Landeshauptstadt
München
**Referat für
Bildung und Sport**

Daten im Griff

mit LibreOffice Base, MySQL & SQL

**Städtische Berufsschule
für Informationstechnik**

Herausgeber Michael Niedermair
Städtische Berufsschule für Informationstechnik
http://www.bsinfo.musin.de/

Autoren Michael Niedermair, Martina Pomper

Quellen, Material, Unterstützung, ...
- Michael Niedermair
- Martina Pomper
- Handbücher LibreOffice
  https://de.libreoffice.org/hilfe-kontakt/handbuecher/
  Base Handbuch (Datenbank) Version 4
  Robert Großkopf, Jost Lange, Michael Niedermair,
  Jochen Schiffers, Jürgen Thomas
- MySQL 5.1 Referenzhandbuch (Deutsch)
- MySQL 5.6 Reference Manual (Englisch)
- Elke Niedermair

Satz Michael Niedermair

Version 2013-06-13 (Revision: 1114) (BETA-Entwurf)

Lizenz http://creativecommons.org/licenses/by-nc-nd/3.0/de/

Inhaltsverzeichnis

1 Vorwort 5

2 Einführung 7

3 Datenbanken 9
3.1 LibreOffice Base 10
3.2 MySQL 12
3.3 SQL - Grundlagen 16
3.4 SQL - Datenbanken erstellen 18

4 Tabellen 21
4.1 SQL - Tabellen anlegen 22
4.2 DFB Tabellen mit SQL erstellen (Stufe 1) 26
4.3 Tabellenentwurf mit MySQL-Workbench 27
4.4 DFB Tabellen mit LibreOffice Base erstellen (Stufe 1) 32
4.5 Verknüpfung von Tabellen - Beziehungen 34
4.6 SQL - Eingabe von Daten 36
4.7 SQL - Ändern von Daten 38
4.8 Eingabe von Daten in Tabellen mit LibreOffice Base 40

5 Abfragen 57
5.1 SQL - Abfrage von Daten 58
5.2 Abfrageerstellung mit der grafischen Benutzeroberfläche 94

6 Berichte mit dem Report-Designer 99
6.1 Die Benutzeroberfläche des Report-Designers 100
6.2 Allgemeine Eigenschaften von Feldern 105
6.3 Besondere Eigenschaften des grafischen Kontrollfeldes 105
6.4 Diagramme im Bericht einbinden 105
6.5 Dateneigenschaften von Feldern 107
6.6 Funktionen im Report-Designer 107

7 Views, Trigger, Index, ... 113
7.1 Views 114
7.2 Index 117
7.3 Funktionen, Prozeduren 118
7.4 Prepared Statements 124
7.5 Trigger 125
7.6 Transaktionen 127

8 Erweiterungen 129
8.1 Datenmodell Stufe II 130
8.2 Datenmodell Stufe III 139

9 Anhang 141
9.1 Datenmodelle 142
9.2 Datentypen 145
9.3 Tools 148

SQL-Befehle, Funktionen, ... 153

Stichwortverzeichnis 157

1 Vorwort

Dieses Skript ist für den Unterrichtseinsatz in den IT-Berufen (Fachinformatiker/-in–Anwendungsentwicklung, Fachinformatiker/-in–Systemintegration, Informatikkaufmann/-frau, IT–Systemkaufmann/-frau und IT–Systemelektroniker/-in) erstellt worden, kann aber auch in vielen anderen Berufen oder anderen Schultypen oder einfach ganz privat verwendet werden, um sich in Datenbanksysteme, SQL und LibreOffice Base einzuarbeiten. Es setzt voraus, dass die Grundlagen zu Datenbanktheorie und Datenbankmodellierung bekannt sind.

Es ist so aufgebaut, dass je nach Leistungsstand und Berufsgruppe eine vertikale, wie auch horizontale Differenzierung durchgeführt werden kann. Man kann individuell die Themeninhalte auswählen, beispielsweise nur SQL auf der Konsole oder nur das Arbeiten mit einer GUI wie mit LibreOffice Base oder MySQL-Workbench. Auch der Umfang der Aufgaben und Übungen kann individuell bestimmt werden. Es gibt Aufgaben, die jeder können sollte und Aufgaben, die für sehr gute oder super schnelle Schüler geeignet sind.

Das Skript ist der „erste Wurf“ und es sind viele Ideen, Aufgaben, Übungen etc. eingeflossen, die sich so in den Jahren als Lehrer angesammelt haben. Leider sind auch viele Sachen nicht eingeflossen, da der Fertigstellungstermin immer näher rückte. Ein großer Dank geht an die vielen Helfer und Unterstützer, die mit Rat und Tat, Ideen, Anregungen, Aufgaben etc. mit geholfen haben.

Das Skript steht unter der Lizenz CC-BY-NC-ND und kann entsprechend der Lizenzbedingungen verwendet werden.[1]

Beim Einsatz werden sicher viele weitere Ideen entstehen, Ungereimtheiten oder Besonderheiten, auch Fehler genannt, auftauchen. Wir werden diese sammeln und am Ende des Schuljahres entsprechend einarbeiten. Daher würden uns über Anregungen, Kritik, Lob etc. freuen.

Wir wünschen viel Spaß und Erfolg beim Durcharbeiten

Michael Niedermair, StD

Martina Pomper, StR

[1] siehe `http://creativecommons.org/licenses/by-nc-nd/3.0/de/`

2 Einführung

Dieses Skript baut auf folgender Software auf:

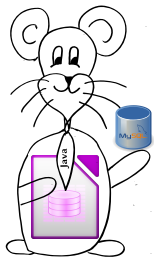
- LibreOffice Base 4 oder neuer
  `http://de.libreoffice.org/`
- MySQL
  `https://www.mysql.com/`
- MySQL Workbench
  `http://www.mysql.de/products/workbench/`

Als zusätzliche Literatur empfehlen wir:

- Handbuch zu LibreOffice Base 4 oder neuer
  `http://de.libreoffice.org/hilfe-kontakt/handbuecher/`
- MySQL 5.1 Referenzhandbuch (Deutsch)
  `http://dev.mysql.com/doc/index.html`
- MySQL 5.6 Reference Manual (Englisch)
  `http://dev.mysql.com/doc/index.html`
- Einführung in SQL
  `http://de.wikibooks.org/wiki/Einf%C3%BChrung_in_SQL`

Die hier im Skript gezeigten SQL-Beispiele werden direkt an MySQL gesendet und die erzeugte Ausgabe wird wieder eingebunden. Somit wird sichergestellt, dass die Ausgabe dem entsprechenden SQL-Statement entspricht. Verwendet man eine etwas andere Version, so kann die Ausgabe leicht abweichen. Die meisten SQL-Ausgaben werden im horizontalen Mode (normale Tabelle) ausgegeben. Bei Systemeinstellungen etc. wird in den vertikalen Modus gewechselt, da hier meist nur eine Zeile ausgegeben wird, die aber deutlich länger ist. Wichtige Leerzeichen werden bei der Ausgabe durch „␣“ hervorgehoben.

Das Skript ist so ausgelegt, dass der Lehrer hier zu jedem Bereich eine entsprechende Einführung gibt, und die Schülerin, den Schüler bei seiner Arbeit begleitet und entscheidet, welche Aufgaben hier sinnvoll sind. Für sehr gute Schüler kann das Skript auch als Selbstlernskript verwendet werden. Wenn Sie nicht sicher sind, welche Aufgaben Sie bearbeiten sollen, so fragen Sie unbedingt Ihren Lehrer. Bei Problemen studieren Sie zuerst die zusätzliche Dokumentation, fragen bei Ihren Mitschüler/-innen nach und dann erst wenden Sie sich an Ihren Lehrer.



Hinweise, Fragen, Tipps, ...

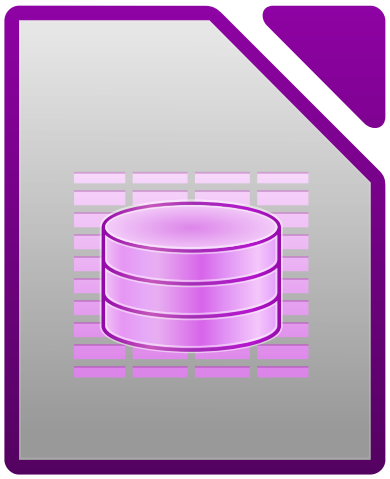
Hallo, ich bin die Maus, die Ihnen hilft, beispielsweise Fragen zu beantworten, die Sie noch gar nicht gestellt haben.
Ich bin erstellt worden aus mehreren Bildern von Michael Niedermair.

Maus	Elke Niedermair
MySQL	`https://commons.wikimedia.org/wiki/File:Database-mysql.svg` (RRZEicons) Lizenz: `https://creativecommons.org/licenses/by-sa/3.0/deed.de`
Base	`https://wiki.documentfoundation.org/Design/LibreOffice_Initial_Icons` Lizenz: `https://creativecommons.org/licenses/by-sa/3.0/deed.de`

3 Datenbanken

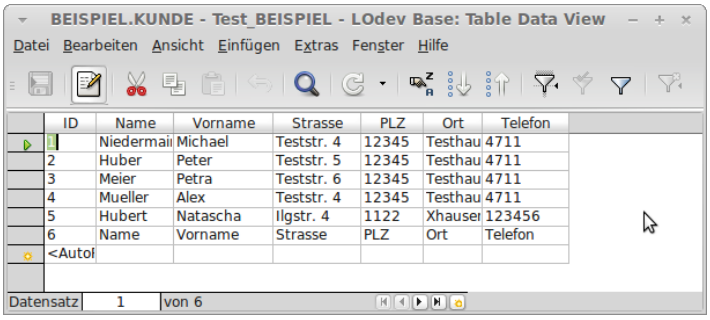
Es gibt viele verschiedene relationale SQL-Datenbanken (RDBMS). Dies kann MySQL, Oracle, HSQLDB etc. sein. In LibreOffice Base ist HSQLDB integriert, wir verwenden hier aber die externe Datenbank MySQL.

Im normalen Sprachgebrauch wird selten zwischen den Begriffen „Datenbank“ und „RDBMS“ unterschieden. Spricht man von einer Datenbank, so wird hier meist ein RDBMS-System wie MySQL oder Oracle gemeint. In der Datenbanktheorie muss hier aber unterschieden werden.



Inhalt

3.1 LibreOffice Base 10
3.1.1 Einführung 10
3.1.2 Datenbankverbindung 11
3.2 MySQL 12
3.2.1 Zugriff über Java 13
3.2.2 Zugriff über LibreOffice-Base 13
3.3 SQL - Grundlagen 16
3.3.1 Geschichte 16
3.3.2 SQL-Übersicht 16
3.4 SQL - Datenbanken erstellen 18



Ein Doppelklick auf Kunde öffnet die Tabelle zum Editieren.

3.3 SQL - Grundlagen

SQL (das Kürzel steht für **S**tructured **Q**uery **L**anguage) ist eine Datenbanksprache zur *Definition*, *Abfrage* und *Manipulation* von Daten in relationalen Datenbanken.

3.3.1 Geschichte

SQL ist von ANSI und ISO standardisiert und wird von fast allen gängigen Datenbanksystemen unterstützt.

1968 Erste Versuche im Rahmen einer studentischen Arbeit in Berkeley: „Phase Zero prototype of SEQUEL“

1974 DBMS-Sprachen von IBM wie R*, SQL/DS, und DB2

1986 erster SQL-Standard vom ANSI verabschiedet (auch **SQL1** genannt)

1987 SQL-Standard wurde von ISO ratifiziert

1992 Überarbeitung von SQL: **SQL92** (auch **SQL2** genannt)

1999 Erneute Überarbeitung: **SQL99** (auch **SQL3** genannt)

2003 Erneute Überarbeitung: **SQL:2003**

2006 Erneute Überarbeitung und Einbindung von XML: **SQL/XML**

2008 Erneute Überarbeitung: **SQL:2008**

2011 Erneute Überarbeitung: **SQL:2011**

3.3.2 SQL-Übersicht

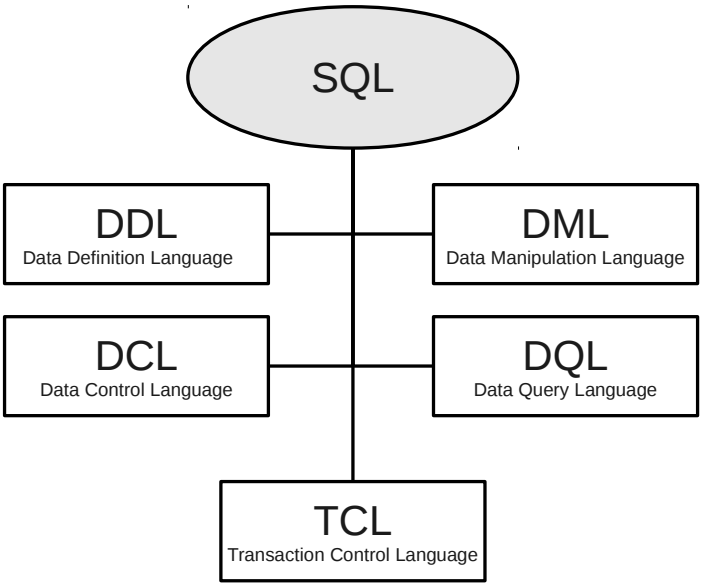


Abb. 3.9: SQL-Sprachen

SQL gliedert sich in mehrere Bereiche auf:

DDL Die Data Definition Language definiert die Struktur der Datenbank. Dies bedeutet, damit wird beispielsweise die Tabelle angelegt, geändert, gelöscht, ...
`CREATE, ALTER, DROP, ...`

```
CREATE TABLE GRUNDBEITRAEGE (
  Beitrags_ID      INTEGER NOT NULL,
  Beitragsgruppe   VARCHAR(20),
  Grundbeitrag     NUMERIC(6,2),
  PRIMARY KEY (Beitrags_ID)
)
```

DML Die Data Manipulation Language dient dazu, die Daten in Tabellen zu ändern, einzufügen, ...
`INSERT, UPDATE, DELETE, ...`

```
INSERT INTO GRUNDBEITRAEGE VALUES (0,'Familienmitglied',0.0000);
INSERT INTO GRUNDBEITRAEGE VALUES (1,'Familienbeitrag',200.0000);
INSERT INTO GRUNDBEITRAEGE VALUES (2,'Erwachsener',100.0000);
INSERT INTO GRUNDBEITRAEGE VALUES (3,'Kind␣/␣Student',60.0000);
```

DQL Die Data Query Language dient dazu, Tabellen abzufragen.
`SELECT`

```
SELECT M_ID,Nachname,Vorname,
       date_format(Geburtsdatum,'%Y-%m-%d')
       FROM MITGLIEDER ORDER BY Geburtsdatum;
```

DCL Die Data Control Language dient dazu, Zugriffsrechte zu definieren.
`GRANT, REVOKE`

```
GRANT SELECT, DELETE, UPDATE ON GRUNDBEITRAEGE TO Mueller;
```

TCL Die Transaction Control Language dient dazu, sicherzustellen, dass Transaktionen (mehrere zusammenhängende SQL-Anweisungen) gesamt abgeschlossen oder der ursprüngliche Zustand im Fehlerfall wieder hergestellt wird.
`COMMIT, ROLLBACK`

Fragen: **f_SQL_UEB**

1. Was versteht man unter SQL?
   - ☐ Standard Quadratur Language
   - ☐ Server's Quick Library
   - ☐ Structured Query Language
   - ☐ Standard Quarterly Lectures
2. Wann ist SQL entstanden?
   - ☐ 1957
   - ☐ 1968
   - ☐ 1992
   - ☐ 1999
3. Welche Teilsprachen gehören zu SQL?
   - ☐ DDL, DCL
   - ☐ DSL, JSQL
   - ☐ JQuery, XQuery
   - ☐ SDL, QUEL

3.4 SQL - Datenbanken erstellen

Datenbank: BEISPIELE

Nachfolgendes Skript erzeugt die Datenbank für die Beispiele.

```
1 # $Id: create_db_beispiel.sql 1006 2012-12-30 16:46:57Z michael $
2 #
3 # create db BEISPIEL, user test - kein PW  (MySQL 5.5.28)
4
5 CONNECT mysql;
6
7 DROP DATABASE IF EXISTS BEISPIEL;
8 CREATE DATABASE BEISPIEL CHARACTER SET utf8;
9
10 DELETE FROM user WHERE user='test';
11 INSERT INTO user SET Host='localhost',
12                     User='test',
13                     Password='';
14
15 DELETE FROM db WHERE Db='BEISPIEL';
16 INSERT INTO db SET Host='localhost',
17                   Db='BEISPIEL',
18                   User='test',
19                   Select_priv='Y',
20                   Insert_priv='Y',
21                   Update_priv='Y',
22                   Delete_priv='Y',
23                   Create_priv='Y',
24                   Drop_priv='Y',
25                   Grant_priv='Y',
26                   References_priv='Y',
27                   Index_priv='Y',
28                   Alter_priv='Y',
29                   Create_tmp_table_priv='Y',
30                   Lock_tables_priv='Y',
31                   Create_view_priv='Y',
32                   Show_view_priv='Y',
33                   Create_routine_priv='Y',
34                   Alter_routine_priv='Y',
35                   Execute_priv='Y',
36                   Event_priv='Y',
37                   Trigger_priv='Y';
38
39 FLUSH PRIVILEGES;
```

Die Zeilen haben dabei folgende Bedeutung:

1-3 Kommentare im SQL-Skript (werden ignoriert)

`#`	Kommentar bis zum Zeilenende
`--`	Kommentar bis zum Zeilenende
`/* ... */`	Kommentar von bis
`/*! MySQL-Code */`	spezieller MySQL-Code im Kommentar verpackt (dieser Code wird nur von MySQL berücksichtigt, andere SQL-Systeme ignorieren diesen)

5 Es wird eine Verbindung zur Datenbank *mysql* aufgebaut. In dieser Datenbank werden alle systeminternen Werte und Einstellungen verwaltet. Nur der User *root* kann auf diese Datenbank zugreifen.

