Datenbanken in Sek II:
MySQL-Syntax

Inhalt:

1. Einleitung

SQL („Structured Query Language“) hat sich seit ihrer Entwicklung in den 70er-Jahren zum Standard für relationale Datenbanksysteme entwickelt. In der Syntax wird festgelegt, wie Anweisungen an Datenbanken strukturiert sein müssen.

Dabei werden folgende Teilbereiche unterschieden:

Bezeichnung	**Anweisungen (Auswahl)**	**Kapitel**
DCL (Data Control Language)	`CREATE DATABASE`, `DROP DATABASE`, `USE`	7, 3
DDL (Data Definition Language)	`CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`, `EXPLAIN`, `SHOW`	7, 3
DML (Data Manupulation Lang.)	`INSERT`, `DELETE`, `UPDATE`, `LOAD DATA`	6
DQL (Data Query Language)	`SELECT`	5

Die hier vorgestellte Syntax bezieht sich ausdrücklich auf MySQL (ab Version 5.0); Abweichungen zu anderen SQL-Dialekten sind möglich, spielen aber für die Anwendung in der Schule keine Rolle. Selbstverständlich wurde die Syntax auf ein für die Schule notwendiges Maß verkürzt; die Original-Dokumentation hat mehr als 1.600 Seiten.

2. Aufbau einer SQL-Anweisung

Jede SQL-Anweisung (mit Ausnahme von „`USE`“) muss mit einem Semikolon abgeschlossen werden.[1] Mehrere Anweisungen können zu einer SQL-Anweisung zusammengefasst werden, wobei (mit Ausnahme von „`USE`“) eine beliebige Aufteilung auf mehrere Zeilen zulässig ist.

2.1 Syntax-Darstellung

Für die Darstellung der SQL-Syntax werden im Folgenden der Übersichtlichkeit halber Schlüsselwörter in Großbuchstaben dargestellt; bei der Eingabe einer SQL-Anweisung ist aber jede Schreibweise aus Groß- und Kleinbuchstaben zulässig. Kursiv dargestellte Syntax-Elemente müssen durch konkrete Namen oder Werte ersetzt werden.

Die SQL-Syntax verwendet üblicherweise folgende Klammern-Darstellung :

`( )` Runde Klammern sind Syntax-Elemente und müssen mit eingegeben werden.

`[ ]` Eckige Klammern schließen optionale Syntax-Elemente ein, die weggelassen werden können. Die Klammern werden nicht eingegeben.

`{ }` Geschweifte Klammern fassen gleichwertige Syntax-Elemente zusammen, z. B. bei Alternativen. Die Klammern werden nicht eingegeben.

`|` Alternative Syntax-Elemente, von denen genau eines angewandt werden kann, werden durch einen senkrechten Strich getrennt, der nicht mit eingegeben wird.

`...` Drei Punkte kennzeichnen Wiederholungen von (gegebenenfalls auch umfangreicheren) Syntax-Elementen, die anstelle der Punkte eingesetzt werden müssen.

2.2 Kommentare

Kommentare innerhalb von SQL-Anweisungen werden mit „`/*`“ eingeleitet und durch „`*/`“ abgeschlossen.
Auf diesem Weg ist es möglich, Teile einer SQL-Anweisung zu Testzwecken auszukommentieren.

1 Einige Programme wie z. B. der MySQL Query Browser lassen bei einzelnen SQL-Anweisungen auch ein fehlendes Semikolon zu.

2.3 Bezeichner / reservierte Wörter

Die Namen von Datenbanken, Tabellen und Tabellenspalten sind in der Regel selbst gewählte Bezeichner, die aus all denen Zeichen bestehen dürfen, die auch für Verzeichnisse zugelassen sind. Wenn der Bezeichner ein reserviertes Wort (s. u.) ist oder Sonderzeichen (einschließlich des Leerzeichens) enthält, muss er in einfache Anführungszeichen „`“ gesetzt werden. Hierbei ist das Akzent-Zeichen (neben der Backspace-Taste) zusammen mit der Umschalt-Taste (und anschließend zu drückender Leertaste) zu verwenden.
Die Groß- und Kleinschreibung von Bezeichnern wird nur unter Linux bei Datenbank- und Tabellennamen unterschieden.

Die wichtigsten reservierten Wörter sind:

ADD	DECIMAL	IGNORE	NUMERIC	SONAME
ALL	DECLARE	IN	ON	SPATIAL
ALTER	DEFAULT	INDEX	OPTIMIZE	SPECIFIC
ANALYZE	DELAYED	INFILE	OPTION	SQL
AND	DELETE	INNER	OPTIONALLY	SSL
AS	DESC	INOUT	OR	STARTING
ASC	DESCRIBE	INSERT	ORDER	TABLE
BEFORE	DETERMINISTIC	INT	OUT	TERMINATED
BETWEEN	DISTINCT	INTEGER	OUTER	THEN
BIGINT	DISTINCTROW	INTERVAL	OUTFILE	TO
BINARY	DIV	INTO	PRECISION	TRAILING
BLOB	DOUBLE	IS	PRIMARY	TRIGGER
BOTH	DROP	ITERATE	PROCEDURE	TRUE
BY	DUAL	JOIN	PURGE	UNDO
CALL	EACH	KEY	READ	UNION
CASCADE	ELSE	KEYS	READS	UNIQUE
CASE	ELSEIF	KILL	REAL	UNLOCK
CHANGE	ENCLOSED	LEADING	REFERENCES	UNSIGNED
CHAR	ESCAPE	LEAVE	REGEXP	UPDATE
CHARACTER	EXISTS	LEFT	RENAME	USAGE
CHECK	EXIT	LIKE	REPEAT	USE
COLLATE	EXPLAIN	LIMIT	REPLACE	USING
COLUMN	FALSE	LINES	REQUIRE	VALUES
CONDITION	FETCH	LOAD	RESTRICT	VARBINARY
CONNECTION	FLOAT	LOCALTIME	RETURN	VARCHAR
CONSTRAINT	FOR	LOCK	RIGHT	VARCHARACTER
CONTINUE	FORCE	LONG	RLIKE	VARYING
CONVERT	FOREIGN	LOOP	SCHEMA	WHEN
CREATE	FROM	MATCH	SCHEMAS	WHERE
CROSS	FULLTEXT	MOD	SELECT	WHILE
CURSOR	GOTO	MODIFIES	SENSITIVE	WITH
DATABASE	GROUP	NATURAL	SEPARATOR	WRITE
DATABASES	HAVING	NOT	SET	XOR
DEC	IF	NULL	SHOW	ZEROFILL

3. Verwenden von Datenbanken

Die **SHOW**-Anweisung listet alle zur Verfügung stehenden Datenbanken auf:

```
SHOW DATABASES;
```

Mit der **USE**-Anweisung wird eine Datenbank zur aktuellen Datenbank erklärt. Alle folgenden Anweisungen beziehen sich dann auf diese Datenbank, wenn den Bezeichnern nicht explizit eine andere Datenbank vorangestellt wird. Die Syntax lautet:

```
USE Datenbank;
```

Die in einer Datenbank enthaltenen Tabellen erhält man wieder über die **SHOW**-Anweisung:

```
SHOW TABLES [FROM Datenbank];
```

Die in einer Tabelle enthaltenen Spalten und deren Datentypen kann man mithilfe folgender gleichwertiger Anweisungen anzeigen lassen:

```
{ DESC | DESCRIBE | EXPLAIN | SHOW COLUMNS FROM } Tabelle;
```

In Programmen wie dem MySQL Query Browser kann man diese Operationen häufig interaktiv mithilfe der Maus durchführen (siehe VLIN-Dokument „Installation und Verwendung von MySQL“).

