechte eingetragen werden. Hierbei nutzen wir das „GRANT“ Statement.

```
GRANT SELECT ON germac.partner TO 'User1'@'localhost';
```

Das Statement würde dem User “User1”, verbunden über den localhost ein Leserecht (SELECT) auf die Tabelle „Partner“ in der Datenbank “germac“ einräumen.

Das Statement ist somit wie folgt aufgebaut:



Gehen wir die einzelnen Punkte durch. Wir beginnen mit dem Recht (oder auch „Privileg“) genannt, also dem „WAS?“:

Privileg	**User hat das Recht:**
ALL	auf alle Privilegien außer GRANT OPTION
ALTER	„ALTER TABLE“ auszuführen.
ALTER ROUTINE	Stored Procedures und Functions zu ändern oder zu löschen.
CREATE	Datenbanken und Tabellen zu erzeugen.
CREATE ROUTINE	Stored Procedures und Functions zu erzeugen.
CREATE USER	User zu erzeugen, umzubenennen bzw. die Rechte zu ändern.
CREATE VIEW	Views zu erzeugen oder zu ändern.
DELETE	Daten zu löschen.
DROP	Datenbanken, Tabellen oder Views zu löschen.
EXECUTE	Stored Procedures und Functions auszuführen.
GRANT OPTION	dass seine Privilegien von anderen Accounts verändert werden.
INDEX	Indexe zu erstellen.
INSERT	Daten einzufügen.
REFERENCES	Foreign Keys zu erzeugen
SELECT	Daten zu selektieren.
SHOW DATABASES	den Befehl “SHOW DATABASES” auszuführen.
SHOW VIEW	den Befehl “SHOW CREATE VIEW” auszuführen.
UPDATE	Daten zu ändern.

Sehen wir uns nun das „WO?“ an. Hier gibt es die Level „Komplettes RDBMS“ – also alle Datenbanken und Tabellen, das Level „bestimmte Datenbank“ und das Level „bestimmtes Element“ (also Tabelle, View oder Stored Procedure/Function):

Level	**Syntax:**
Komplettes RDBMS	*.*
Bestimmte Datenbank (bspw. „germac“)	germac.*
Bestimmtes Element (bspw. Tabelle „partner“ der Datenbank „germac“)	germac.partner

Die Befehle „SELECT“, „INSERT“ und „UPDATE“ können noch zusätzlich auf Spalten eingeschränkt werden, indem die Spalten in Klammern nach dem entsprechenden Privileg eingetragen werden. Wenn bspw. in der Tabelle „Partner“ der Datenbank „germac“ nur die „PartnerID“, der „Vorname“ und der „Nachname“ selektiert werden darf, sieht das Statement wie folgt aus:

```
GRANT SELECT(PartnerID, Vorname, Nachname)
```

```
      ON germac.partner TO 'User1'@'localhost';
```

Bei Views wird anstatt des Tabellennamens einfach der Viewname eingetragen. Für Funktionen und Prozeduren gilt allerdings ein Sonderfall. Hier muss nach dem „TO“ noch stehen, ob es sich um eine Prozedur:

```
GRANT EXECUTE ON FUNCTION germac.INSERT_SPACER2 TO 'User1'@'localhost';
```

oder eine Funktion handelt:

```
GRANT EXECUTE ON PROCEDURE germac.GetAllProducts TO 'User1'@'localhost';
```

Zum Schluss muss noch eingetragen werden, welchem User das Recht eingetragen wird. Dieser Syntax entspricht dem CREATE USER Statement.

8.3 Entfernen von Rechten

Rechte können den Usern auch wieder entzogen werden. Hierbei nutzen wir das „REVOKE“ Statement. Dies ist exakt wie das GRANT Statement aufgebaut und nimmt die entsprechenden Rechte wieder zurück. Wenn beispielsweise das Recht auf Lesen vom User1 wie im vorausgehenden Kapitel gezeigt entzogen werden soll, so ist folgendes einzugeben:

```
REVOKE SELECT(PartnerID, Vorname, Nachname)
      ON germac.partner FROM 'User1'@'localhost';
```

Wie erwähnt, sieht das Statement wie der GRANT Befehl aus, lediglich „GRANT“ wird durch „REVOKE“ und das „TO“ durch „FROM“ ersetzt.

Wer keinen Überblick mehr über die vergebenen Rechte hat, kann dies mit folgendem Befehl anzeigen lassen:

```
SHOW GRANTS for 'User1'@'localhost';
```

8.4 Entfernen von Usern

Wie zu erwarten ist, werden User mittels „DROP“ entfernt:

```
DROP USER 'User1'@'localhost';
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

9 Lizenz