7 Die Datenbank „BEISPIEL“ wird gelöscht, wenn diese vorhanden ist. Auf diese Weise kann das Skript mehrfach ausgeführt werden und bricht nicht ab, wenn die Datenbank schon vorhanden ist.

8 Die Datenbank „BEISPIEL“ wird erzeugt. Dabei wird als Zeichenkodierung (Encoding] UTF-8 verwendet.

10 Die Tabelle „user“ verwaltet alle User. Der User „test“ wird hier gelöscht. Ist er nicht vorhanden, passiert nichts.

11 Der User „test“ wird angelegt. Dabei darf dieser nur vom lokalen Rechner (`localhost`) zugreifen. Das Passwort bleibt leer.

15 Die Tabelle ’db’ beinhaltet alle Rechte für die Datenbanken. Die Einträge für die Datenbank „BEISPIEL“ werden erst mal alle gelöscht.

16 Der Eintrag für die Daten „BEISPIEL“ wird erstellt. Dabei darf auf die Datenbank nur vom lokalen Rechner (`localhost`) zugegriffen werden. Der User „Test“ erhält dabei alle Rechte.

39 Die Rechte sind jetzt eingetragen, aber noch nicht aktiv. Erst nach diesem Befehl werden die Einstellungen wirksam.

Der Aufruf sieht dabei wie folgt aus:

```
mysql -u root -p mysql < [Verzeichnis]create_db_beispiel.sql
```

Datenbank: FB

Nachfolgendes Skript erzeugt die Datenbank für die Fußballtabellen. Der Aufbau entspricht dabei obigem Skript.

```
# $Id: create_db_fb.sql 1000 2012-12-26 16:53:47Z michael $
#
# create db FB, user test - kein PW  (MySQL 5.5.28)

CONNECT mysql;

DROP DATABASE IF EXISTS FB;
CREATE DATABASE FB CHARACTER SET utf8;

DELETE FROM user WHERE user='test';
INSERT INTO user SET Host='localhost',
                     User='test',
                     Password='';

DELETE FROM db WHERE Db='FB';
INSERT INTO db SET Host='localhost',
                   Db='FB',
                   User='test',
                   Select_priv='Y',
                   Insert_priv='Y',
                   Update_priv='Y',
                   Delete_priv='Y',
                   Create_priv='Y',
                   Drop_priv='Y',
                   Grant_priv='Y',
                   References_priv='Y',
                   Index_priv='Y',
                   Alter_priv='Y',
                   Create_tmp_table_priv='Y',
                   Lock_tables_priv='Y',
                   Create_view_priv='Y',
                   Show_view_priv='Y',
                   Create_routine_priv='Y',
                   Alter_routine_priv='Y',
                   Execute_priv='Y',
                   Event_priv='Y',
                   Trigger_priv='Y';

FLUSH PRIVILEGES;
```

Der Aufruf sieht dabei wie folgt aus:

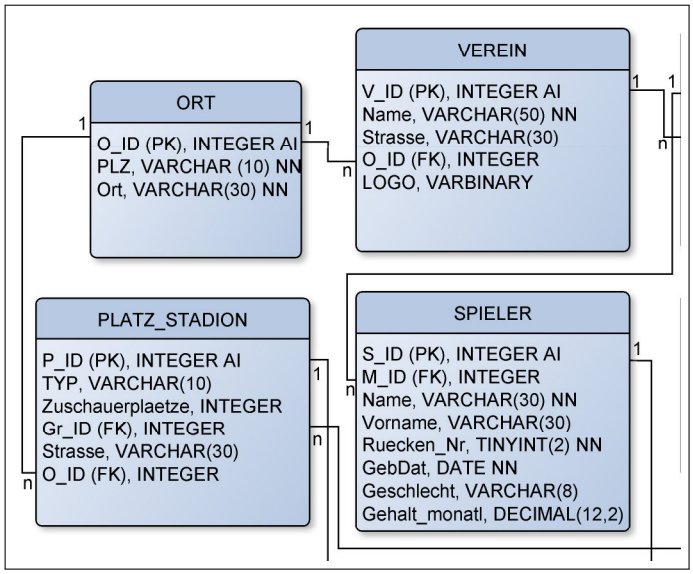
```
mysql -u root -p mysql < [Verzeichnis]create_db_fb.sql
```

Aufgaben

1. Geben Sie die beiden obigen Skripte ein (nutzen sie „copy & paste“) und erstellen Sie die beiden Datenbanken.
2. Testen Sie, ob Sie jeweils eine Verbindung vom lokalen Rechner (mittels des MySQL-Clients `mysql`) und einem entfernten Rechner (dies sollte abgelehnt werden) aufbauen können.
3. Richten Sie LibreOffice-Base so ein, dass Sie auf die Datenbank zugreifen können.

4 Tabellen

Tabellen sind die primären Grundlagen von Relationalen Datenbank-Management-Systemen (RDBMS). Daher ist das Verständnis, der Zusammenhang, die Verknüpfung und die Abfragemöglichkeiten von Tabellen sehr wichtig.



Inhalt

4.1 SQL - Tabellen anlegen 22
4.1.1 Tabellen löschen 24
4.1.2 Tabellen ändern 25
4.2 DFB Tabellen mit SQL erstellen (Stufe 1) 26
4.3 Tabellenentwurf mit MySQL-Workbench 27
4.4 DFB Tabellen mit LibreOffice Base erstellen (Stufe 1) 32
4.5 Verknüpfung von Tabellen - Beziehungen 34
4.6 SQL - Eingabe von Daten 36
4.6.1 Datensätze aus Textdatei laden 37
4.6.2 Programm zum Datenimport 38
4.7 SQL - Ändern von Daten 38
4.8 Eingabe von Daten in Tabellen mit LibreOffice Base 40
4.8.1 Eingabe über die grafische Benutzeroberfläche der Tabelle 40
4.8.2 Formulare als Eingabeerleichterung 41

4.1 SQL - Tabellen anlegen

`CREATE TABLE` erzeugt eine Tabelle. Dabei sieht der vereinfachte Aufruf wie folgt aus:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
   [(create_definition,...)]
   [table_options]
   [select_statement]
```

Standardmäßig wird die Tabelle in der Standarddatenbank erstellt. Ist die Tabelle bereits vorhanden, wird ein Fehler ausgegeben. Über `IF NOT EXISTS` wird sichergestellt, dass die Tabelle nur erstellt wird, wenn diese noch nicht vorhanden ist.

Der Tabellenname kann als `db_name.tbl_name` angegeben werden, um die Tabelle in einer bestimmten Datenbank zu erstellen. Die Namen können auch separat in einfache Anführungszeichen gesetzt werden, z. B. `'mydb'.'mytbl'`.

Das Schlüsselwort `TEMPORARY` bei der Erstellung einer Tabelle sorgt dafür, dass die Tabelle nur für die aktuelle Verbindung sichtbar ist und beim Beenden automatisch gelöscht wird. Temporäre Tabellen werden nur für die jeweilige Sitzung angelegt und sind in anderen Sitzungen nicht zu sehen. Daher können die selben Tabellennamen in unterschiedlichen Sitzungen verwendet werden.

Die Regeln für zulässige Tabellennamen sind in Tabelle 4.1 dargestellt.

Bezeichner	*Maximale Länge (Byte)*	*Erlaubte Zeichen*
Datenbank	64	Jedes Zeichen, das in dem Verzeichnisnamen des Dateisystems zulässig ist (ohne '/', '\' und '.').
Tabelle	64	Jedes Zeichen, das in dem Dateinamen des Dateisystems zulässig ist (ohne '/', '\' und '.').
Spalte	64	Alle Zeichen
Index	64	Alle Zeichen
Alias	255	Alle Zeichen

Tab. 4.1: Namensregeln für Datenbanken, Tabellen, ...

Felddefinition

```
col_name type [NOT NULL | NULL] [DEFAULT default_value]
[AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
[COMMENT 'string'] [reference_definition]
```

Feldname (`col_name`) Der Feldname definiert den Name der Spalte. Dieser muss innerhalb der Tabelle eindeutig sein.

Datentyp (`type`) Eine Übersicht über die erlaubten Datentypen der Spalten findet sich unter Abschnitt 9.2 auf Seite 145.

`NOT NULL` Der Wert der Spalte darf nicht `NULL` sein und wird vom DBMS garantiert. Wichtig beim Primärschlüssel. Das Gegenstück ist `NULL` (Standardwert).

`DEFAULT` Legt den Standardwert für die Spalte fest.

`AUTO_INCREMENT` Bei numerischen Werten, die als Primärschlüssel verwendet werden, wird der Wert automatisch hochgezählt.[1]

`UNIQUE KEY` Legt fest, dass jedes Feld innerhalb der betroffenen Spalte eindeutig zu sein. Im Gegensatz zum `PRIMARY KEY`, kann `UNIQUE KEY` bei mehreren Spalten verwendet werden.

`PRIMARY KEY` Legt fest, dass die Spalte der Primärschlüssel der Tabelle ist (Kurzform `KEY`).[2] Bei einem zusammengesetzten Schlüssel muss dies in der Bedingungsdefinition erfolgen.

`COMMENT` Ein Kommentar.

[1] Mit `ALTER TABLE <tabelle> AUTO_INCREMENT = N` wird der Wert auf `N` zurückgesetzt.
[2] Mit der Funktion `LAST_INSERT_ID()` wird der letzte erzeugte Schlüssel ermittelt.

Beispiel

Nachfolgend wird die Tabelle Kunde angelegt.

```
CREATE TABLE IF NOT EXISTS KUNDE (
  ID        INTEGER      NOT NULL AUTO_INCREMENT PRIMARY KEY,
  Name      VARCHAR(30)  NOT NULL,
  Vorname   VARCHAR(30),
  Strasse   VARCHAR(30),
  PLZ       CHAR(5),
  Ort       VARCHAR(20),
  Telefon   VARCHAR(20)
)
```

Mit dem Befehl `DESCRIBE KUNDE` (Kurzform `DESC KUNDE`) wird die Tabellendefinition ausgegeben.

```
DESC KUNDE;
```

```
+---------+-------------+------+-----+---------+----------------+
| Field   | Type        | Null | Key | Default | Extra          |
+---------+-------------+------+-----+---------+----------------+
| ID      | int(11)     | NO   | PRI | null    | auto_increment |
| Name    | varchar(30) | NO   |     | null    |                |
| Vorname | varchar(30) | YES  |     | null    |                |
| Strasse | varchar(30) | YES  |     | null    |                |
| PLZ     | char(5)     | YES  |     | null    |                |
| Ort     | varchar(20) | YES  |     | null    |                |
| Telefon | varchar(20) | YES  |     | null    |                |
+---------+-------------+------+-----+---------+----------------+
```

Man kann auch den kompletten SQL-Befehl mit `SHOW CREATE TABLE` ausgeben lassen. Dabei werden nicht angegebene Default-Werte angezeigt. Auch eine evtl. Optimierung wird hier deutlich (Ausgabe vertikal).

```
SHOW CREATE TABLE KUNDE;
```

```
Table=KUNDE
Create Table=CREATE TABLE `KUNDE` (
  `ID` int(11) NOT NULL AUTO_INCREMENT,
  `Name` varchar(30) NOT NULL,
  `Vorname` varchar(30) DEFAULT NULL,
  `Strasse` varchar(30) DEFAULT NULL,
  `PLZ` char(5) DEFAULT NULL,
  `Ort` varchar(20) DEFAULT NULL,
  `Telefon` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Bei `ENGINE` wird hier automatisch „InnoDB“ angegeben. Damit die Engine sicher verwendet wird, sollte der Eintrag immer erfolgen.

Mit dem Befehl `SHOW ENGINES` werden alle vorhandenen Engines angezeigt.

```
mysql> SHOW ENGINES;
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| Engine             | Support | Comment                                                        | Transactions | XA   | Savepoints |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
| MyISAM             | YES     | MyISAM storage engine                                          | NO           | NO   | NO         |
| MRG_MYISAM         | YES     | Collection of identical MyISAM tables                          | NO           | NO   | NO         |
| MEMORY             | YES     | Hash based, stored in memory, useful for temporary tables      | NO           | NO   | NO         |
| BLACKHOLE          | YES     | /dev/null storage engine (anything you write to it disappears) | NO           | NO   | NO         |
| CSV                | YES     | CSV storage engine                                             | NO           | NO   | NO         |
| FEDERATED          | NO      | Federated MySQL storage engine                                 | NULL         | NULL | NULL       |
| ARCHIVE            | YES     | Archive storage engine                                         | NO           | NO   | NO         |
| InnoDB             | DEFAULT | Supports transactions, row-level locking, and foreign keys     | YES          | YES  | YES        |
| PERFORMANCE_SCHEMA | YES     | Performance Schema                                             | NO           | NO   | NO         |
+--------------------+---------+----------------------------------------------------------------+--------------+------+------------+
```

Bedingungsdefinition

Bedingungsdefinitionen (Constraints) definieren Bedingungen, die beim Einfügen von Daten erfüllt sein müssen. Diese werden direkt nach der Felddefinition eingefügt. Constraints können einen Namen haben.

```
column_definition
| [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
| KEY [index_name] [index_type] (index_col_name,...)
| INDEX [index_name] [index_type] (index_col_name,...)
| [CONSTRAINT [symbol]] UNIQUE [INDEX]
[index_name] [index_type] (index_col_name,...)
| FULLTEXT [INDEX] [index_name] (index_col_name,...)
[WITH PARSER parser_name]
| SPATIAL [INDEX] [index_name] (index_col_name,...)
| [CONSTRAINT [symbol]] FOREIGN KEY
```

```
[index_name] (index_col_name,...) [reference_definition]
| CHECK (expr)
```

PRIMARY KEY Legt den Primärschlüssel der Tabelle fest. Hier können auch mehrere Spalten und auch Funktionen angegeben werden.