4. Datentypen

Die folgende Tabelle listet die wichtigsten, in SQL verwendeten Datentypen auf:

Datentyp	Beschreibung
CHAR(*L*)	Zeichenkette fester Länge, die durch die Zahl „L“ bestimmt wird
TEXT	Zeichenkette variabler Länge mit bis zu 65535 Zeichen
DATE	Datum im Format „JJJJ-MM-TT“; Eingabe in einfachen Anführungszeichen: „'2006-06-12'“
YEAR	Jahreszahl zwischen 1901 und 2155
INT[(*S*)]	Ganzzahl; bei der Ausgabe werden „S“ Stellen angezeigt
BIGINT[(*S*)]	sehr große Ganzzahl; bei der Ausgabe werden „S“ Stellen angezeigt
FLOAT[(*S*,*N*)]	Fließkommazahl; bei der Ausgabe werden „S“ Stellen und „N“ Nachkommastellen angezeigt; Eingabe z. B. als „4.6E+4“ für 46.000
DOUBLE[(*S*,*N*)]	Fließkommazahl wie „**FLOAT**“, aber größerer Zahlenbereich und größere Genauigkeit
ENUM(*Liste*)	Datentyp, dessen Werte nur aus der „Liste“ stammen können, die aus in einfache Anführungszeichen gesetzten Zeichenketten besteht

Weitere Datentypen können der MySQL-Dokumentation entnommen werden.

Der Wert „**NULL**“ bedeutet, dass der zugehörige Wert nicht vorliegt.

5. Datenbank-Abfragen

Hauptanwendungsgebiet von SQL (vor allem in der Schule) ist das Abrufen von Daten aus Datenbanken. Die dafür verwendete **SELECT**-Anweisung wird in diesem Kapitel vorgestellt; die einzelnen Abschnitte des Kapitels behandeln Teilaspekte der **SELECT**-Anweisung.

5.1 SELECT-Syntax

Mithilfe der **SELECT**-Anweisung können Daten aus Datenbanken abgerufen werden. Die (bereits gekürzte) Syntax sieht folgendermaßen aus:

```
SELECT [DISTINCT] Select-Ausdruck [,...]
  FROM Tabellenreferenz
  [WHERE Bedingung]
  [GROUP BY { Spaltenname | Spaltennummer } [DESC] [, ...]
    [HAVING Bedingung] ]
  [ORDER BY { Spaltenname | Spaltennummer } [DESC] [, ...]]
  [LIMIT [Startzeile,] Zeilenanzahl];
```

Erläuterungen zu Syntax-Elementen, die nicht in den folgenden Abschnitten ausführlich aufgegriffen werden:

- Das Prädikat „**DISTINCT**“ sorgt dafür, dass identische Zeilen der Ausgabe nur einmal aufgelistet werden.
- Select-Ausdrücke legen fest, welche Spalten die Ausgabe haben soll. Neben „*“ für „alle Tabellenspalten“ und der direkten Auflistung von Tabellenspalten (ggf. mit Spalten-Alias; siehe Abschnitt 5.4) gibt es die Möglichkeit, Funktionen oder numerische Ausdrücke (siehe Abschnitte 5.5 bis 5.7) zu verwenden.
- Die Gruppierungsklausel „**GROUP BY**“ fasst die Tabellenspalten der Ausgabe zu Gruppen zusammen (siehe Abschnitt 5.5). Dies ist vor allem sinnvoll bzw. notwendig beim Einsatz der so genannten „Aggregatfunktionen“ (siehe Abschnitt 5.5). Sie schließt gleichzeitig eine Sortierung wie mit der Anweisung „**ORDER BY**“ mit ein. Der zusätzliche Einsatz von „**ORDER BY**“ überschreibt die Sortierung durch „**GROUP BY**“.
- Bei der Gruppierung und der Sortierung bestimmt die Reihenfolge der Parameter die Sortierreihenfolge. „Spaltennummer“ bezieht sich auf die Nummer der Spalte in der Ausgabe, wobei die erste Spalte die Nummer „1“ hat. Das Prädikat „**DESC**“ bewirkt eine absteigende Sortierung, wobei das Prädikat hinter jeden absteigend zu sortierenden Spaltennamen zu setzen ist. Sonderzeichen werden nach ihrem ASCII-Code sortiert.
- Im Gegensatz zur **WHERE**-Klausel bezieht sich **HAVING** nicht auf die Ursprungsdatensätze, sondern auf die für die Ausgabe gruppierten Datensätze, die jetzt eine Bedingung erfüllen müssen (siehe Abschnitt 5.5).
- Die **LIMIT**-Klausel listet nur die angegebene Anzahl von Zeilen auf. Wenn die optionale Startzeile fehlt, werden die ersten Zeilen aufgelistet. Die erste Zeile hat die Nummer „0“.

5.2 Tabellenreferenzen

Im einfachsten Fall kommen die Daten für die SQL-Abfrage nur aus einer Tabelle; dann muss auch nur diese Tabelle als Tabellenreferenz in der **SELECT**-Anweisung nach „**FROM**“ angegeben werden.

Sobald man Daten aus mehr als einer Tabelle verwenden möchte, müssen die Tabellen miteinander verknüpft werden. Die kürzeste Variante „*Tabelle1, Tabelle2*“ als Tabellenreferenz bewirkt, dass jeder Datensatz (also jede Zeile) der Tabelle 1 mit jedem Datensatz der Tabelle 2 verknüpft wird. Die Anzahl der Datensätze der entstandenen Verbund-Tabelle ist dann folglich das Produkt der beiden Anzahlen der verwendeten Tabellen.
In der Regel ist eine solche Verknüpfung nicht gewünscht, weil hierbei in der Verbund-Tabelle sinnlose Datensätze entstehen, z. B. „Bochum, Mongolei“ oder „New York, Kroatien“ etc., wenn z. B. die Tabellen „Stadt“ und „Land“ miteinander verknüpft werden. Die Verknüpfung ist meistens mit einer Bedingung (siehe Abschnitt 5.3) verbunden, bei der ein Attributwert der einen Tabelle mit einem Attributwert der anderen Tabelle übereinstimmen muss. Häufig sind dies Primär- und Fremdschlüssel der beteiligten Tabellen.

Im Wesentlichen gibt es drei verschiedene Möglichkeiten, diese bedingte Verknüpfung durchzuführen. Als Beispiel werden die Tabellen „Stadt“ und „Land“ verwendet, bei denen die Attributwerte der Attribute „Stadt.Landkuerzel“ und „Land.Kuerzel“ übereinstimmen müssen:

a) Die Tabellenreferenz ist eine Liste der Tabellen, die Verknüpfungsbedingung ist Bestandteil der **WHERE**-Klausel:

```
SELECT *
  FROM Stadt, Land
  WHERE Stadt.Landkuerzel = Land.Kuerzel;
```

Es wird also zuerst eine große Verbund-Tabelle erstellt, aus der dann nur die Datensätze selektiert werden, bei denen die Verknüpfungsbedingung erfüllt ist.
Es können auch mehr als zwei Tabellen aufgelistet werden. Es können auch weitere Bedingungen in der **WHERE**-Klausel mithilfe von „**AND**“ bzw. „**OR**“ (siehe Abschnitt 5.3) er-

gänzt werden, die eine Selektion von Datensätzen vornehmen, aber nicht zur Verknüpfungsbedingung gehören. Das Komma zwischen den Tabellennamen steht abkürzend für „**JOIN**“: „`Stadt` **`JOIN`** `Land`“.

b) Die Tabellenreferenz enthält selbst die Verknüpfungsbedingung:

```
SELECT *
  FROM Stadt INNER JOIN Land ON Stadt.Landkuerzel = Land.Kuerzel;
```

In diesem Fall werden also von vornherein nur die Datensätze zusammengefasst, die bezüglich der Verknüpfungsbedingung zusammengehören.
Selbstverständlich kann dann noch eine **WHERE**-Klausel folgen, die eine Selektion von Datensätzen der Verbund-Tabelle vornimmt.
Im Gegensatz zur Variante a) ist hier die syntaktische Trennung zwischen der *Verknüpfung von Tabellen* in der Tabellenreferenz und der *Selektion von Datensätzen aus dieser Verknüpfung* mithilfe der **WHERE**-Klausel deutlicher.
Wenn man mehr als eine Tabelle verknüpfen möchte, sieht die Syntax folgendermaßen aus: „*Tabelle1* **INNER JOIN** *Tabelle2* **INNER JOIN** *Tabelle3* **ON** *Bedingung*“, wobei die Bedingung in der Regel aus mehreren, mit „**AND**“ verknüpften Bedingungen besteht.
Der Zusatz „**INNER**“ kann weggelassen werden, er wurde nur der Übersichtlichkeit halber als Abgrenzung zur Variante c) mit aufgenommen.

c) Als Erweiterung der Variante b) ist Folgendes möglich:

```
SELECT *
  FROM Stadt LEFT OUTER JOIN Land ON Stadt.Landkuerzel = Land.Kuerzel;
```

Mit dieser Variante werden im Gegensatz zur Variante b) auch Datensätze der Tabelle „Stadt“ aufgelistet, die keinen zugehörigen Datensatz in der Tabelle „Land“ haben; die Attribute der Tabelle „Land“ erhalten dann für den Verbund-Datensatz in der Verbund-Tabelle den Wert „**NULL**“.
Im Gegensatz zu den Varianten a) und b) bewirkt eine Vertauschung der Tabellennamen eventuell eine Veränderung des Ergebnisses, weil dann *Länder* aufgelistet werden, die keinen zugehörigen Datensatz in der Tabelle „Stadt“ haben.
Eine Verknüpfung von mehr als einer Tabelle geschieht analog der Darstellung bei Variante b).
Der Zusatz „**OUTER**“ kann weggelassen werden, er wurde nur der Übersichtlichkeit halber als Abgrenzung zur Variante b) mit aufgenommen.

5.3 Bedingungen

Bedingungen in **WHERE**- bzw. **HAVING**-Klauseln, als Verknüpfungsbedingung oder in bedingten Funktionen (siehe Abschnitt 5.8) sind mit Bedingungen in Programmiersprachen vergleichbar. Es gibt jedoch einige Unterschiede:

- Mit „**AND**“, „**OR**“ etc. verknüpfte Bedingungen müssen im Gegensatz zu Delphi nicht notwendigerweise eingeklammert werden.
- Die Gleichheit wird im Gegensatz zu Java durch „=“ überprüft.
- Groß- und Kleinschreibung wird bei zu vergleichenden Zeichenketten in der Regel nicht unterschieden. Wenn dies gewünscht wird, muss einer der Zeichenketten „**BINARY**“ vorangestellt werden. Analoges gilt für abschließende Leerzeichen, die bei Vergleichen in der Regel ignoriert werden.
- Das Zutreffen einer Bedingung wird durch „1“ für „true“ bzw. „0“ für „false“ gekennzeichnet. Dementsprechend kann mit Bedingungen algebraisch gerechnet werden.

Folgende, aus den Programmiersprachen bekannte Vergleichsoperatoren stehen zur Verfügung: „=“ (gleich), „<>“ oder „!=“ (ungleich), „<“ (kleiner als), „>“ (größer als), „<=“ (kleiner oder gleich) und „>=“ (größer oder gleich).

Verbindungen aus Verknüpfungen sind mithilfe von „**AND**“ bzw. „`&&`“ oder „**OR**“ bzw. „`||`“ möglich.

Darüber hinaus gibt es Erweiterungen:

- „**BETWEEN**“ ermöglicht eine Abfrage, ob ein Ausdruck zwischen zwei Werten liegt, z. B. „*Ausdruck* **BETWEEN** *unterer_Wert* **AND** *oberer_Wert*“. Die beiden Randwerte sind dabei mit eingeschlossen. „**BETWEEN**“ kann mit jedem Datentyp verwendet werden.
- „**ISNULL**(*Ausdruck*)“ bzw. „*Ausdruck* **IS NULL**“ testet, ob der Ausdruck den Wert „**NULL**“ hat. Diese Bedingung kann man z. B. in einer **WHERE**-Klausel der Verknüpfungsvariante c) aus Abschnitt 5.2 verwenden.
- „**IN**“ prüft, ob ein Ausdruck in einer Liste enthalten ist: „*Ausdruck* **IN** *Liste*“, wobei die Liste von der Form „(*Wert1*,*Wert2*,...)“ ist.
  Bei Subquerys (siehe Abschnitt 5.9) wird „**IN**“ analog verwendet, wobei die Liste eine durch eine **SELECT**-Anweisung erzeugte Tabelle ist.
- „**LIKE**“ stellt Möglichkeiten für den Vergleich von Zeichenkettenmustern zur Verfügung: „*Ausdruck* **LIKE** *Muster*“[1]. Im „Muster“ steht das Zeichen „`%`“ für beliebig viele beliebige Zeichen (ggf. auch gar kein Zeichen), das Zeichen „_“ (Unterstrich) für genau ein beliebiges Zeichen. Das Muster „`_a%`“ findet also alle Zeichenketten, bei denen der zweite Buchstabe ein „a“ ist.
  Wenn man nach den Zeichen „%“ und „_“ suchen möchte, muss man diesen im Muster das Zeichen „`\`“ voranstellen: Das Muster „`%\%`“ findet also alle Zeichenketten, die mit einem Prozentzeichen enden.