FOREIGN KEY Zwischen zwei Tabellen kann ein Fremdschlüssel-Contraint angelegt werden (Primärschlüssel in der einen Tabelle ist Fremdschlüssel in der anderen Tabelle) -> „Referentielle Integrität". Die `REFERENCES`-Klausel gibt die Tabelle und die Indexspalten an, die die Bedingung für diesen Constraint darstellen. Über `ON DELETE` und `ON UPDATE` werden Aktionen angegeben, die bei entsprechenden Operationen auf der Haupttabelle, bei der abhängigen Tabelle erfolgen.

CASCADE bewirkt ein automatisches Löschen bei der abhängigen Tabelle

SET DEFAULT bewirkt ein automatisches Setzen auf den `DEFAULT`-Wert der Spalte[3]

SET NULL bewirkt ein automatisches auf `NULL` Setzen bei der abhängigen Tabelle

Ist die Tabelle, auf die der `FOREIGN KEY` verweist, noch nicht erstellt, so führt dies zu einem Fehler. Eine gängige Methode ist daher, zuerst alle Tabellen ohne `FOREIGN KEY` zu erstellen und dann über `ALTER TABLE` die Tabelle anzupassen. Beispiel:

```
ALTER TABLE VEREIN ADD FOREIGN KEY (O_ID) REFERENCES ORT (O_ID);
```

CHECK Wird eine neue Zeile eingefügt oder eine Zeile geändert, so muss die Bedingung bei `CHECK` erfüllt sein, ansonsten wird die Änderung verweigert.

MySQL bietet noch eine Vielzahl an weiteren Einstellmöglichkeiten beim Erstellen von Tabellen. Mehr dazu im MySQL-Handbuch.

4.1.1 Tabellen löschen

Mit `DROP TABLE` wird eine Tabelle wieder gelöscht.

```
DROP TABLE "Tabellenname" [IF EXISTS] [RESTRICT | CASCADE];
```

Löscht die Tabelle „Tabellenname".

IF EXISTS Löscht die Tabelle nur, wenn diese schon existiert. Somit werden Fehlermeldungen in Skripten vermieden, wenn die Tabelle nicht existiert.

RESTRICT Die Tabelle wird nur gelöscht, wenn diese nicht mit einer anderen Tabelle durch einen Fremdschlüssel oder mit einer View verbunden ist (Standardeinstellung).

RESTRICT CASCADE Die Tabelle wird gelöscht. In den verknüpften Tabellen werden dann alle Fremdschlüsselfelder auf `NULL` gesetzt. Alle Views, in denen auf die entsprechende Tabelle Bezug genommen wird, werden komplett gelöscht.

MySQL unterstützt den Parameter `CASCADE` nicht. Wird eine Tabelle gelöscht, die mit einem Fremdschlüssel auf eine andere Tabelle verweist, wird die Ausführung mit einer Fehlermeldungen abgebrochen. Das Problem kann umgangen werden, in dem man bei MySQL die Überprüfung auf den Fremdschlüssel abschaltet.

Beispiel

```
/* FK-/UNIQUE-Check ausschalten */
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;

/* Tabellen löschen */
DROP TABLE IF EXISTS VEREIN;
DROP TABLE IF EXISTS SPIELER;
DROP TABLE IF EXISTS ZT_SPIEL_SPIELER;
DROP TABLE IF EXISTS ORT;
DROP TABLE IF EXISTS PLATZ_STADION;
DROP TABLE IF EXISTS MANNSCHAFT;
DROP TABLE IF EXISTS ZT_SPIEL_MANNSCHAFT;
DROP TABLE IF EXISTS SPIEL;
DROP TABLE IF EXISTS KATEGORIE;
DROP TABLE IF EXISTS GROESSE_SPIELFELD;
DROP TABLE IF EXISTS LIGA;
```

[3] `SET DEFAULT` funktioniert nicht mit der GUI. Die GUI verwaltet separate „Standardwerte", die nicht als Defaultwerte in der Datenbank vorhanden sind.

```
/* FK-/UNIQUE-Check einschalten */
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

4.1.2 Tabellen ändern

Mit `ALTER TABLE` wird eine Tabelle geändert.

```
ALTER TABLE "Tabellenname" [...]
```

ADD

Fügt eine neue Spalte hinzu. Wird zusätzlich `BEFORE` verwendet, so kann damit die Position bestimmt werden. Die zusätzliche Bezeichnung `COLUMN` ist dabei nicht erforderlich.

```
ALTER TABLE "Tabellenname" ADD [COLUMN] <Felddefinition>
      [BEFORE "bereits_existierender_Feldname"];
ALTER TABLE "Adresse" ADD "Vorname" VARCHAR(25) BEFORE "Name";
```

Fremdschlüssel hinzufügen:

```
ALTER TABLE "Tabellenname" ADD FOREIGN KEY (COLUMN)
      REFERENCES "Ref-Tabellenname (REF-COLUMN);
ALTER TABLE VEREIN ADD FOREIGN KEY (O_ID) REFERENCES ORT (O_ID);
```

DROP

Löscht eine Spalte, wenn diese nicht in einer anderen Tabelle als Fremdschlüssel oder in einer View verwendet wird.

```
ALTER TABLE "Tabellenname" DROP [COLUMN] "Feldname";
```

ALTER ...RENAME TO

Ändert den Namen eines Feldes.

```
ALTER TABLE "Tabellenname" ALTER COLUMN "Feldname"
      RENAME TO "neuer_Feldname"
```

ALTER ...SET DEFAULT

Fügt dem Feld einen bestimmten Standardwert hinzu. NULL entfernt einen bestehenden Standardwert.

```
ALTER TABLE "Tabellenname" ALTER COLUMN "Feldname"
      SET DEFAULT <Standardwert>};
```

ALTER ...SET NULL

Setzt oder entfernt eine NOT NULL Bedingung für ein Feld.

```
ALTER TABLE "Tabellenname" ALTER COLUMN "Feldname"
      SET [NOT] NULL
```

ALTER <Felddefinition>

Ändert die Felddefinition. Die Felddefinition entspricht im Wesentlichen der aus der Tabellenerstellung.

```
ALTER TABLE "Tabellenname" ALTER COLUMN <Felddefinition>;
```

4.2 DFB Tabellen mit SQL erstellen (Stufe 1)

Für unser Beispiel "DFB" werden insgesamt drei verschiedene Ausbaustufen verwendet. Für die ersten Beispiele wird ein sehr einfaches Modell benutzt, welches dann schrittweise erweitert wird.

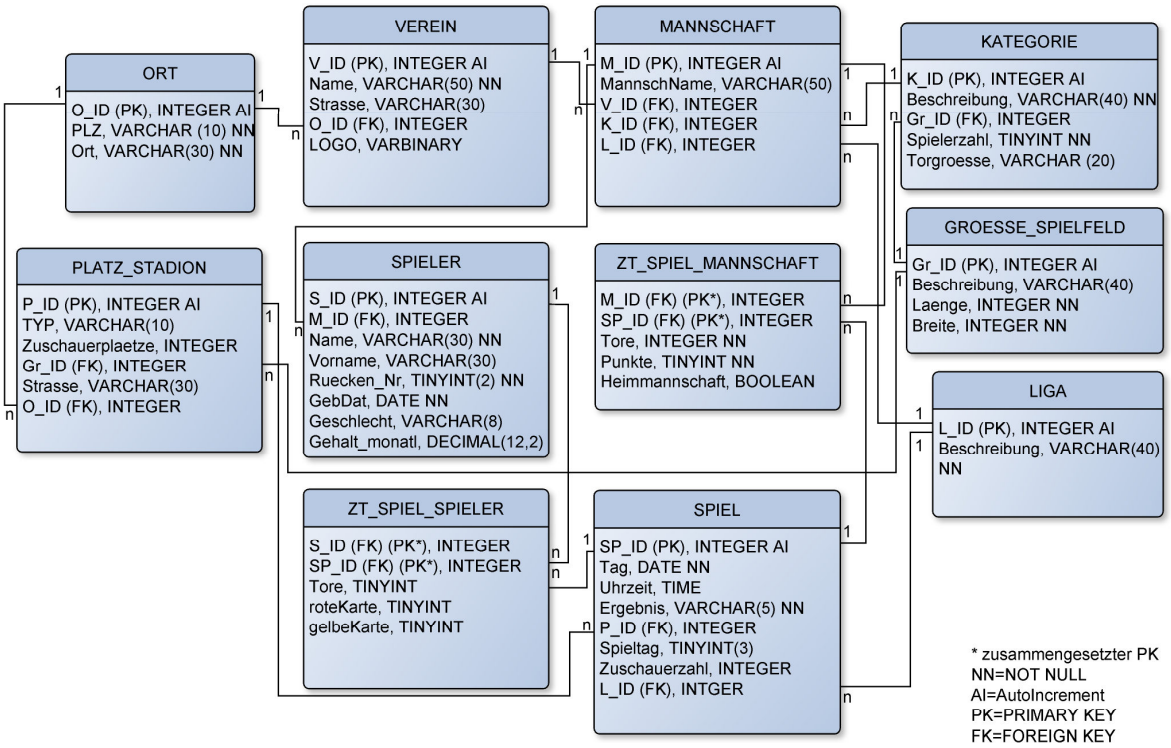


Abb. 4.1: DFB-Modell – Stufe 1

4.3 Tabellenentwurf mit MySQL-Workbench

Die MySQL-Workbench ist ein konsolidiertes visuelles Werkzeug für Datenbankarchitekten, Entwickler und DBAs. MySQL-Workbench bietet Funktionen für Datenmodellierung und SQL-Entwicklung sowie leistungsstarke Verwaltungstools für Serverkonfiguration, Benutzerverwaltung und vieles mehr. MySQL-Workbench ist auf Windows, Linux und Mac OS verfügbar. Die Workbench ermöglicht es DBAs, Entwicklern und Datenarchitekten, beliebige Arten von Datenbanken visuell zu entwerfen, zu modellieren, zu erstellen und zu verwalten. Die MySQL Workbench stellt visuelle Werkzeuge für das Erstellen, Ausführen und Optimieren von SQL-Abfragen bereit.[4]

Das MySQL-Workbench teilt sich in drei Funktionsbereiche auf:

1. SQL Development:
   Hier kann die Verbindung zu bestehenden Datenbanken hergestellt werden und mit entsprechenden SQL-Statements (DDL, DML, DQL) die Datenbank bearbeitet und Abfragen erstellt werden.
2. Data Modeling
   Hier können grafische Datenbankmodelle erstellt und bearbeitet werden. Außerdem ist über diese Funktion ein Export in die entsprechenden SQL-Create-Statements möglich. Mit diesem Tool kann also zunächst die Datenbank modelliert und anschließend die MySQL-Datenbank generiert werden. Außerdem erstellt dieses Tool aus einer bestehenden MySQL-Datenbank ein relationales Datenbankmodell.
3. Server Administration
   Im dritten Funktionsbereich des MySQL-Workbench kann z. B. der Datenbankserver konfiguriert werden oder Benutzerkonten eingerichtet und gepflegt werden.

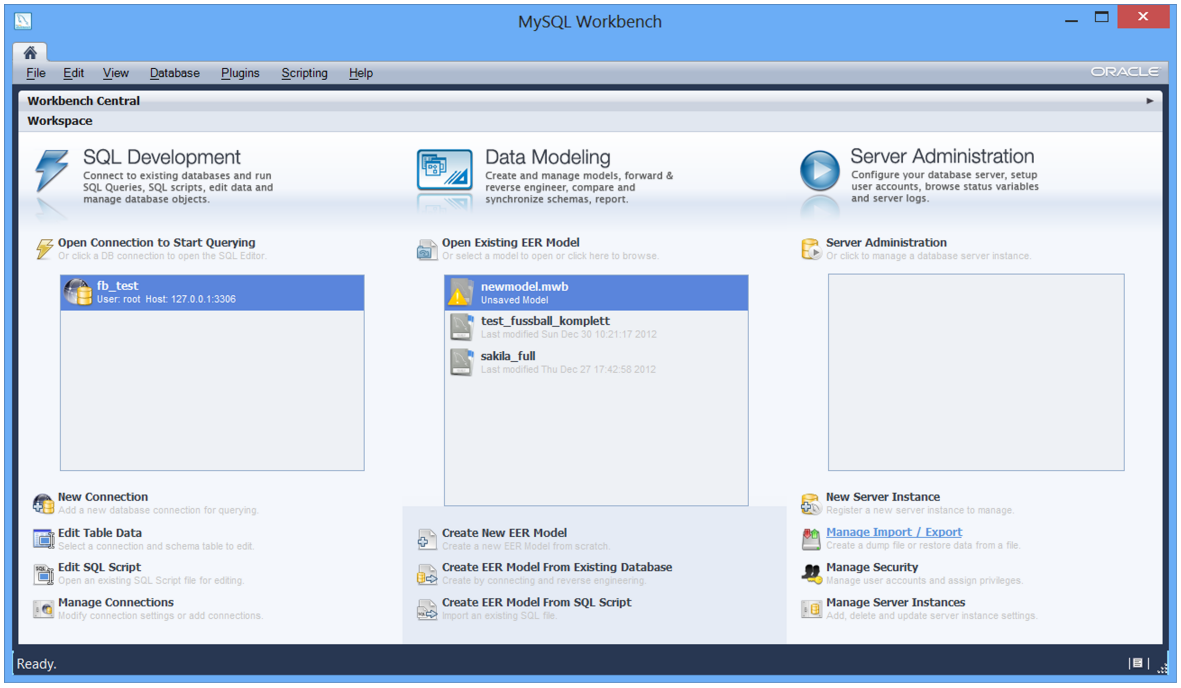


Abb. 4.2: Startbildschirm der MySQL-Workbench

[4] Quellenangabe: `http://www.mysql.de/products/workbench`

Übung:

1. SQL Development:
   Stellen Sie, entsprechend der folgenden Anweisungen, die Verbindung zur Datenbank `FB` her und testen Sie einfache SQL-Statements.

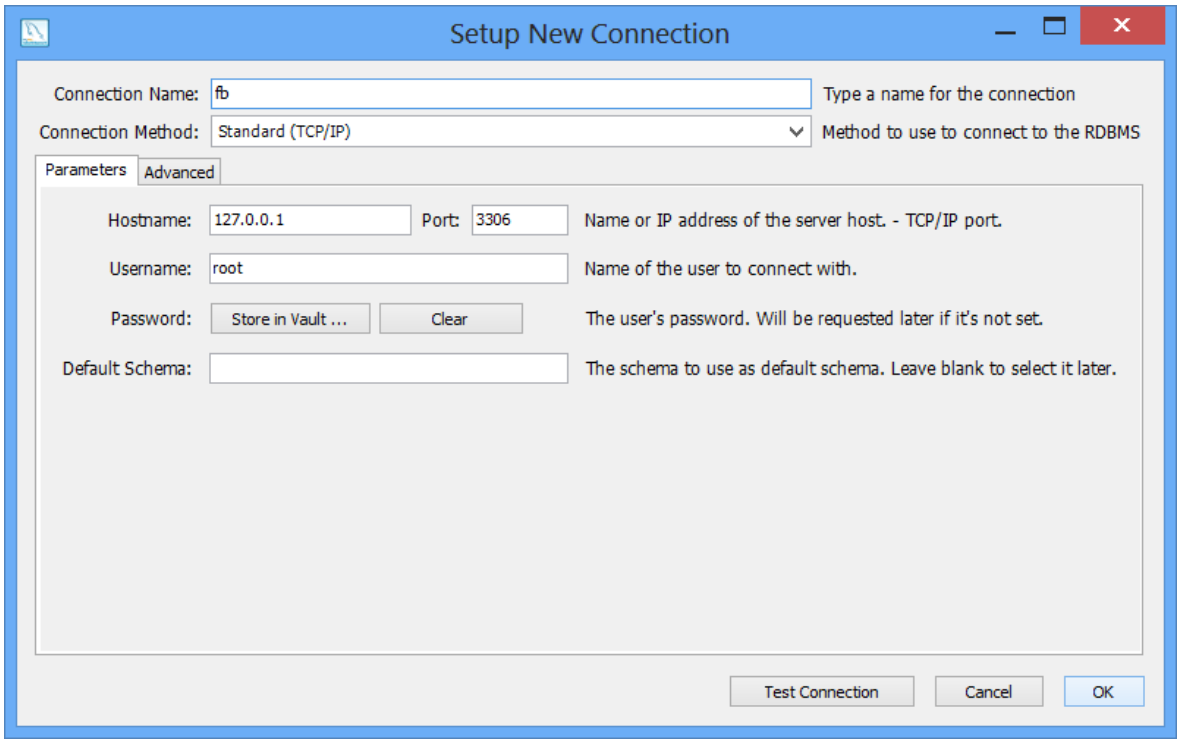


Abb. 4.3: New Connection

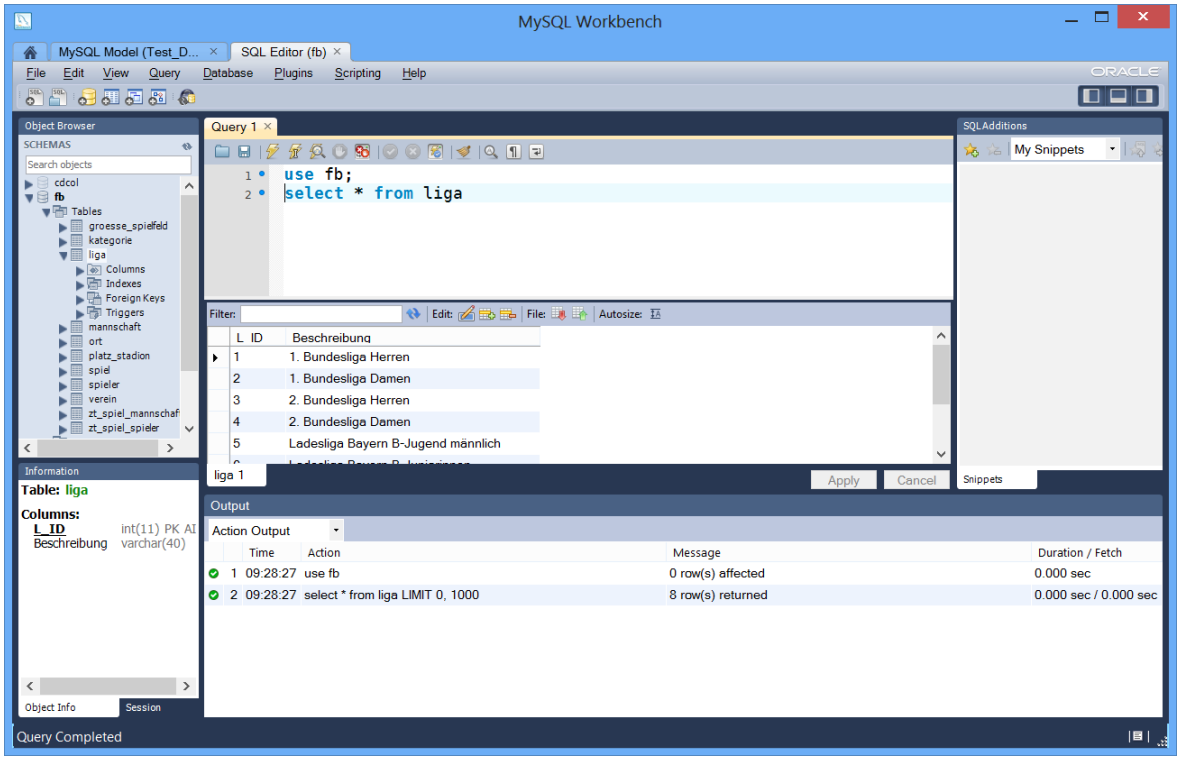


Abb. 4.4: Test mit SQL-Statement

2. Data Modeling:
   Entwerfen Sie, entsprechend des folgenden Beispiels, ein einfaches Datenbankmodell. Anschließend exportieren Sie daraus das SQL-Create-Statement.

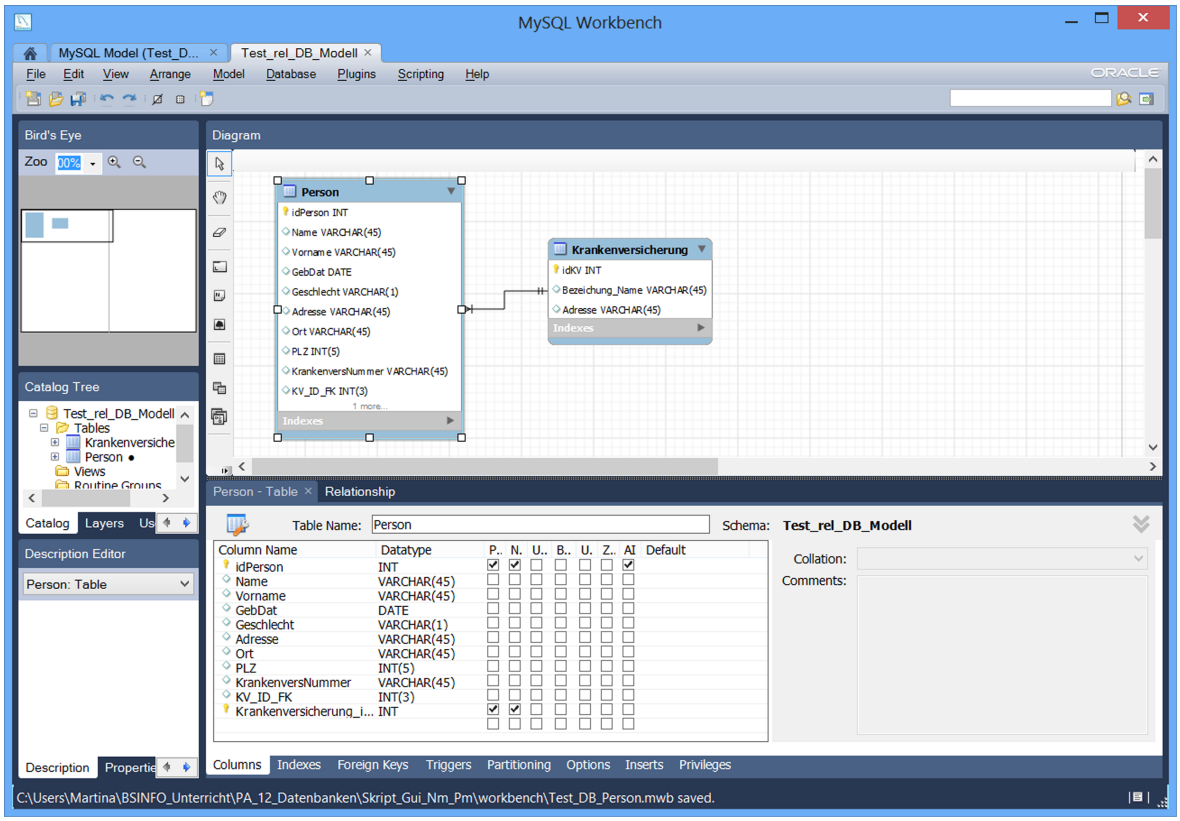


Abb. 4.5: Create New ER Model

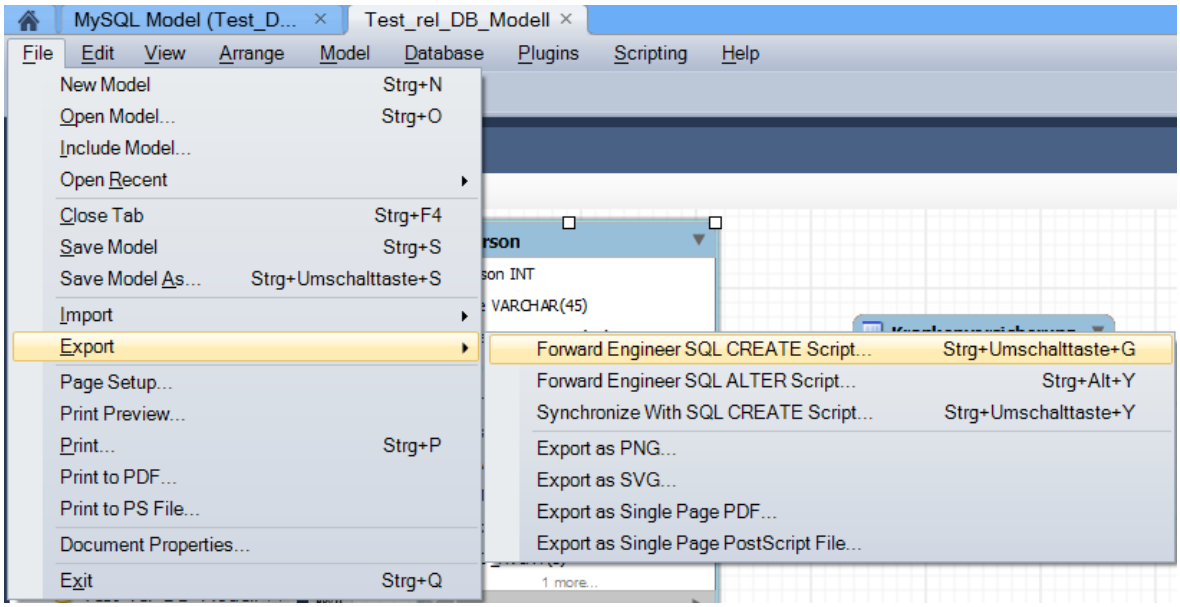


Abb. 4.6: Export

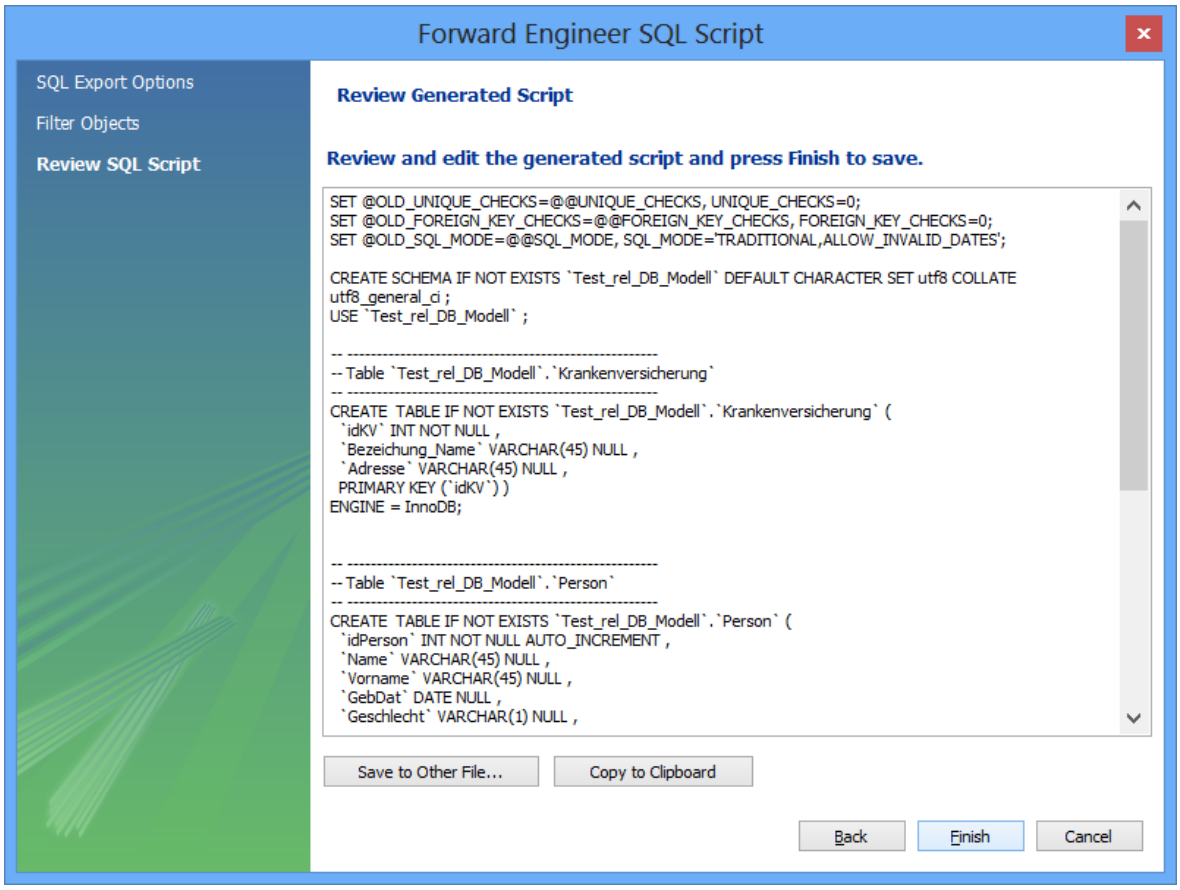


Abb. 4.7: Generiertes SQL-Skript

Die Anweisung `SET` weist verschiedenen Arten von Variablen Werte zu. Die Benutzervariablen (`@...`) in `SET`-Anweisungen werden als neue Systemvariablen (`@@...`) referenziert.

Beispiel:

Setzt man den Fremdschlüssel nun durch die `SET`-Anweisungen auf einen neuen Fremdschlüssel mit `checks=0` bedeutet dies, dass die Fremdschlüsselüberprüfung ausgeschaltet wird. So kann ein Fremdschlüssel gesetzt werden, auch wenn noch keine entsprechenden Datensätze in der Tabelle vorhanden sind.

Die MySQL-Workbench generiert diese `SET`-Anweisungen, um Probleme bei der Ausführung des SQL-Scripts zu vermeiden.

3. SQL Development:
   Generieren Sie das relationale DB-Modell zur bestehenden Datenbank `FB`.

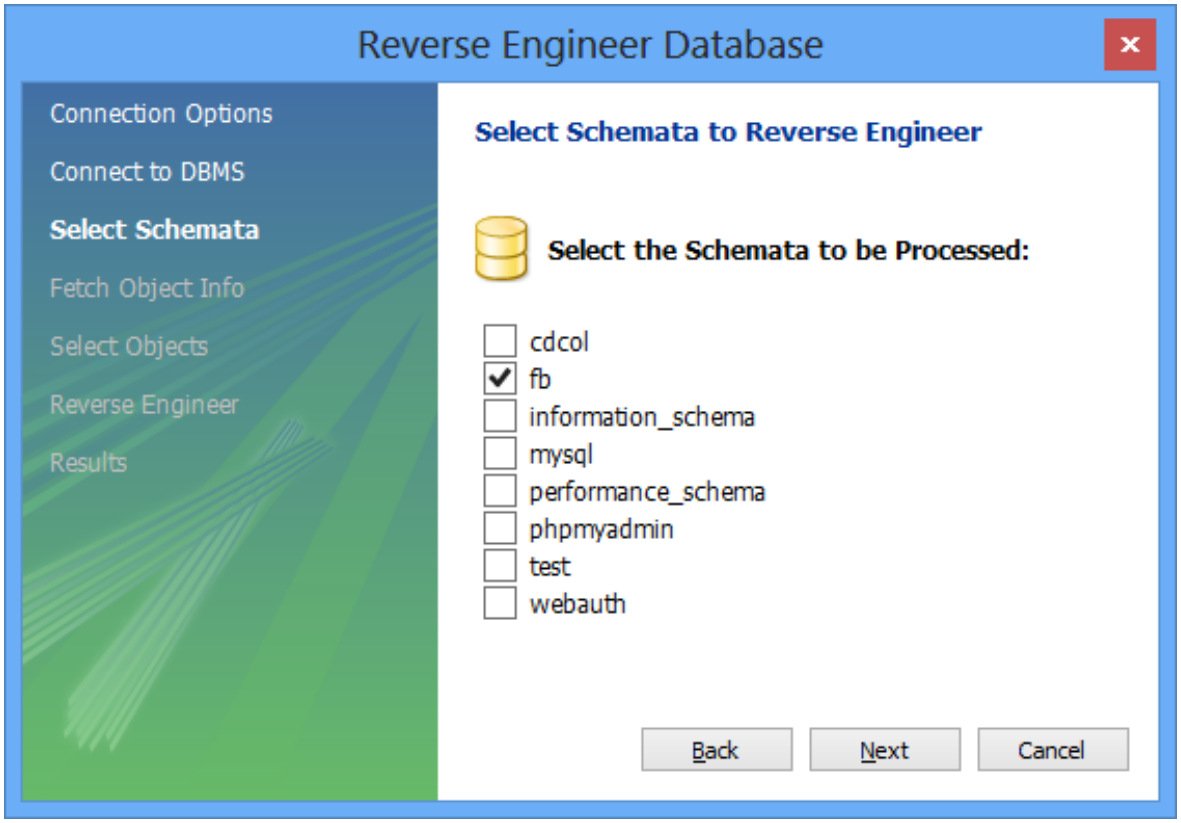


Abb. 4.8: Create EER Modell from Existing Database

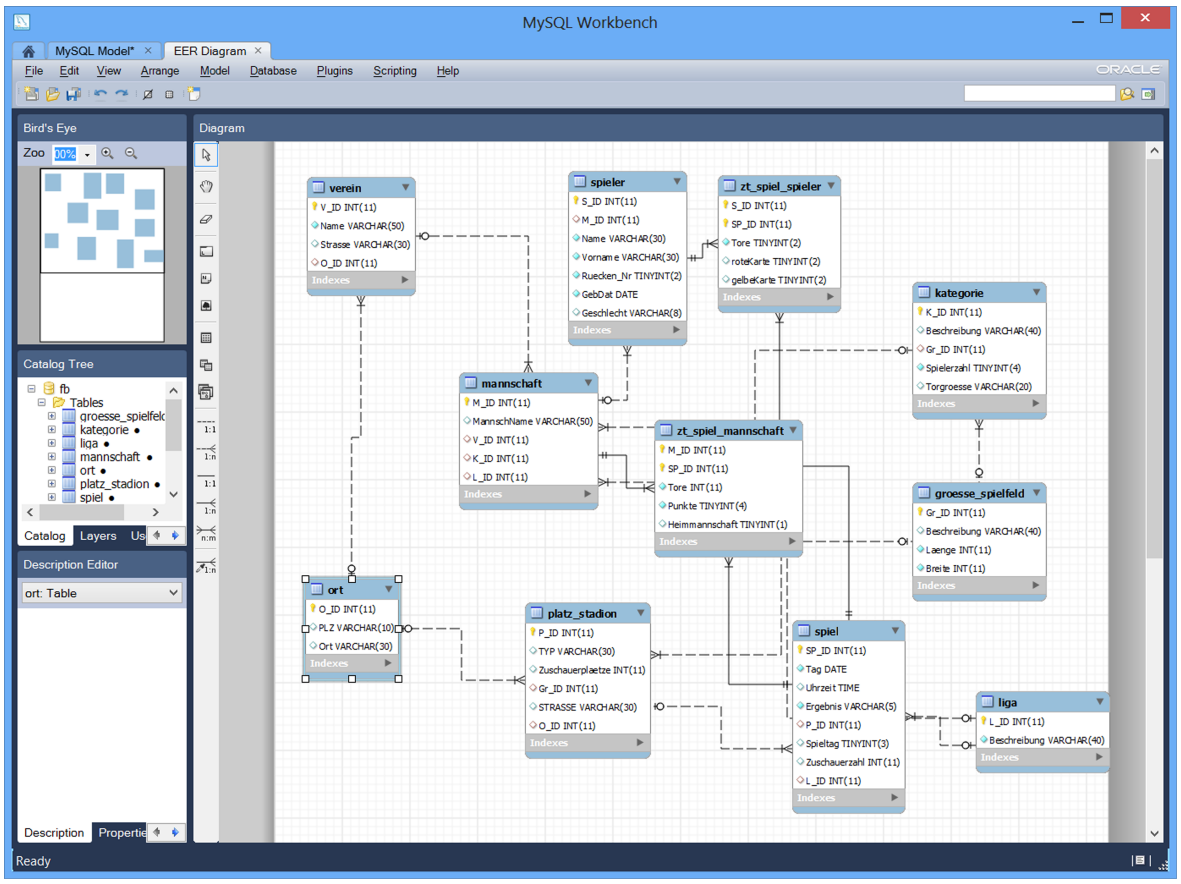


Abb. 4.9: Select Schema

4.6 SQL - Eingabe von Daten

Mit dem Befehl `INSERT` werden Daten in eine Tabelle eingefügt.

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

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

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO] tbl_name [(col_name,...)]
   SELECT ...
   [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Mit `INSERT ... VALUES` und `INSERT ... SET` werden die angegebenen Daten eingefügt. Mit `INSERT ... SELECT` werden Daten aus einer oder mehreren Tabellen eingefügt.

LOW_PRIORITY Die Ausführung von `INSERT` wird verzögert, bis kein Client mehr aus der Tabelle liest.

DELAYED Die Datensätze werden in einem Puffer abgelegt. Die weitere Verarbeitung wird sofort fortgesetzt. Sobald die Tabelle frei ist, startet der Server mit dem Einfügen der Datensätze und prüft dabei periodisch, ob neue Leseanforderungen für die Tabelle anstehen.

HIGH_PRIORITY Die Datensätze werden sofort in die Tabelle geschrieben. Gleichzeitige Einfügeoperationen sind damit nicht möglich.

IGNORE Fehler während der Ausführung werden als Warnungen behandelt und es wird nicht abgebrochen.

VALUES Es werden die Werte zu den vorher angegebenen Spalten eingefügt. Werden die Spalten nicht angegeben, so muss die Reihenfolge der Werte der Reihenfolge der Spalten entsprechen, wie sie bei `CREATE TABLE` angegeben worden sind. Es ist auch möglich, hier mehrere Zeilen einzufügen.