Die Negation einer Bedingung ist über „**NOT**(*Bedingung*)“ bzw. „`!`(*Bedingung*)“ möglich. Häufig sind auch Kurzvarianten wie „**NOT BETWEEN**“, „**NOT IN**“, „**NOT LIKE**“, etc. möglich.

5.4 Alias-Definitionen

Sowohl in Select-Ausdrücken als auch in Tabellenreferenzen (siehe Abschnitt 5.2) sind Alias-Definitionen zulässig. Dies bedeutet, dass man für Spalten oder Tabellen selbst gewählte Namen vergibt, auf die an anderer Stelle innerhalb der **SELECT**-Anweisung Bezug genommen werden kann. Bei den Alias-Namen sind die Grundsätze für die Bezeichner (siehe Abschnitt 2.3) zu beachten.

Die Einsatzzwecke einer Alias-Definition hängen von der Position der Definition ab. Man unterscheidet zwischen Tabellen- und Spalten-Alias:

- Alias-Namen in Tabellenreferenzen („Tabellen-Alias“) sollen die **SELECT**-Anweisung verkürzen. Wenn auf zwei Tabellen Bezug genommen wird, die gleiche Spaltennamen haben, muss vor dem jeweiligen Spaltenname der Name der Tabelle angegeben werden (siehe Abschnitt 5.2). Beispiel für die Ausgabe aller Länder mit zugehöriger Hauptstadt:

  ```
  SELECT Stadt.Name, Land.Name
    FROM Stadt INNER JOIN Land ON Stadt.Nummer = Land.Hauptstadtnummer;
  ```

  Die Alias-Definition wird direkt hinter den Tabellennamen geschrieben. Sie *muss* jetzt überall in der **SELECT**-Anweisung anstelle des Tabellennamens stehen:

  ```
  SELECT s.Name, l.Name
    FROM Stadt s INNER JOIN Land l ON s.Nummer = l.Hauptstadtnummer;
  ```

  Die Verwendung von „**AS**“ ist optional möglich, aber unüblich: „`Stadt` **AS** `s`“.

  Wenn eine Subquery (siehe Abschnitt 5.9) als Tabellenreferenz verwendet wird, *muss* sie mit einem Tabellen-Alias versehen werden.

1 Je nach Installation des MySQL-Servers bietet es sich an, anstelle von „*Ausdruck*“ den Term „*Ausdruck* **COLLATE** `latin1_general_ci`“ zu verwenden, weil ansonsten ein schwedischer Zeichensatz zugrunde gelegt wird, bei dem z. B. „ü“ und „y“ gleichbedeutend sind. Wenn man also mithilfe des Musters „`%ü%`“ alle Namen mit „ü“ anzeigen lassen möchte, bekommt man auch alle mit „y“ angezeigt und umgekehrt. Dies gilt nicht für die Funktionen zur Verarbeitung von Zeichenketten (siehe Abschnitt 5.7).

- Alias-Namen in Select-Ausdrücken („Spalten-Alias“) haben zweierlei Funktionen:
  Zum einen werden Sie als Überschriften der Ausgabespalten verwendet. Es lassen sich auf diesem Weg aussagekräftige Spaltenköpfe erzeugen:

```
SELECT Stadt.Name AS `Name(Stadt)`, Land.Name AS `Name(Land)`
  FROM Stadt INNER JOIN Land ON Stadt.Landkuerzel = Land.Kuerzel;
```

  Zum anderen können diese Alias-Namen in den Klauseln „**GROUP BY**“ (siehe Abschnitt 5.5) bzw. „**ORDER BY**“ verwendet werden, so dass die Lesbarkeit der **SELECT**-Anweisung erhöht wird:

```
SELECT Stadt.Name, Stadt.Einwohner,
    Stadt.Einwohner / Land.Einwohner AS `Anteil an Gesamtbevölkerung`
  FROM Stadt INNER JOIN Land ON Stadt.Landkuerzel = Land.Kuerzel
  ORDER BY `Anteil an Gesamtbevölkerung`;
```

  Dieses Beispiel zeigt darüber hinaus, dass es mithilfe der Alias-Namen einfacher möglich ist, auf Spalten mit Berechnungen (siehe Abschnitte 5.5 bis 5.7) Bezug zu nehmen.

  Die Verwendung eines Spalten-Alias in **WHERE**-Klauseln ist nicht möglich. Die Verwendung eines Spalten-Alias mit Anführungszeichen ist in **HAVING**-Klauseln nicht möglich.
  Das „**AS**“ ist optional, kann also weggelassen werden. Der zweite von zwei Begriffen wird dann als Spalten-Alias interpretiert. Wenn zwischen zwei Select-Ausdrücken ein Komma vergessen wird, kann dies folglich zu unerwünschten Ausgabeergebnissen führen!

5.5 Aggregat-Funktionen und Gruppierungen

Aggregat-Funktionen dienen dazu, einen Wert zu berechnen, der auf mehreren Datensätzen beruht, wie z. B. den Durchschnitt von Werten.

Es gibt folgende Aggregat-Funktionen:

Aggregat-Funktion	Bedeutung
COUNT(*Spaltenname*)	ermittelt die Anzahl der Einträge in der Spalte
SUM(*Spaltenname*)	berechnet die Summe der Spaltenwerte
AVG(*Spaltenname*)	berechnet den Durchschnitt der Spaltenwerte
MIN(*Spaltenname*)	bestimmt das Minimum der Spaltenwerte
MAX(*Spaltenname*)	bestimmt das Maximum der Spaltenwerte

Anmerkungen:
- Zwischen dem Funktionsnamen und der öffnenden Klammer darf kein Leerzeichen stehen!
- Es werden nur die Spaltenwerte berücksichtigt, die nicht den Wert „**NULL**“ haben; dies hat vor allem Auswirkung auf das Ergebnis von „**COUNT**“ und „**AVG**“. Wenn man *alle* Datensätze zählen möchte, verwendet man „**COUNT**(*)“.
- In allen Aggregat-Funktionen darf vor dem Spaltennamen ein „**DISTINCT**“ stehen, so dass doppelte Einträge nicht berücksichtigt werden.

Eingesetzt werden Aggregat-Funktionen vor allem in Select-Ausdrücken: Die SQL-Abfrage „**SELECT SUM**(Einwohner) **FROM** Land;“ ermittelt z. B. die Anzahl aller Einwohner aller Länder. Häufig bietet es sich an, einen Spalten-Alias zu verwenden (siehe Abschnitt 5.4).

Vor der Ausführung der Aggregat-Funktion kann noch eine Selektion der einzubeziehenden Datensätze mithilfe einer **WHERE**-Klausel durchgeführt werden. Wenn man z. B. die Anzahl der Einwohner Afrikas ermitteln will, verwendet man folgende SQL-Abfrage: „**SELECT SUM**(Einwohner) **FROM** Land **WHERE** Kontinent = 'Africa';“.
Dieser Sachverhalt macht auch deutlich, warum Aggregat-Funktionen in **WHERE**-Klauseln nicht verwendet werden dürfen: Die Bedingung in einer **WHERE**-Klausel bezieht sich immer auf einen einzelnen Datensatz, während die Aggregat-Funktion mehrere Datensätze einbezieht.

Eine gleichzeitige Verwendung von Spaltennamen und Aggregat-Funktionen als Select-Ausdruck in einer **SELECT**-Anweisung ist nicht ohne weiteres möglich. Betrachten wir folgendes Beispiel:

```
SELECT Kontinent, AVG(Lebenserwartung)
  FROM Land;
```

Mit dieser **SELECT**-Anweisung sollte die Lebenserwartung der jeweiligen Kontinente berechnet werden. Die Aggregat-Funktion „**AVG**“ berechnet jedoch den Durchschnitt der Lebenserwartungen der *gesamten* Tabelle „Land“, ohne auf den jeweiligen Kontinent Rücksicht zu nehmen. Deswegen kann auch nicht die Bezeichnung eines Kontinents ausgegeben werden, weil keine der Bezeichnungen zu dem berechneten Durchschnittswert passt.
In diesem Fall muss MySQL angewiesen werden, die Tabelle „Land“ mithilfe von „**GROUP BY**“ in Gruppen der jeweiligen Kontinente aufzuteilen und von diesen Gruppen den jeweiligen Durchschnitt zu berechnen:

```
SELECT Kontinent, AVG(Lebenserwartung) AS `Durchschnitt LE`
  FROM Land
  GROUP BY Kontinent
  ORDER BY `Durchschnitt LE` DESC;
```

In diesem Beispiel wurde zusätzlich ein Spalten-Alias eingesetzt (siehe Abschnitt 5.4) sowie eine absteigende Sortierung nach der durchschnittlichen Lebenserwartung durchgeführt.

Wenn man nur bestimmte Ergebnisse der Aggregat-Funktionen mit Gruppierung anzeigen lassen möchte, darf man diese Bedingung nicht in die **WHERE**-Klausel setzen, weil die Bedingung nicht den einzelnen Datensatz, sondern das Ergebnis der gesamten Gruppe betrifft. In diesem Fall ist die **HAVING**-Klausel zu verwenden:

```
SELECT Kontinent, AVG(Lebenserwartung) AS `Durchschnitt LE`
  FROM Land
  GROUP BY Kontinent HAVING AVG(Lebenserwartung) > 70
  ORDER BY `Durchschnitt LE` DESC;
```

Es werden also nur die Kontinente angezeigt, bei denen die durchschnittliche Lebenserwartung größer als 70 Jahre ist.
Dieses Beispiel zeigt darüber hinaus, wie vorzugehen ist, wenn der Spalten-Alias Anführungszeichen enthält, so dass ein direkter Einsatz in der **HAVING**-Klausel nicht möglich ist (siehe Abschnitt 5.4).

5.6 Rechenoperationen

Folgende mathematische Rechenoperationen bzw. Funktionen stehen zur Verfügung, wobei die Operanden bzw. Parameter Bezeichner von Tabellenspalten sein können:

- Grundrechenarten „`+`“, „`-`“, „`*`“ und „`/`“ sowie die Integer-Divisionen „**MOD**“ bzw. „`%`“ und „**DIV**“. Beispiel: „`Einwohner / 1000 AS `Bevölkerung in Tausend`“
- Rundungsfunktionen: „**ROUND**(*Zahl*)“ sowie „**CEIL**(*Zahl*)“ und „**FLOOR**(*Zahl*)“ für Auf- und Abrundung.
  Möchte man eine bestimmte Anzahl von Nachkommastellen erhalten, verwendet man „**ROUND**(*Zahl*,*Nachkommastellen*)“ bzw. „**TRUNCATE**(*Zahl*,*Nachkommastellen*)“. Für den Parameter „Nachkommastellen“ sind auch negative Zahlen zugelassen, so dass eine Rundung auf Zehner etc. erfolgt: „**ROUND**(`137.19,-1`)“ ergibt den Wert „140“.
  Bei der Verwendung von Aggregat-Funktionen (insbesondere „**SUM**“; siehe Abschnitt 5.5), die auf Spalten mit Dezimalzahlen angewandt werden, sollte anschließend noch eine Rundung vorgenommen werden, weil es aufgrund der Speicherung der Zahlen im Dualzahl-Format zu Abweichungen zum algebraischen Dezimalzahl-Ergebnis kommen kann.
- Mathematische Funktionen:
  „**ABS**(*Zahl*)“ ergibt den Betrag einer Zahl, „**SQRT**(*Zahl*)“ berechnet die Wurzel einer Zahl, „**PI**()“ ergibt den Wert der Kreiszahl π.

Potenz: „**POWER**(2,3)“ liefert den Wert 8.
Exponential- und Logarithmusfunktionen: „**EXP**(*Zahl*)“, „**LN**(*Zahl*)“, „**LOG2**(*Zahl*)“, „**LOG10**(*Zahl*)“
Trigonometrische Funktionen: „**COS**(*Winkel*)“, „**SIN**(*Winkel*)“, „**TAN**(*Winkel*)“, „**COT**(*Winkel*)“, „**ACOS**(*Winkel*)“, „**ASIN**(*Winkel*)“ und „**ATAN**(*Winkel*)“, bei denen „Winkel“ im Bogenmaß anzugeben ist.

- „**RAND**()“: Zufallszahl zwischen 0 und 1.
  Diese Funktion kann auch eingesetzt werden, um eine bestimmte Anzahl von zufällig ausgewählten Datensätzen anzuzeigen: „`... ORDER BY RAND() LIMIT 5;`“

Wie bei den Aggregat-Funktionen darf zwischen Funktionsnamen und öffnender Klammer kein Leerzeichen stehen. Die Funktionen dürfen in allen Teilen einer **SELECT**-Anweisung verwendet werden.
Damit die Berechnungsformel in der Ausgabe nicht als Spaltenkopf erscheint, bietet es sich an, wie im ersten Beispiel einen Spalten-Alias zu verwenden (siehe Abschnitt 5.4).