```
INSERT INTO WERTE (a,b,c)
  VALUES ('a1','a2','a3'),('b1','b2','b3'),('c1','c2','c3');
```

DEFAULT Hiermit wird der Wert einer Spalte explizit auf ihren Standardwert gesetzt. Somit kann eine Reihe verwendet werden, ohne dass für die Spalte ein Wert angegeben werden muss.

ON DUPLICATE KEY UPDATE Mit `ON DUPLICATE KEY UPDATE` wird anstelle von `INSERT` ein `UPDATE` verwendet, wenn der eindeutige Schlüssel schon in der Tabelle enthalten ist.

SET Hiermit wird die Spalte explizit angegeben.

```
INSERT INTO WERTE SET a='a1', b='a2', c='a3';
```

Der Befehl `INSERT ... SELECT` wird später behandelt.

Beispiel

```
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Niedermair','Michael','Teststr.␣4','12345','Testhausen','4711');
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Huber','Peter','Teststr.␣5','12345','Testhausen','4711');
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Meier','Petra','Teststr.␣6','12345','Testhausen','4711');
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Mueller','Alex','Teststr.␣4','12345','Testhausen','4711');
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Hubert','Natascha','Ilgstr.␣4','1122','Xhausen','123456');
INSERT INTO KUNDE (Name,Vorname,Strasse,PLZ,Ort,Telefon)
       VALUES('Name','Vorname','Strasse','PLZ','Ort','Telefon');
```

Werte ausgeben

Die Werte lassen sich mit `SELECT` ausgeben.

```
SELECT * FROM KUNDE;
```

```
+----+------------+----------+------------+-------+------------+---------+
| ID | Name       | Vorname  | Strasse    | PLZ   | Ort        | Telefon |
+----+------------+----------+------------+-------+------------+---------+
| 1  | Niedermair | Michael  | Teststr. 4 | 12345 | Testhausen | 4711    |
| 2  | Huber      | Peter    | Teststr. 5 | 12345 | Testhausen | 4711    |
| 3  | Meier      | Petra    | Teststr. 6 | 12345 | Testhausen | 4711    |
| 4  | Mueller    | Alex     | Teststr. 4 | 12345 | Testhausen | 4711    |
| 5  | Hubert     | Natascha | Ilgstr. 4  | 1122  | Xhausen    | 123456  |
| 6  | Name       | Vorname  | Strasse    | PLZ   | Ort        | Telefon |
+----+------------+----------+------------+-------+------------+---------+
```

4.6.1 Datensätze aus Textdatei laden

Mit dem Befehl `LOAD DATA INFILE` lassen sich sehr schnell Datensätze aus einer Textdatei laden. Hier eine kurze Übersicht des Befehls, mehr dazu im MySQL-Handbuch.

```
LOAD DATA INFILE 'file_name' INTO TABLE tbl_name
   [FIELDS [TERMINATED BY 'string']
   [[OPTIONALLY] ENCLOSED BY 'char']
   [ESCAPED BY 'char']
   [LINES [TERMINATED BY 'string']
```

Beispiel

Die CSV-Datei sieht wie folgt aus:

```
1,"Niedermair","Michael","Teststr. 4","12345","Testhausen","4711"
2,"Huber","Peter","Teststr. 5","12345","Testhausen","4711"
3,"Meier","Petra","Teststr. 6","12345","Testhausen","4711"
4,"Mueller","Alex","Teststr. 4","12345","Testhausen","4711"
5,"Hubert","Natascha","Ilgstr. 4","1122","Xhausen","123456"
6,"Name","Vorname","Strasse","PLZ","Ort","Telefon"
```

Dann wird diese wie folgt eingelesen:[5]

```
LOAD DATA INFILE '/tmp/x.csv' INTO TABLE KUNDE
   FIELDS TERMINATED BY ','
   OPTIONALLY ENCLOSED BY '"'
   LINES TERMINATED BY '\n';
```

Der Export der CSV-Datei erfolgt über nachfolgenden Befehl als Administrator.

```
SELECT * INTO OUTFILE '/tmp/x.csv'
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   LINES TERMINATED BY '\n'
   FROM KUNDE;
```

[5] Die entsprechenden Administrator-Rechte müssen vorhanden sein!

4.6.2 Programm zum Datenimport

Mit dem Programm `mysqlimport` lassen sich auch sehr schnell Daten aus Textdateien importieren. Die Parameter sind dem `LOAD DATA INFILE` sehr ähnlich.

```
shell> mysqlimport [options] db_name textfile1 [textfile2 ...]
```

Die wichtigsten Parameter sind:

`--columns=column_list`	oder `-c column_list` Diese Option nimmt eine kommagetrennte Liste der Spaltennamen als Wert entgegen. Die Reihenfolge der Spaltennamen muss den Spalten der Textdatei entsprechen.
`--fields-terminated-by=...`	siehe `FIELDS TERMINATED BY 'string'`
`--fields-enclosed-by=...`	siehe `FIELDS ENCLOSED BY 'char'`
`--fields-optionallyenclosed-by=...`	siehe `FIELDS OPTIONALLY ENCLOSED BY 'char'`
`--fields-escaped-by=...`	siehe `FIELDS ESCAPED BY 'char'`
`--lines-terminated-by=...`	siehe `LINES TERMINATED BY 'string'`

4.7 SQL - Ändern von Daten

Mit dem Befehl `UPDATE` werden Daten in einer Tabelle geändert. Dieser ähnelt dem `INSERT` sehr.

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
   SET col_name1=expr1 [, col_name2=expr2 ...]
   [WHERE where_condition]
   [ORDER BY ...]
   [LIMIT row_count]

UPDATE [LOW_PRIORITY] [IGNORE] table_references
   SET col_name1=expr1 [, col_name2=expr2 ...]
   [WHERE where_condition]
```

`LOW_PRIORITY`	Die Ausführung von `UPDATE` wird verzögert, bis kein Client mehr aus der Tabelle liest.
`IGNORE`	Fehler während der Ausführung werden als Warnungen behandelt und es wird nicht abgebrochen.
`SET`	Hiermit wird die Spalte angegeben.
`WHERE`	Legt fest, welche Zeilen geändert werden sollen.
`ORDER BY`	Legt fest, in welcher Reihenfolge die Datensätze aktualisiert werden.
`LIMIT`	Begrenzt die Anzahl der zu aktualisierenden Datensätze.

Beispiel

```
UPDATE KUNDE SET Telefon='1234' WHERE ID=2;
```

Aufgaben: **f_SQL_INUP**

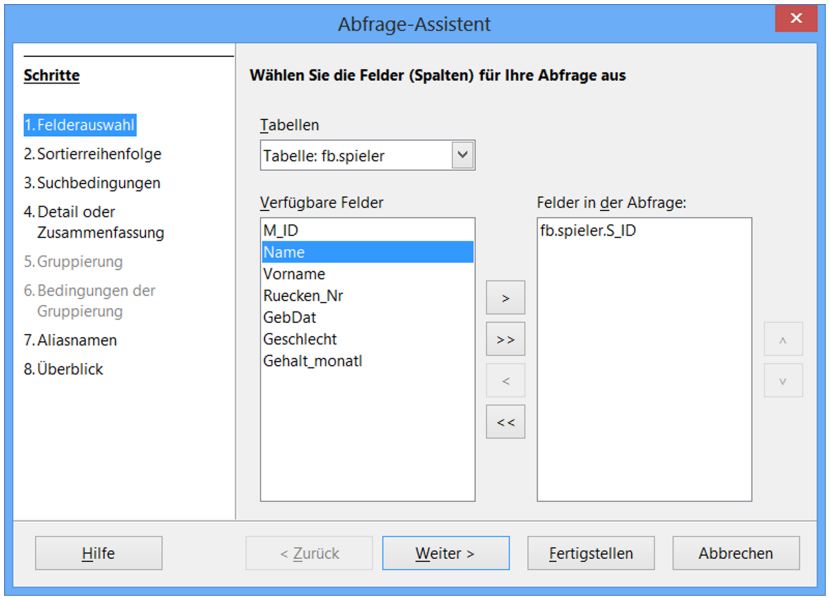
1. Ändern Sie die Beschreibung im Datensatz `L_ID=6` der Tabelle `LIGA` in „Landesliga Bayern B-Jugend weiblich“.
2. Fügen Sie in den Datensatz des Spielers Cacau den Vornamen „Max“ ein!
3. Ändern Sie die Beschreibung der Spalte Geschlecht in der Tabelle Spieler wie folgt: männlich->m, weiblich->w!
4. Verdoppeln Sie das Gehalt der Spieler mit der `S_ID` 17 und 39!

5a. Geben Sie das Gehalt der 5 Spieler mit dem niedrigsten Gehalt aus (Name, Vorname, Gehalt_monatl)!

5b. Erhöhen Sie das Gehalt der 5 Spieler mit dem niedrigsten Gehalt um 15%! Überprüfen Sie so dann das Resultat.

5 Abfragen

Abfragen stellen den größten und umfangreichsten Bereich dar, den SQL bietet. Das `SELECT`-Statement wird am häufigsten gebraucht und bietet im Vergleich zu anderen Statements die meisten Möglichkeiten.



Inhalt

5.1 SQL - Abfrage von Daten 58
5.1.1 Verknüpfungen von Tabellen 59
5.1.2 Funktion 67
5.1.3 Gruppierung 86
5.1.4 Subselect 89
5.1.5 Gebräuchliche Abfragen 92
5.2 Abfrageerstellung mit der grafischen Benutzeroberfläche 94
5.2.1 Funktionen in der Abfrage 97

5.1 SQL - Abfrage von Daten

Mit dem Befehl `SELECT` werden Tabellen abgefragt. Dieser ist der mächtigste und umfangreichste Befehl, den SQL zu bieten hat.

Die vereinfachte Ausführung sieht dabei wie folgt aus:

```
SELECT select_expr, ...
   [FROM table_references
   [WHERE where_condition]
   [GROUP BY {col_name | expr | position}
   [HAVING where_condition]
   [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
   [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

FROM Legt die Tabellen fest, die in die Abfrage einbezogen werden.

WHERE Legt die Bedingung für die Abfrage fest.

GROUP BY Gruppiert die Abfrage.

HAVING Definiert eine Bedingung, die für `GROUP BY` gilt.

ORDER BY Legt die Sortierreihenfolge fest.

LIMIT Legt fest, wie viel Zeilen ausgegeben werden.

Den kompletten Befehlsumfang der `SELECT`-Anweisung finden Sie im MySQL-Handbuch.

Beispiel

```
SELECT * FROM KUNDE ORDER BY Name, Vorname LIMIT 3
```

```
+----+--------+----------+------------+-------+------------+---------+
| ID | Name   | Vorname  | Strasse    | PLZ   | Ort        | Telefon |
+----+--------+----------+------------+-------+------------+---------+
| 2  | Huber  | Peter    | Teststr. 5 | 12345 | Testhausen | 4711    |
| 5  | Hubert | Natascha | Ilgstr. 4  | 1122  | Xhausen    | 123456  |
| 3  | Meier  | Petra    | Teststr. 6 | 12345 | Testhausen | 4711    |
+----+--------+----------+------------+-------+------------+---------+
```

Reihenfolge

Beim `SELECT`-Befehl ist wichtig zu wissen, in welcher Reihenfolge die einzelnen Bereiche abgearbeitet werden. Im ersten Schritt wird der gesamte Ausdruck von der SQL-Engine eingelesen. Danach ist die Reihenfolge wie folgt:

1. `FROM`
   Zuerst werden die Tabellen berücksichtigt. Ein dort definierter Alias ist in allen weiteren Bereichen gültig. Die Tabelle, die die meisten Datensätze enthält, sollte an erster Stelle stehen.
2. `WHERE`
   Danach wird die Datenmenge eingeschränkt. Sind dort mehrere Bedingungen enthalten, sollte die Bedingung an erster Stelle stehen, die am meisten Daten ausfiltert. Aliase aus `FROM` können hier verwendet werden.
3. `GROUP BY`
   Nun folgt die Gruppierung (wenn vorhanden).
4. `HAVING`
   Für die Gruppierung kann zusätzlich eine Bedingung gesetzt werden.
5. `Spalten`
   Jetzt werden die Spalten abgearbeitet. Ein dort definierter Alias kann erst danach verwendet werden.
6. `ORDER BY`
   Nun kann eine Sortierung der Spalten erfolgen. Dabei kann der Spaltenname, der Alias oder die Spaltennummer verwendet werden.
7. `LIMIT`
   Damit kann zuletzt die Ausgabe eingeschränkt werden.

5.1.1 Verknüpfungen von Tabellen

Es gibt mehrere Möglichkeiten, zwei oder mehrere Tabellen miteinander zu verbinden. Dies wird in SQL als `JOIN` bezeichnet, allgemein als Verbund.

Als Beispiel sollen nun folgende zwei Tabellen dienen.

```
DROP TABLE IF EXISTS TMITGLIED;

CREATE TABLE TMITGLIED (
  ID           INTEGER       NOT NULL,
  Vorname      VARCHAR(20),
  Nachname     VARCHAR(20),
  Beitrag_ID   CHAR(1),
  Betreuer     INTEGER,
  PRIMARY KEY (ID)
);

DROP TABLE IF EXISTS TBEITRAG;

CREATE TABLE TBEITRAG (
  ID           CHAR(1)       NOT NULL,
  Bezeichnung  VARCHAR(20),
  Beitrag      NUMERIC(6,2),
  PRIMARY KEY (ID)
);


INSERT INTO TBEITRAG VALUES ('M','Familienmitglied',0.00);
INSERT INTO TBEITRAG VALUES ('F','Familienbeitrag',200.00);
INSERT INTO TBEITRAG VALUES ('E','Erwachsener',100.00);
INSERT INTO TBEITRAG VALUES ('K','Kind␣/␣Student',60.00);

INSERT INTO TMITGLIED VALUES (1,'Hans','Huber', 'F', 0);
INSERT INTO TMITGLIED VALUES (2,'Beate','Huber', 'M', 1);
INSERT INTO TMITGLIED VALUES (3,'Peter','Maier', 'E', 1);
INSERT INTO TMITGLIED VALUES (4,'Pumuckl','Klein',null, 0);
INSERT INTO TMITGLIED VALUES (5,'Susi','Schlau',null, 0);
```

```
DESC TBEITRAG;
```

```
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| ID          | char(1)      | NO   | PRI | null    |       |
| Bezeichnung | varchar(20)  | YES  |     | null    |       |
| Beitrag     | decimal(6,2) | YES  |     | null    |       |
+-------------+--------------+------+-----+---------+-------+
```

```
DESC TMITGLIED;
```

```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| ID         | int(11)     | NO   | PRI | null    |       |
| Vorname    | varchar(20) | YES  |     | null    |       |
| Nachname   | varchar(20) | YES  |     | null    |       |
| Beitrag_ID | char(1)     | YES  |     | null    |       |
| Betreuer   | int(11)     | YES  |     | null    |       |
+------------+-------------+------+-----+---------+-------+
```

Übersicht

INNER JOIN Ist eine Verknüpfung „innerhalb“ zweier Tabellen, d. h. der Wert der Verknüpfung muss in beiden Tabellen vorhanden sein.

OUTER JOIN Ist eine Verknüpfung, bei der auch Datensätze geliefert werden, die die Auswahlbedingung nicht erfüllen. Das Wort „OUTER“ kann auch entfallen. Dazu zählen:

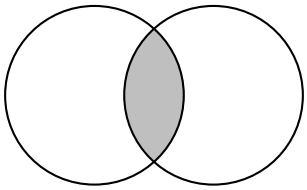
- **LEFT JOIN** Alle Werte aus der linken Tabelle, verknüpft mit der rechten Tabelle. Fehlende Verknüpfungen werden mit „null“ aufgefüllt.
- **RIGHT JOIN** Alle Werte aus der rechten Tabelle, verknüpft mit der linken Tabelle. Fehlende Verknüpfungen werden mit „null“ aufgefüllt.
- **FULL JOIN** Liefert alle Datensätze beider Tabellen. Fehlende Verknüpfungen werden mit „null“ aufgefüllt.

SELF JOIN Eine Tabelle wird mit sich selbst verknüpft.

INNER JOIN

Ein `INNER JOIN` (auch natürlicher Verbund genannt – Equivalent Join), verknüpft zwei Tabellen über ein gemeinsames Attribut. Es werden jeweils Zeilen mit gleichen Attributwerten zu einer neuen verbunden.

Betrachtet man dies mit den Möglichkeiten der Mengenlehre, so wird die Schnittmenge gebildet.



Folgender Befehl verknüpft die beiden Tabellen bzgl. der `Beitrag_ID`. Damit bei gleichen Spaltennamen in den zwei Tabellen nicht immer der komplette Tabellenname angegeben werden muss, wird jeweils ein Alias für die Tabelle definiert.

```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m INNER JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 3  | Maier    | Peter   | E    | Erwachsener      |
+----+----------+---------+------+------------------+
```

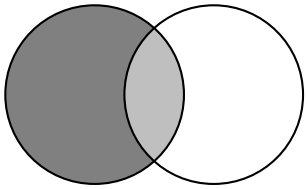
Der `INNER JOIN` lässt sich auch mit einer `WHERE`-Bedingung realisieren.

```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m, TBEITRAG b
   WHERE m.Beitrag_ID = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 3  | Maier    | Peter   | E    | Erwachsener      |
+----+----------+---------+------+------------------+
```

LEFT JOIN

Beim `LEFT JOIN` (auch `LEFT OUTER JOIN`) werden alle Zeilen der „linken“ Tabelle in die Ergebnistabelle übernommen, auch wenn es zu ihnen keine entsprechenden Zeilen in der „rechten“ Tabelle gibt. Fehlende Werte werden mit `NULL` belegt.

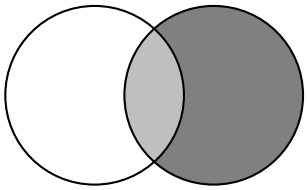


```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 3  | Maier    | Peter   | E    | Erwachsener      |
| 4  | Klein    | Pumuckl | null | null             |
| 5  | Schlau   | Susi    | null | null             |
+----+----------+---------+------+------------------+
```

RIGHT JOIN

Beim `RIGHT JOIN` (auch `RIGHT OUTER JOIN`) werden alle Zeilen der „rechten“ Tabelle in die Ergebnistabelle übernommen, auch wenn es zu ihnen keine entsprechenden Zeilen in der „linken“ Tabelle gibt. Fehlende Werte werden mit `NULL` belegt. Ein `RIGHT JOIN` kann auch mit einem `LEFT JOIN` realisiert werden, indem man die beiden Tabellen vertauscht.

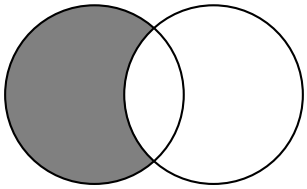


```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m RIGHT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| nu | null     | null    | K    | Kind / Student   |
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 3  | Maier    | Peter   | E    | Erwachsener      |
+----+----------+---------+------+------------------+
```

LEFT JOIN mit Bedingung

Damit man nun mit dem `LEFT JOIN` alle die Datensätze bekommt, die **nicht** der Verknüpfung entsprechen (also kein entsprechender Eintrag vorhanden ist), wird zusätzlich mit der `WHERE`-Bedingung auf `NULL` geprüft.



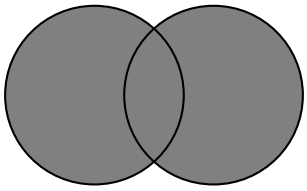
```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   WHERE b.ID IS NULL
   ORDER BY m.ID;
```

```
+----+----------+---------+------+-------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung |
+----+----------+---------+------+-------------+
| 4  | Klein    | Pumuckl | null | null        |
| 5  | Schlau   | Susi    | null | null        |
+----+----------+---------+------+-------------+
```

FULL JOIN

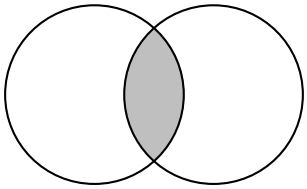
Beim `FULL JOIN` (auch `FULL OUTER JOIN`) werden alle Datensätze beider Tabellen verknüpft. Fehlende Verknüpfungen werden mit „null“ aufgefüllt. Ein `FULL JOIN` kann auch durch eine Kombination von `LEFT JOIN` und `RIGHT JOIN` mittels `UNION` erreicht werden.

MySQL unterstützt diesen Typ nicht; er wird auch in der Praxis nur sehr selten gebraucht.



LEFT JOIN mit IS NOT NULL

Einen `INNER JOIN` kann man auch mit einem `LEFT JOIN` mit `IS NOT NULL` ausdrücken.



```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   WHERE b.ID IS NOT NULL
   ORDER BY m.ID;
```

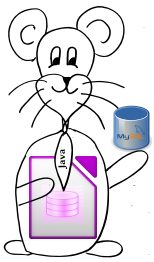
```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 3  | Maier    | Peter   | E    | Erwachsener      |
+----+----------+---------+------+------------------+
```

CROSS JOIN

Bei einem `CROSS JOIN` werden alle Zeilen der „linken“ Tabelle mit allen Zeilen der „rechten“ Tabelle verknüpft (kartesisches Produkt). Der `CROSS JOIN` sollte aus Performance- und Speicherplatzgründen unbedingt vermieden werden, wenn keine weitere Bedingung verwendet wird.

```
SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
   FROM TMITGLIED m, TBEITRAG b
   ORDER BY m.ID;
```

```
+----+----------+---------+------+------------------+
| ID | Nachname | Vorname | B-ID | Bezeichnung      |
+----+----------+---------+------+------------------+
| 1  | Huber    | Hans    | M    | Familienmitglied |
| 1  | Huber    | Hans    | K    | Kind / Student   |
| 1  | Huber    | Hans    | F    | Familienbeitrag  |
| 1  | Huber    | Hans    | E    | Erwachsener      |
| 2  | Huber    | Beate   | M    | Familienmitglied |
| 2  | Huber    | Beate   | K    | Kind / Student   |
| 2  | Huber    | Beate   | F    | Familienbeitrag  |
| 2  | Huber    | Beate   | E    | Erwachsener      |
| 3  | Maier    | Peter   | M    | Familienmitglied |
| 3  | Maier    | Peter   | K    | Kind / Student   |
| 3  | Maier    | Peter   | F    | Familienbeitrag  |
| 3  | Maier    | Peter   | E    | Erwachsener      |
| 4  | Klein    | Pumuckl | M    | Familienmitglied |
| 4  | Klein    | Pumuckl | K    | Kind / Student   |
| 4  | Klein    | Pumuckl | F    | Familienbeitrag  |
| 4  | Klein    | Pumuckl | E    | Erwachsener      |
| 5  | Schlau   | Susi    | M    | Familienmitglied |
| 5  | Schlau   | Susi    | K    | Kind / Student   |
| 5  | Schlau   | Susi    | F    | Familienbeitrag  |
| 5  | Schlau   | Susi    | E    | Erwachsener      |
+----+----------+---------+------+------------------+
```



Merke

Werden n Tabellen bei der Abfrage in der `FROM`-Bedingung verwendet, so müssen mindestens $(n-1)$ Bedingungen bei der `WHERE`-Bedingung verwendet werden. Diese werden mit `AND` verknüpft.
Ansonsten ist die Wahrscheinlichkeit sehr hoch, dass das Ergebnis nicht dem entspricht, was man erwartet!

SELF JOIN

Oft werden in einem Datensatz Informationen gespeichert, die einen anderen Datensatz der selben Tabelle betreffen. In der Tabelle `TMITGLIED` gibt es die Spalte `Betreuer`. Der Betreuer eines Mitglieds ist selbst wiederum auch Mitglied.

Mit nachfolgendem Befehl werden die Mitglieder, die einen Betreuer haben, ausgegeben.

```
SELECT m.ID,m.Nachname,m.Vorname, CONCAT(b.Nachname, ',␣' , b.Vorname) AS 'Betreuer'
   FROM TMITGLIED m, TMITGLIED b
   WHERE m.Betreuer = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+-------------+
| ID | Nachname | Vorname | Betreuer    |
+----+----------+---------+-------------+
| 2  | Huber    | Beate   | Huber, Hans |
| 3  | Maier    | Peter   | Huber, Hans |
+----+----------+---------+-------------+
```

Verwendet man einen `LEFT JOIN`, so werden alle Mitglieder ausgegeben.