Das Format der *ausgegebenen* Zahl, die aber intern mit maximaler Genauigkeit berechnet wird, folgt nachstehenden Kriterien:

- Die Funktion „**PI**()“ liefert 6 Nachkommastellen.
- Bei Dezimalzahlen werden maximal 14 signifikante Ziffern angezeigt. Dies betrifft insbesondere die mathematischen Funktionen mit Ausnahme von „**ABS**“, „**PI**()“ und „**POWER**“
- Die Funktion „**ABS**“ sowie die Aggregat-Funktionen „**MIN**“, „**MAX**“ und „**SUM**“ geben den Datentyp des übergebenen Parameters bzw. der übergebenen Tabellenspalte zurück.
- Die Grundrechenarten (mit Ausnahme von „/“) mit zwei Zahlen geben den Datentyp der genaueren Zahl zurück. Für den Fall, dass beide Zahlen eine Ganzzahl sind, ist dies eine Ganzzahl.
- Bei der Division mit „/“ werden vier Nachkommastellen mehr ausgegeben als die geringere Anzahl der Nachkommastellen der beiden Zahlen. Dies bewirkt auch, dass die Aggregat-Funktion „**AVG**“ vier Nachkommastellen mehr zurückgibt als der Datentyp der übergebenen Tabellenspalte.

Eine Änderung dieses Ausgabeformats ist ggf. über die Verwendung einer Rundungsfunktion oder durch die Multiplikation mit der Dezimalzahl „1.0“ mit der erforderlichen Anzahl von Nullen als Nachkommastellen möglich: „**PI**()`*1.00000000`“.

5.7 Zeichenkettenoperationen

Für die Verarbeitung von Zeichenketten – im Folgenden „*ZK*“ abgekürzt – stehen u. a. folgende Funktionen zur Verfügung:

- „**LENGTH**(*ZK*)“ ergibt die Länge von ZK.
- „**ASCII**(*ZK*)“ ergibt den ASCII-Code des ersten Zeichens von ZK. Die Umwandlung eines ASCII-Codes in eine ZK geschieht mithilfe von „**CHAR**(*Zahl1*,*Zahl2*,...)“, wobei so viele Zeichen ausgegeben werden wie Zahlen übergeben wurden.
- „**LOWER**(*ZK*)“ bzw. „**UPPER**(*ZK*)“ wandelt alle Buchstaben in Klein- bzw. Großbuchstaben um.
- „**REVERSE**(*ZK*)“ ergibt ZK in umgekehrter Zeichenfolge.
- „**CONCAT**(*ZK1*,*ZK2*,...)“ ergibt eine ZK, die aus den übergebenen ZK zusammengefügt ist.
- „**LOCATE**(*Teil-ZK*,*ZK*)“ ergibt die Position des ersten Auftretens der Teil-ZK in der ZK. Wenn die Teil-ZK nicht vorhanden ist, lautet das Ergebnis „0“.
  Wenn man erst ab einer bestimmten Position innerhalb von ZK suchen lassen möchte, verwendet man „**LOCATE**(*Teil-ZK*,*ZK*,*Position*)“.
  Diese Funktion berücksichtigt im Gegensatz zu „**REPLACE**“ (s. u.) keine Groß- und Kleinschreibung. Ggf. muss vor einer der ZK „**BINARY**“ stehen.

- „**SUBSTRING**(*ZK*,*Position*,*Länge*)“ bzw. „**SUBSTRING**(*ZK*,*Position*)“ ergibt eine Teil-ZK aus der übergebenen ZK ab der angegebenen Position in der angegebenen Länge bzw. in der zweiten Variante bis zum Ende. Alternativ kann man die Funktionen „**LEFT**(*ZK*,*Länge*)“ bzw. „**RIGHT**(*ZK*,*Länge*)“ verwenden, um eine Teil-ZK der angegebenen Länge vom linken bzw. rechten Rand der ZK zu erhalten.
- „**REPLACE**(*ZK*,*ZK_alt*,*ZK_neu*)“ ergibt die ZK, wobei aber alle Vorkommen der alten ZK durch die neue ZK ersetzt sind.
  Diese Funktion berücksichtigt im Gegensatz zu „**LOCATE**“ (s. o.) Groß- und Kleinschreibung.
- „**INSERT**(*ZK*,*Position*,*Länge*,*ZK_neu*)“ ergibt die ZK, wobei ab der angegebenen Position die Teil-ZK aus der übergebenen ZK mit der angegebenen Länge durch die neue ZK ersetzt ist, deren Länge abweichen kann.
- „**TRIM**(*ZK_weg* **FROM** *ZK*)“ ergibt die ZK, bei der alle Auftreten der wegzunehmenden ZK am Anfang und am Ende entfernt wurden. Wenn man nur am Anfang oder nur am Ende „kürzen“ möchte, muss vor der wegzunehmenden ZK noch „**LEADING**“ bzw. „**TRAILING**“ stehen. Die Kurzform „**TRIM**(*ZK*)“ entfernt alle Leerzeichen am Rand.

Wie bei den Aggregat-Funktionen darf zwischen Funktionsnamen und öffnender Klammer kein Leerzeichen stehen. Die Funktionen dürfen in allen Teilen einer **SELECT**-Anweisung verwendet werden.
Zu beachten ist bei allen Funktionen, dass sie nicht den Datenbestand verändern, sondern nur einen Wert zurückgeben, auf den weiter Bezug genommen werden kann. So ist es z. B. möglich, mithilfe von „**GROUP BY LEFT**(*ZK*,1)“ nach dem Anfangsbuchstaben zu gruppieren.

Die Umwandlung von Zeichenketten in Zahlen geschieht automatisch. MySQL akzeptiert also Ausdrücke wie „`'17'*'4'`“.

5.8 Bedingte Funktionen

Es gibt zwei Funktionen, deren Ergebnis von einer Bedingung bzw. von bestimmten Werten abhängig ist:

- „**IF**(*Bedingung*,*Dann-Wert*,*Sonst-Wert*)“ liefert den „Dann-Wert“ zurück, wenn die Bedingung (siehe Abschnitt 5.3) erfüllt ist, ansonsten den „Sonst-Wert“.
- Für eine verschachtelte bedingte Funktion gibt es die **CASE**-Funktion:

```
CASE Wert
  WHEN Vergleichswert THEN Ergebnis
  [WHEN Vergleichswert2 THEN Ergebnis2 ...]
  [ELSE Ergebnis_sonst]
END
```

  Alternativ:

```
CASE WHEN Bedingung THEN Ergebnis
  [WHEN Bedingung2 THEN Ergebnis2 ...]
  [ELSE Ergebnis_sonst]
END
```

  Die erste Variante vergleicht einen Wert mit aufgelisteten Vergleichswerten und gibt den zugehörigen Ergebniswert zurück. Bei der zweiten Variante wird der Ergebniswert zurückgegeben, bei dem als erstem die zugehörige Bedingung wahr ist.
  Wenn kein **ELSE**-Zweig existiert, wird ggf. „**NULL**“ zurückgegeben.