```
SELECT m.ID,m.Nachname,m.Vorname, CONCAT(b.Nachname, ',␣' , b.Vorname) AS 'Betreuer'
   FROM TMITGLIED m LEFT JOIN TMITGLIED b ON m.Betreuer = b.ID
   ORDER BY m.ID;
```

```
+----+----------+---------+-------------+
| ID | Nachname | Vorname | Betreuer    |
+----+----------+---------+-------------+
| 1  | Huber    | Hans    | null        |
| 2  | Huber    | Beate   | Huber, Hans |
| 3  | Maier    | Peter   | Huber, Hans |
| 4  | Klein    | Pumuckl | null        |
| 5  | Schlau   | Susi    | null        |
+----+----------+---------+-------------+
```

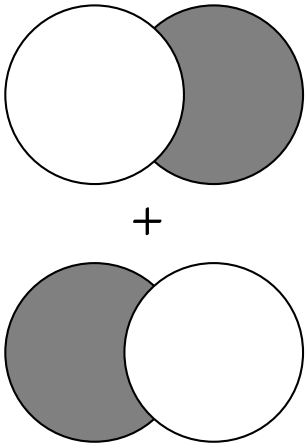
Eine beliebte Frage ist auch: Gib alle Mitglieder aus, die keinen Betreuer haben!

```
SELECT m.ID,m.Nachname,m.Vorname, CONCAT(b.Nachname, ',␣' , b.Vorname) AS 'Betreuer'
   FROM TMITGLIED m LEFT JOIN TMITGLIED b ON m.Betreuer = b.ID
   WHERE b.ID IS NULL
   ORDER BY m.ID;
```

```
+----+----------+---------+----------+
| ID | Nachname | Vorname | Betreuer |
+----+----------+---------+----------+
| 1  | Huber    | Hans    | null     |
| 4  | Klein    | Pumuckl | null     |
| 5  | Schlau   | Susi    | null     |
+----+----------+---------+----------+
```

UNION

Mit dem Schlüsselwort `UNION` kann man zwei oder mehr Abfragen vereinigen. Wichtig dabei ist, dass die Spaltenzahl bei beiden Abfragen identisch ist.



Im nachfolgenden Beispiel werden alle Mitglieder ausgegeben, die entweder 'F' oder 'M' als Beitrags-ID haben.

```
SELECT m.ID,m.Nachname,m.Vorname, m.Beitrag_ID
    FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
    WHERE m.Beitrag_ID = 'F'
UNION SELECT m.ID,m.Nachname,m.Vorname, m.Beitrag_ID
    FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
    WHERE m.Beitrag_ID = 'M';
```

```
+----+----------+---------+------------+
| ID | Nachname | Vorname | Beitrag_ID |
+----+----------+---------+------------+
| 1  | Huber    | Hans    | F          |
| 2  | Huber    | Beate   | M          |
+----+----------+---------+------------+
```

In vielen Fällen kann man aber auch die Abfrage mit einem `OR` in der `WHERE`-Bedingung realisieren.

```
SELECT m.ID,m.Nachname,m.Vorname, m.Beitrag_ID
    FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
    WHERE m.Beitrag_ID = 'F' OR m.Beitrag_ID = 'M';
```

```
+----+----------+---------+------------+
| ID | Nachname | Vorname | Beitrag_ID |
+----+----------+---------+------------+
| 1  | Huber    | Hans    | F          |
| 2  | Huber    | Beate   | M          |
+----+----------+---------+------------+
```

Aufgaben: **f_SQL_JOIN**

1. Was versteht man unter einem `INNER JOIN`?
   - ☐ Eine Verknüpfung, bei der eine Tabelle mit sich selbst (innere Verknüpfung) verknüpft wird.
   - ☐ Ein natürlicher Verbund zweier Tabellen.
   - ☐ Eine Verknüpfung über ein gemeinsames Attribut, die erfüllt sein muss.
   - ☐ Eine Schnittmenge der inneren Gruppe.
2. Wie kann nachfolgender SQL-Ausdruck noch dargestellt werden?

   ```
   SELECT m.ID,m.Nachname,m.Vorname,b.ID,b.Bezeichnung
   FROM TMITGLIED m INNER JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
   ORDER BY m.ID;
   ```
3. Was unterscheidet einen `LEFT JOIN` von einem `INNER JOIN`?
   - ☐ Nichts, `LEFT JOIN` ist die neue Schreibweise für `INNER JOIN` seit SQL-99.
   - ☐ Es werden alle Elemente der linken Tabellen verwendet.
   - ☐ Gibt es nicht, es heißt `LEFT OUTER JOIN`.
   - ☐ Die linke Tabelle wird mit geringer Priorität behandelt.
   - ☐ Aus der linken Tabelle werden die Elemente der Verknüpfung genommen.
4. Wie kann man aus einem `RIGHT JOIN` einen `LEFT JOIN` machen?
   - ☐ Geht nicht!
   - ☐ Es werden die beiden Tabellen vertauscht.
   - ☐ Der Ausdruck bei `ON` wird negiert.
   - ☐ Es wird `NOT ON` anstelle von `ON` verwendet.
   - ☐ Es wird ein Ausrufezeichen (Negation) vor `!RIGHT JOIN` geschrieben.
5. Was ist am nachfolgenden Ausdruck falsch?

   ```
   SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
      FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
      WHERE b.ID=NULL
      ORDER BY m.ID;
   ```
6. Wie kann man nachfolgenden Ausdruck noch schreiben?

   ```
   SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
      FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
      WHERE b.ID IS NOT NULL
      ORDER BY m.ID;
   ```
7. Verwendet man bei einem `INNER JOIN` (mit `WHERE`-Bedingung) n-Tabellen, wie viele Verknüpfungen bei `WHERE` werden mindestens benötigt?

5.1.2 Funktion

MySQL stellt viele Funktionen und Operatoren zur Verfügung. Nachfolgend eine Auswahl der wichtigsten Funktionen.

In MySQL lassen sich Funktionen direkt im `select`-Befehl nutzen, ohne dabei eine Tabelle anzugeben. Andere DBMS können dies nicht und stellen dafür die Tabelle `DUAL` zur Verfügung. MySQL unterstützt dies auch.

```
SELECT 1+1;
```

```
+-----+
| 1+1 |
+-----+
| 2   |
+-----+
```

```
SELECT 1+1 FROM DUAL;
```

```
+-----+
| 1+1 |
+-----+
| 2   |
+-----+
```

Vergleichsoperatoren

Hierbei gilt: 1 = `TRUE`, 0 = `FALSE`, `NULL` = `FALSE`

`=`	Gleichheit
`<=>`	`NULL`-sicheres Gleich. Dieser Operator führt ebenso wie '=' eine Vergleichsoperation durch, gibt aber 1 statt `NULL` zurück, wenn beide Operanden `NULL` sind, und 0 statt `NULL`, wenn einer der Operanden `NULL` ist.
`<>, !=`	Ungleich
`<=`	Kleiner oder gleich
`<`	Kleiner als
`>=`	Größer oder gleich
`>`	Größer als
`IS bool`	bzw. `IS NOT bool` Vergleicht einen Wert mit einem booleschen Wert, wobei `bool` `TRUE`, `FALSE` oder `UNKNOWN` sein kann.
`IS NULL`	bzw. `IS NOT NULL` Prüft, ob ein Wert `NULL` ist oder nicht.
`expr BETWEEN min AND max`	Wenn `expr` größer oder gleich `min` und `expr` kleiner oder gleich `max` ist, gibt `BETWEEN` 1 zurück, andernfalls 0 ($min \leq expr$ AND $expr \leq max$)
`expr NOT BETWEEN min AND max`	entspricht `NOT (expr BETWEEN min AND max)`
`COALESCE(value,...)`	Gibt den ersten Nicht-`NULL`-Wert in der Liste oder aber `NULL` zurück, sofern keine Nicht-`NULL`-Werte vorhanden sind.
`GREATEST(v1,v2,...)`	Gibt den höchsten Wert zurück.
`expr IN (value,...)`	Gibt 1 zurück, wenn `expr` gleich einem der Werte in der `IN`-Liste ist, andernfalls wird 0 zurückgegeben.
`expr NOT IN (value,...)`	entspricht `NOT (expr IN (value,...))`
`ISNULL(expr)`	Wenn `expr` `NULL` ist, gibt `ISNULL()` 1 zurück, andernfalls 0.
`INTERVAL(N,N1,N2,N3,...)`	Gibt 0 zurück, wenn `N` < `N1` ist, 1, wenn `N` < `N2` usw., oder -1, wenn `N` `NULL` ist.
`LEAST(v1,v2,...)`	Gibt den kleinsten Wert zurück.

Beispiele

```
SELECT 5 IN (1, 2, 3, 4, 5, 6);
```

```
+-------------------------+
| 5 IN (1, 2, 3, 4, 5, 6) |
+-------------------------+
| 1                       |
+-------------------------+
```

```
SELECT GREATEST(1, 2, 3, 4, 5, 6);
```

```
+----------------------------+
| GREATEST(1, 2, 3, 4, 5, 6) |
+----------------------------+
| 6                          |
+----------------------------+
```

```
SELECT LEAST(1, 2, 3, 4, 5, 6);
```

```
+-------------------------+
| LEAST(1, 2, 3, 4, 5, 6) |
+-------------------------+
| 1                       |
+-------------------------+
```

Logische Operatoren

`NOT, !`	Nicht
`AND, &&`	Und
`OR, ||`	Oder
`XOR`	Exklusives Oder

```
SELECT 0 AND 1;
```

```
+---------+
| 0 AND 1 |
+---------+
| 0       |
+---------+
```

```
SELECT 0 OR 1;
```

```
+--------+
| 0 OR 1 |
+--------+
| 1      |
+--------+
```

Aufgaben: **f_SQL_F01**

Welche Ergebnisse liefern folgende SQL-Statements?

1. `SELECT 1 = 2;`
2. `SELECT 1 = 1;`
3. `SELECT 1 <= 2;`
4. `SELECT 4 BETWEEN 0 AND 10;`
5. `SELECT GREATEST(1,-1,5,1,0,9,8);`
6. `SELECT 7 IN (1,-1,5,1,0,9,8);`
7. `SELECT LEAST(1,-1,5,1,0,9,8);`
8. `SELECT 0 AND 1;`
9. `SELECT 1 AND 1;`
10. `SELECT 0 OR 1;`
11. `SELECT 1 OR 1;`
12. `SELECT (0 AND 1) OR (1 AND 1) OR (1 OR 1);`

Ablaufsteuerungsfunktionen

(switch) case in SQL

```
CASE value
   WHEN [compare_value] THEN result
   [WHEN [compare_value] THEN result ...]
   [ELSE result]
   END
CASE
   WHEN [condition] THEN result
   [WHEN [condition] THEN result ...]
   [ELSE result]
   END
```

Mit der `CASE`-Anweisung lässt sich ein switch-case wie in Java realisieren.

```
SELECT
   CASE 2*3
   WHEN 4   THEN 'vier'
   WHEN 5   THEN 'fünf'
   WHEN 6   THEN 'sechs'
   WHEN 7   THEN 'sieben'
   ELSE          'was␣anderes'
   END as 'Ergebnis';
```

```
+----------+
| Ergebnis |
+----------+
| sechs    |
+----------+
```

if then else in SQL

```
IF(expr1,expr2,expr3)
```

Wenn `expr1` `TRUE` ist (`expr1<> 0` und `expr1 <> NULL`), dann gibt `IF()` `expr2` zurück, andernfalls gibt es `expr3` zurück.

```
SELECT IF(5>2,'größer','kleiner')
```

```
+----------------------------+
| IF(5>2,'größer','kleiner') |
+----------------------------+
| größer                     |
+----------------------------+
```

Prüfung auf NULL

```
IFNULL(expr1,expr2)
NULLIF(expr1,expr2)
```

Wenn `expr1` nicht `NULL` ist, gibt `IFNULL()` `expr1` zurück; andernfalls wird `expr2` zurückgegeben. `NULLIF()` gibt `NULL` zurück, wenn `expr1 = expr2` wahr ist; andernfalls wird `expr1` zurückgegeben.

```
SELECT IFNULL(null, 'Kein␣Eintrag␣gefunden!')
```

```
+----------------------------------------+
| IFNULL(null, 'Kein␣Eintrag␣gefunden!') |
+----------------------------------------+
| Kein Eintrag gefunden!                 |
+----------------------------------------+
```

```
SELECT IFNULL('Peter', 'Kein␣Eintrag␣gefunden!')
```

```
+-------------------------------------------+
| IFNULL('Peter', 'Kein␣Eintrag␣gefunden!') |
+-------------------------------------------+
| Peter                                     |
+-------------------------------------------+
```

String-Funktionen

Funktionen zum Bearbeiten von Zeichenketten stellen eine wichtige Funktion dar, da dies ermöglicht, mit SQL umfangreiche Ausgaben zu erhalten.

ASCII(str)
Gibt den ASCII-Wert des ersten Zeichens zurück.

```
SELECT ASCII('Peter')
```

```
+----------------+
| ASCII('Peter') |
+----------------+
| 80             |
+----------------+
```

BIN(N)
Gibt die Zahl N als Binärwert zurück.

```
SELECT BIN(257)
```

```
+-----------+
| BIN(257)  |
+-----------+
| 100000001 |
+-----------+
```

BIT_LENGTH(str)
Gibt die Länge der Zeichenkette in Bit zurück.

```
SELECT BIT_LENGTH('Uschi')
```

```
+---------------------+
| BIT_LENGTH('Uschi') |
+---------------------+
| 40                  |
+---------------------+
```

CHAR(N,... [USING charset_name])
Jede Zahl wird als Zeichen interpretiert und es wird die komplette Zeichenkette zurückgeliefert.

```
SELECT CHAR(0x41, 0x46, 0x46, 0x45 USING ASCII)
```

```
+-----------------------------------------+
| CHAR(0x41, 0x46, 0x46, 0x45 USING ASCII) |
+-----------------------------------------+
| AFFE                                    |
+-----------------------------------------+
```

CHAR_LENGTH(str)
Gibt die Länge der Zeichenkette unter Berücksichtigung von Multibytezeichen zurück (alternativ `CHARACTER_LENGTH(str)`).

```
SELECT CHAR_LENGTH('Uschi')
```

```
+----------------------+
| CHAR_LENGTH('Uschi') |
+----------------------+
| 5                    |
+----------------------+
```

LENGTH(str)
Gibt die Länge des Strings in Byte zurück.

```
SELECT LENGTH('Das␣Haus␣ist␣rot!')
```

```
+----------------------------+
| LENGTH('Das␣Haus␣ist␣rot!') |
+----------------------------+
| 17                         |
+----------------------------+
```

CONCAT(str1,str2,...)
Verkettet die einzelnen Zeichenketten zu einer Zeichenkette.

```
SELECT CONCAT('Uschi',',␣','Meier')
```

```
+-----------------------------+
| CONCAT('Uschi',',␣','Meier') |
+-----------------------------+
| Uschi, Meier                |
+-----------------------------+
```

CONCAT_WS(separator, str1,str2,...)
Wie `CONCAT()`, jedoch wird ein Separatorzeichen eingefügt.

```
SELECT CONCAT_WS(',␣', 'Hans','Meier')
```

```
+---------------------------------+
| CONCAT_WS(',␣', 'Hans','Meier') |
+---------------------------------+
| Hans, Meier                     |
+---------------------------------+
```

CONV(N,from_base,to_base)
Wandelt eine Zahl zwischen verschiedenen Zahlenbasen um.

```
SELECT CONV('B', 16, 10)
```

```
+-------------------+
| CONV('B', 16, 10) |
+-------------------+
| 11                |
+-------------------+
```

ELT(N,str1,str2,str3,...)
Gibt `str1` zurück, wenn `N = 1` ist, `str2`, wenn `N = 2` usw.

```
SELECT ELT(2, 'eins', 'zwei', 'drei', 'vier')
```

```
+----------------------------------------+
| ELT(2, 'eins', 'zwei', 'drei', 'vier') |
+----------------------------------------+
| zwei                                   |
+----------------------------------------+
```

FIELD(str,str1,str2,str3,...)
Gibt die Position von `str` in der Liste zurück oder 0, wenn nichts gefunden wurde.