In beiden Fällen ist zu beachten, dass es Funktionen sind, die einen Wert zurückgeben, auf den weiter Bezug genommen werden kann. Beide Funktionen dürfen in allen Teilen einer **SELECT**-Anweisung verwendet werden.

5.9 Subquerys

Bestimmte Datenbank-Abfragen sind ohne so genannte Subquerys nicht möglich. Dazu folgendes Beispiel:

Es sollen alle Städte aufgelistet werden, die mindestens ein Drittel der Einwohner der bevölkerungsreichsten Stadt der Welt haben. Eine nicht funktionierende Variante wäre:

```
SELECT Name, Einwohner
  FROM Stadt
  WHERE Einwohner >= MAX(Einwohner) / 3;
```

Der Grund liegt darin, dass in der **WHERE**-Klausel keine Aggregat-Funktionen verwendet werden dürfen (siehe Abschnitt 5.5).

„Von Hand“ würde man das Problem so lösen: Zuerst lässt man sich das Maximum der Einwohnerzahl der Städte anzeigen, notiert diese Zahl und ersetzt in der falschen **SELECT**-Anweisung den Eintrag „**MAX**(Einwohner)“ durch diese Zahl. Diesen Vorgang kann man auch automatisieren:

```
SELECT Name, Einwohner
  FROM Stadt
  WHERE Einwohner >= (SELECT MAX(Einwohner) FROM Stadt) / 3;
```

In den Klammern steht jetzt eine eigene **SELECT**-Anweisung (allerdings ohne das abschließende Semikolon), die nur einen einzigen Zahlenwert liefert, nämlich gerade die Einwohnerzahl der bevölkerungsreichsten Stadt der Welt. Mit diesem Zahlenwert wird jetzt verglichen, und das gewünschte Ergebnis wird angezeigt. Eine solche eingeschachtelte **SELECT**-Anweisung heißt „Subquery“ (zu Deutsch: Unterabfrage). Für eine Subquery gilt die gleiche Syntax wie für eine (normale) **SELECT**-Anweisung, nur dass keine **LIMIT**-Klausel verwendet werden darf.

Eine weitere Verwendungsmöglichkeit ist der Einsatz von „**IN**“ vor einer Subquery. Hierbei gibt die Subquery nicht einen einzelnen Wert, sondern eine gesamte Tabellenspalte zurück, in der der betreffende Ausdruck vorhanden sein muss. Dazu folgendes Beispiel:

Es sollen alle Städte mit ihren Daten aufgelistet werden, die in den Ländern liegen, die mindestens 50 Millionen Einwohner haben; die Daten der zugehörigen Länder sollen nicht mit aufgelistet werden:

```
SELECT *
  FROM Stadt
  WHERE Landkuerzel IN
   (SELECT Kuerzel
    FROM Land
    WHERE Einwohner > 50000000);
```

Das Beispiel ist zugegebenermaßen etwas gekünstelt, weil das Ergebnis mit dem der folgenden Abfrage identisch ist:

```
SELECT Stadt.*
  FROM Stadt INNER JOIN Land ON Landkuerzel = Kuerzel
  WHERE Land.Einwohner > 50000000;
```

Die dritte wesentliche Verwendungsmöglichkeit von Subquerys ist der Einsatz einer Subquery als Tabellenreferenz nach „**FROM**“. Dies bedeutet, dass die Subquery nicht nur einen Zahlenwert oder eine Tabellenspalte zurückgibt, sondern eine gesamte Tabelle, die dann als Tabellenreferenz der äußeren **SELECT**-Anweisung verwendet wird. Die Subquery muss in diesem Fall mit einem Tabellen-Alias versehen werden (siehe Abschnitt 5.4), damit man sich auf sie beziehen kann. Dazu folgendes Beispiel:

Es soll eine Liste der Städte erstellt werden, die mehr Einwohner haben als die Hauptstadt des betreffenden Landes:

```
SELECT SQ.Land, SQ.Hauptstadt, SQ.Einwohner, Stadt.Name AS Stadt,
       Stadt.Einwohner
  FROM Stadt INNER JOIN
       (SELECT Land.Name AS Land, Stadt.Name AS Hauptstadt, Land.Kuerzel,
               Stadt.Einwohner
          FROM Stadt INNER JOIN Land ON Nummer = Hauptstadtnummer) AS SQ
       ON Stadt.Landkuerzel = SQ.Kuerzel
  WHERE Stadt.Einwohner > SQ.Einwohner
  ORDER BY SQ.Land, Stadt.Einwohner DESC;
```

Die Subquery erstellt eine Tabelle „SQ“ mit den Spaltenköpfen „Land“, „Hauptstadt“, „Kuerzel“ und „Einwohner“, die alle Länder mit ihren Hauptstädten und deren Einwohnerzahl enthält.
Die äußere **SELECT**-Anweisung verknüpft diese Tabelle mit der Tabelle „Stadt“, wobei die Länder-Kürzel übereinstimmen müssen. Es werden nur die Datensätze der Verbund-Tabelle angezeigt, bei denen die Einwohnerzahl der Hauptstadt überschritten ist.

Etwas merkwürdig, weil nirgendwo dokumentiert, ist folgender Fall:[1]

```
SELECT Name
  FROM Land
  WHERE (SELECT SUM(Einwohner)
           FROM Stadt
           WHERE Stadt.Landkuerzel = Land.Kuerzel) > 10000000;
```

Die Subquery enthält einen Tabellenverweis, der außerhalb der Subquery deklariert ist; das bedeutet, die Subquery allein kann gar nicht ausgeführt werden. Trotzdem funktioniert die gesamte **SELECT**-Anweisung und liefert die Namen aller Länder, bei denen die Summe der Einwohner der verzeichneten Städte größer als 10 Millionen ist.

6. Verändern des Datenbestandes einer Datenbank

Die bisherigen Ausführungen beschränkten sich darauf, vorhandene Datenbanken zu verwenden, ohne den Datenbestand selbst zu verändern. Dieses kann durch Veränderung einzelner Daten oder das Löschen bzw. Einfügen von ganzen Datensätzen geschehen.

6.1 Veränderung von Werten in einer Datenbank

Wenn einzelne Werte in einer Datenbank verändert werden sollen, wird dafür die **UPDATE**-Anweisung verwendet. Deren Syntax lautet:

```
UPDATE Tabellenreferenz
  SET Spalte1 = Wert1 [, Spalte2 = Wert2] [,...]
  [WHERE Bedingung];
```

Dabei werden mit „**SET**“ die Spaltennamen und die neuen Datenwerte festgelegt. Mithilfe der **WHERE**-Klausel können die zu verändernden Datensätze ausgewählt werden. Die Tabellenreferenz (siehe Abschnitt 5.2) ist in der Regel nur eine einzelne Tabelle.

Wenn beispielsweise die Stadt „London“ in Kanada umbenannt werden soll, muss folgende Anweisung zum Einsatz kommen:

```
UPDATE Stadt
  SET Name = 'New London'
  WHERE Name = 'London' AND Landkuerzel = 'CAN';
```

Bei der **SET**-Option sind auch Selbstbezüge möglich. In folgendem Beispiel werden die Bevölkerungszahlen aller deutschen Städte um 1000 nach oben verändert:

```
UPDATE Stadt
  SET Einwohner = Einwohner + 1000
  WHERE Landkuerzel = 'DEU';
```

1 Mein Dank für dieses Beispiel geht an zwei mutige Schüler, die sich nicht an mein Skript gehalten hatten und einfach etwas ausprobiert hatten.

Da es in MySQL keine Undo-Funktion gibt, sollte vorher anhand einer **SELECT**-Anweisung die **WHERE**-Klausel dahingehend überprüft werden, ob sie genau die zu verändernden Datensätze liefert.

6.2 Löschen von Datensätzen

Das Löschen von Datensätzen geschieht mithilfe der **DELETE**-Anweisung, die folgende Syntax hat:

```
DELETE
  FROM Tabelle
  [WHERE Bedingung];
```

Wie schon in Abschnitt 6.1 gesagt, sollte vorher anhand einer **SELECT**-Anweisung die **WHERE**-Klausel dahingehend überprüft werden, ob sie genau die zu löschenden Datensätze liefert.

6.3 Einfügen von neuen Datensätzen

Das Einfügen von (neuen) Datensätzen kann auf mehrere Weisen geschehen. Das direkte Einfügen ist mithilfe der **INSERT**-Anweisung möglich:

```
INSERT Tabelle [ (Spalte1 [, Spalte2] ...) ]
  VALUES ( Wert1 [, Wert2] ... ) [, (...)] ...;
```

Auf die Auflistung der Spalten kann verzichtet werden, wenn für *jede* Spalte der angegebenen Tabelle ein neuer Wert aufgelistet wird. In diesem Fall muss die Reihenfolge allerdings mit der Reihenfolge in der Datendefinition übereinstimmen. Da in Spalten mit Auto-Increment keine Werte eingetragen werden sollten, ist die Auflistung der zu ergänzenden Spalten zu empfehlen. Beispiel:

```
INSERT Stadt (Name, Landkuerzel, Gebiet, Einwohner)
  VALUES ('Hallig Hooge', 'DEU', 'Schleswig-Holstein', 27);
```

Die **VALUES**-Option lässt die Eingabe *mehrerer* Datensätze in einem Schritt mithilfe einer **INSERT**-Anweisung zu; die jeweiligen Datensätze werden von runden Klammern eingeschlossen.

Eine weitere Möglichkeit ist das Einlesen einer Text-Datei, in der die Datensätze durch einen Zeilenumbruch und die Datenfelder innerhalb eines Datensatzes durch Tabulatoren getrennt sind. Die zugehörige Syntax sieht folgendermaßen aus:

```
LOAD DATA INFILE Datei
  INTO TABLE Tabelle
  [FIELDS TERMINATED BY Zeichenkette];
```

Der Dateiname muss dabei in Anführungszeichen gesetzt sein; optional können andere Trennzeichen für die Trennung der Datenfelder angegeben werden (z. B. bei einer CSV-Datei).

Des Weiteren gibt es die Möglichkeit, so genannte Skriptdateien (Dateien mit der Endung „.sql“) einzulesen, die **INSERT**-Anweisungen zum Einfügen von Datensätzen enthalten. Da der Umgang mit Skriptdateien vom jeweils verwendeten Werkzeug abhängig ist, soll hier nicht weiter darauf eingegangen werden.[1]

1 Für das Programm MySQL Query Browser findet man entsprechende Informationen im VLIN-Dokument „Installation und Verwendung von MySQL“.

7. Verändern der Struktur einer Datenbank

Die Abschnitte dieses Kapitels behandeln das Erstellen und Löschen von Datenbank- bzw. Tabellen-Strukturen sowie die Veränderung von Tabellen-Strukturen.
In Programmen wie dem MySQL Query Browser kann man diese Operationen häufig interaktiv mithilfe der Maus durchführen (siehe VLIN-Dokument „Installation und Verwendung von MySQL“).

7.1 Erstellen bzw. Löschen einer Datenbank

Eine neue Datenbank wird mit dem Befehl „**CREATE DATABASE** *Datenbankname;*“ erzeugt. (Anschließend muss „**USE**“ verwendet werden; siehe Kapitel 3).
Das Löschen einer Datenbank ist mithilfe von „**DROP DATABASE** *Datenbankname;*“ möglich.

7.2 Erstellen bzw. Löschen einer Tabellendefinition

Eine neue Tabelle innerhalb dieser Datenbank wird dann mit folgender Anweisung erzeugt:

```
CREATE TABLE Tabellenname (
  Spaltenname Datentyp [Optionen] [, ...],
  PRIMARY KEY (Spaltenname [,...])
  );
```

Als „Optionen“ sind einer oder mehrere der Einträge „**DEFAULT** *Standardwert*“, „**NOT NULL**“ oder „**AUTO_INCREMENT**“ zugelassen.

Die Verwendung macht man sich am besten an einem Beispiel aus der Datenbank „Welt“ deutlich:

```
CREATE TABLE 'Stadt' (
  'Nummer'      INT(11)  NOT NULL AUTO_INCREMENT,
  'Name'        CHAR(35) NOT NULL DEFAULT '',
  'Landkuerzel' CHAR(3)  NOT NULL DEFAULT '',
  'Gebiet'      CHAR(25) NOT NULL DEFAULT '',
  'Einwohner'   INT(11)  NOT NULL DEFAULT '0',
  PRIMARY KEY ('Nummer')
  );
```

Das Löschen einer Tabelle ist mithilfe von „**DROP TABLE** *Tabellenname;*“ möglich.

7.3 Änderung einer Tabellendefinition

Änderungen an bestehenden Tabellendefinitionen können nachstehende Vorgänge betreffen. Die gemeinsame Syntax lautet: „**ALTER TABLE** *Tabellenname Aktion;*“

Die „Aktion“ kann dabei eine der folgenden sein, wobei die Spaltendefinition wie bei der **CREATE-TABLE**-Anweisung aussieht:

- „**DROP** *Spalte*“ (Spalte löschen)
- „**ADD** *Spaltendefinition*“ (Spalte hinzufügen)
- „**CHANGE** *Spaltenname_alt Spaltenname_neu Spaltendefinition*“ (Neu-Definition einer Spalte und ggf. Umbenennung der Spalte)
- „**MODIFY** *Spaltenname Spaltendefinition*“ (Neu-Definition einer Spalte ohne Umbenennung der Spalte)

Die Änderung des Tabellennamens ist über „**RENAME TABLE** *Tabellenname_alt* **TO** *Tabellenname_neu;*“ möglich.