```
SELECT FIELD('drei', 'eins', 'zwei', 'drei', 'vier')
```

```
+-----------------------------------------------+
| FIELD('drei', 'eins', 'zwei', 'drei', 'vier') |
+-----------------------------------------------+
| 3                                             |
+-----------------------------------------------+
```

FORMAT(X,D)
Formatiert die Zahl X im Format `#,###,###.##` und rundet auf D Dezimalstellen.

```
SELECT FORMAT(12345.6789,2)
```

```
+----------------------+
| FORMAT(12345.6789,2) |
+----------------------+
| 12,345.68            |
+----------------------+
```

HEX(N_or_S)
Wenn N_or_S eine Zahl ist, wird eine String-Darstellung des Hexadezimalwerts von N zurückgegeben (siehe CONV(N,10,16)). Wenn N_or_S ein String ist, wird jeder Buchstabe in seinen Hexwert umgewandelt und ausgegeben. Das Gegenstück dazu ist UNHEX().

```
SELECT HEX('AFFE')
```

```
+-------------+
| HEX('AFFE') |
+-------------+
| 41464645    |
+-------------+
```

INSERT(str,pos,len,newstr)
Erzeugt einen neuen String, wobei der Teilbereich zwischen pos und len durch newstr ersetzt wird.

```
SELECT INSERT('Das␣Haus␣ist␣rot!',5,4,'Automobil')
```

```
+---------------------------------------------+
| INSERT('Das␣Haus␣ist␣rot!',5,4,'Automobil') |
+---------------------------------------------+
| Das Automobil ist rot!                      |
+---------------------------------------------+
```

INSTR(str,substr)
Gibt die Position von substr in str zurück.

```
SELECT INSTR('Das␣Haus␣ist␣rot!','Haus')
```

```
+-----------------------------------+
| INSTR('Das␣Haus␣ist␣rot!','Haus') |
+-----------------------------------+
| 5                                 |
+-----------------------------------+
```

LOCATE(substr,str)
Wie INSTR(), nur die Reihenfolge der Parameter ist anders.

```
SELECT LOCATE('Haus', 'Das␣Haus␣ist␣rot!')
```

```
+-------------------------------------+
| LOCATE('Haus', 'Das␣Haus␣ist␣rot!') |
+-------------------------------------+
| 5                                   |
+-------------------------------------+
```

LCASE(str)
Wandelt Groß- in Kleinbuchstaben um (alternativ LOWER()).

```
SELECT LOWER('Das␣Haus␣ist␣rot!')
```

```
+----------------------------+
| LOWER('Das␣Haus␣ist␣rot!') |
+----------------------------+
| das haus ist rot!          |
+----------------------------+
```

UCASE(str)
Wandelt Klein- in Großbuchstaben um (alternativ UPPER()).

```
SELECT UPPER('Das␣Haus␣ist␣rot!')
```

```
+----------------------------+
| UPPER('Das␣Haus␣ist␣rot!') |
+----------------------------+
| DAS HAUS IST ROT!          |
+----------------------------+
```

LPAD(str,len,padstr)
Füllt den String von links mit `padstr` auf, bis die Länge `len` erreicht ist.

```
SELECT LPAD('Haus',15,'.')
```

```
+---------------------+
| LPAD('Haus',15,'.') |
+---------------------+
| ...........Haus     |
+---------------------+
```

RPAD(str,len,padstr)
Füllt den String von rechts mit `padstr` auf, bis die Länge `len` erreicht ist.

```
SELECT RPAD('Haus',15,'.')
```

```
+---------------------+
| RPAD('Haus',15,'.') |
+---------------------+
| Haus...........     |
+---------------------+
```

LTRIM(str)
Entfernt Leerzeichen auf der linken Seite.

```
SELECT LTRIM('␣␣␣Haus␣␣␣')
```

```
+---------------------+
| LTRIM('␣␣␣Haus␣␣␣') |
+---------------------+
| Haus                |
+---------------------+
```

RTRIM(str)
Entfernt Leerzeichen auf der rechten Seite.

```
SELECT RTRIM('␣␣␣Haus␣␣␣')
```

```
+---------------------+
| RTRIM('␣␣␣Haus␣␣␣') |
+---------------------+
|    Haus             |
+---------------------+
```

TRIM([{BOTH | LEADING | TRAILING} [remstr] FROM] str)
Gibt einen String zurück, bei dem alle `remstr`-Präfixe oder -Suffixe entfernt wurden.

```
SELECT CONCAT('<',TRIM(BOTH '␣' FROM '␣␣␣Haus␣␣␣'),'>')
```

```
+---------------------------------------------------+
| CONCAT('<',TRIM(BOTH '␣' FROM '␣␣␣Haus␣␣␣'),'>') |
+---------------------------------------------------+
| <Haus>                                            |
+---------------------------------------------------+
```

QUOTE(str)
Setzt einen String in Anführungszeichen.

```
SELECT QUOTE('␣␣␣Haus␣␣␣')
```

```
+---------------------+
| QUOTE('␣␣␣Haus␣␣␣') |
+---------------------+
| '␣␣␣Haus␣␣␣'        |
+---------------------+
```

REPEAT(str,count)
Erzeugt einen String aus `count` Zeichen.

```
SELECT REPEAT('<->',10)
```

```
+--------------------------------+
| REPEAT('<->',10)               |
+--------------------------------+
| <-><-><-><-><-><-><-><-><-><-> |
+--------------------------------+
```

SPACE(N)
Erzeugt eine Zeichenkette mit N Leerzeichen.

```
SELECT CONCAT('<',SPACE(10),'>')
```

```
+---------------------------+
| CONCAT('<',SPACE(10),'>') |
+---------------------------+
| <          >              |
+---------------------------+
```

REPLACE(str,from_str,to_str)
Ersetzt Zeichenfolgen durch andere.

```
SELECT REPLACE('Das␣Haus␣ist␣rot!','Haus','Automobil')
```

```
+--------------------------------------------------+
| REPLACE('Das␣Haus␣ist␣rot!','Haus','Automobil') |
+--------------------------------------------------+
| Das Automobil ist rot!                           |
+--------------------------------------------------+
```

REVERSE(str)
Dreht die Reihenfolge der Zeichen um.

```
SELECT REVERSE('Das␣Haus␣ist␣rot!')
```

```
+------------------------------+
| REVERSE('Das␣Haus␣ist␣rot!') |
+------------------------------+
| !tor tsi suaH saD            |
+------------------------------+
```

RIGHT(str,len)
Gibt den rechten Teil der Zeichenkette zurück.

```
SELECT RIGHT('Das␣Haus␣ist␣rot!',3)
```

```
+------------------------------+
| RIGHT('Das␣Haus␣ist␣rot!',3) |
+------------------------------+
| ot!                          |
+------------------------------+
```

LEFT(str,len)
Gibt den linken Teil der Zeichenkette zurück.

```
SELECT LEFT('Das␣Haus␣ist␣rot!',4)
```

```
+-----------------------------+
| LEFT('Das␣Haus␣ist␣rot!',4) |
+-----------------------------+
| Das                         |
+-----------------------------+
```

SUBSTRING(str,pos)
Gibt einen Teilstring zurück (alternativ `SUBSTRING(str FROM pos)`, `SUBSTRING(str,pos,len)`, `SUBSTRING(str FROM pos FOR len)`, `MID(str,pos,len)`).

```
SELECT SUBSTR('Das␣Haus␣ist␣rot!',5,4)
```

```
+---------------------------------+
| SUBSTR('Das␣Haus␣ist␣rot!',5,4) |
+---------------------------------+
| Haus                            |
+---------------------------------+
```

SOUNDEX(str)
Gibt den Standard-Soundex-String der Zeichenkette zurück (z. B. für phonetische Suche) (alternativ `expr1 SOUNDS LIKE expr2`).

```
SELECT SOUNDEX('Meier'), SOUNDEX('Maier'), SOUNDEX('Mayer')
```

```
+------------------+------------------+------------------+
| SOUNDEX('Meier') | SOUNDEX('Maier') | SOUNDEX('Mayer') |
+------------------+------------------+------------------+
| M600             | M600             | M600             |
+------------------+------------------+------------------+
```

Aufgaben: **f_SQL_F02**

1. Erstellen Sie ein `SELECT`-Statement, welches die Anrede (Herr, Frau) der Spieler (Datenbank FB) ausgibt. Dabei sollen folgende Spalten angezeigt werden: Name, Vorname, Geschlecht, Anrede (sortiert nach Anrede absteigend)
2. Bauen Sie nachfolgendes `SELECT`-Statement so um, dass anstelle von `null` bei `B-ID` nichts und bei `Bezeichnung` „kein Beitrag zugeordnet“ angezeigt wird.

   ```
   SELECT m.ID,m.Nachname,m.Vorname,b.ID AS 'B-ID',b.Bezeichnung
      FROM TMITGLIED m LEFT JOIN TBEITRAG b ON m.Beitrag_ID = b.ID
      ORDER BY m.ID;
   ```

3. Geben Sie alle Spieler mit Name und Vorname aus. Dabei sollen diese nach der Länge des Nachnamens sortiert sein.
4. Geben Sie alle Spieler mit Name und Vorname aus. Dabei soll zwischen Name und Vorname ein „, ␣“ gesetzt werden.
5. Erstellen Sie eine Abfrage, welche die Werte der Spalte Ort der Tabelle Ort in einem Hexadezimalwert ausgibt!
6. Erstellen Sie ein Abfrage, welche die ersten 5 Zeichen des Mannschaftsnamen der Tabelle Mannschaft ausgibt!
7. Erstellen Sie eine Abfrage für die Tabelle Spieler, welche den Namen in Großbuchstaben, den Vornamen in Kleinbuchstaben und das Geburtsdatum ausgibt!

String-Vergleichsfunktionen

Jokerzeichen

`expr LIKE pat [ESCAPE 'escape_char']`

Mustervergleich unter Verwendung von Jokerzeichen.
Hierbei gilt: 1 = `TRUE`, 0 = `FALSE`

`%` entspricht einer beliebigen Anzahl von Zeichen (einschließlich null Zeichen).

_ entspricht genau einem Zeichen.

Soll `%` oder _ als Zeichen verwendet werden, so muss es mit `\` oder `escape_char` maskiert werden.

```
SELECT 'Das␣Haus␣ist␣rot!' LIKE '%Haus%'
```

```
+-----------------------------------+
| 'Das␣Haus␣ist␣rot!' LIKE '%Haus%' |
+-----------------------------------+
| 1                                 |
+-----------------------------------+
```

Reguläre Ausdrücke

`expr REGEXP pat`

Mustervergleich unter Verwendung von Regulären Ausdrücken.
Hierbei gilt: 1 = `TRUE`, 0 = `FALSE`

Die regulären Ausdrücke entsprechen im Wesentlichen denen, wie sie in Java, Grep, Sed, ...verwendet werden. Hier eine kurze Übersicht. Weitere Möglichkeiten finden sich im MySQL-Handbuch.

`^` Anfang einer Zeile/Strings

`$` Ende einer Zeile/Strings

`.` Jedes beliebige Zeichen

`*` vorheriges Zeichen 0-n mal.

`+` vorheriges Zeichen 1-n mal.

`?` vorheriges Zeichen 0-1 mal.

`|` oder

```
SELECT 'Das␣Haus␣ist␣rot!' REGEXP '^.*␣Haus␣.*'
```

```
+-----------------------------------------+
| 'Das␣Haus␣ist␣rot!' REGEXP '^.*␣Haus␣.*' |
+-----------------------------------------+
| 1                                       |
+-----------------------------------------+
```

`STRCMP(expr1,expr2)`

Gibt 0 zurück, wenn die Strings identisch sind, -1, wenn das erste Argument entsprechend der aktuellen Sortierreihenfolge kleiner ist als das zweite, und 1 in jedem anderen Fall.

```
SELECT STRCMP('Affe','Baum'), STRCMP('Baum','Baum'), STRCMP('add','abs')
```

```
+-----------------------+-----------------------+---------------------+
| STRCMP('Affe','Baum') | STRCMP('Baum','Baum') | STRCMP('add','abs') |
+-----------------------+-----------------------+---------------------+
| -1                    | 0                     | 1                   |
+-----------------------+-----------------------+---------------------+
```

Numerische Funktionen

Neben den normalen arithmetischen Operationen (+, -, *, /) stehen folgende Funktion zur Verfügung:

ABS(X)
Gibt den absoluten Wert von X zurück.

```
SELECT ABS(5), ABS(-7)
```

```
+--------+---------+
| ABS(5) | ABS(-7) |
+--------+---------+
| 5      | 7       |
+--------+---------+
```

ACOS(X)
Arkuskosinus
ASIN(X)
Arkussinus
ATAN(X)
Arkustangens
ATAN2(Y,X)
Gibt den Arkustangens der beiden Variablen X und Y zurück. Dies ähnelt der Berechnung des Arkustangens von Y / X.
COS(X)
Kosinus
COT(X)
Kotangens
SIN(X)
Sinus
TAN(X)
Tangens
CEILING(X)
Gibt den kleinsten Integer-Wert zurück, der nicht kleiner als X ist (alternativ `CEIL(X)`).

```
SELECT CEIL(1.2345)
```

```
+--------------+
| CEIL(1.2345) |
+--------------+
| 2            |
+--------------+
```

CRC32(expr)
Berechnet einen Prüfsummenwert (CRC, Cyclic Redundancy Check) und gibt einen vorzeichenlosen 32-Bit-Wert zurück.

```
SELECT CRC32('Das␣Haus␣ist␣rot!')
```

```
+----------------------------+
| CRC32('Das␣Haus␣ist␣rot!') |
+----------------------------+
| 681516774                  |
+----------------------------+
```

DEGREES(X)
Umwandlung von Rad in Grad.
EXP(X)
Berechnet die Potenz auf Basis von e.
FLOOR(X)
Gibt den größten Integer-Wert zurück, der nicht größer als X ist.

```
SELECT FLOOR(1.2345)
```

```
+---------------+
| FLOOR(1.2345) |
+---------------+
| 1             |
+---------------+
```

LN(X)
natürlicher Logarithmus
LOG(X), LOG(B,X)
Logarithmus (natürlich, bzw. zur Basis B)
LOG2(X)
Logarithmus (zur Basis 2)
LOG10(X)
Logarithmus (zur Basis 10)
MOD(N,M)
Modulooperation
PI()
Pi

```
SELECT PI()
```

```
+----------+
| PI()     |
+----------+
| 3.141593 |
+----------+
```

POW(X,Y), POWER(X,Y)
Potenz
RADIANS(X)
Wandelt Grad in Rad.
RAND(), RAND(N)
Zufallszahl zwischen 0 und 1. `N` Anfangswert für eine wiederholbare Sequenz.

```
SELECT RAND()
```

```
+---------------------+
| RAND()              |
+---------------------+
| 0.24376793414705242 |
+---------------------+
```

ROUND(X), ROUND(X,D)
Gibt das Argument `X` gerundet auf den nächstgelegenen Integer zurück. Bei `D` auf entsprechende Dezimalstellen gerundet.

```
SELECT ROUND(1.2345,2)
```

```
+-----------------+
| ROUND(1.2345,2) |
+-----------------+
| 1.23            |
+-----------------+
```

SIGN(X)
Gibt das Vorzeichen des Arguments als -1, 0 oder 1 abhängig davon zurück, ob `X` negativ, `null` oder positiv ist.
SQRT(X)
Quadratwurzel
TRUNCATE(X,D)
Gibt die Zahl `X` auf `D` Dezimalstellen beschnitten zurück.

```
SELECT TRUNCATE(1.2389,2)
```

```
+--------------------+
| TRUNCATE(1.2389,2) |
+--------------------+
| 1.23               |
+--------------------+
```

Datums- und Zeitfunktionen

CURDATE()
Gibt das aktuelle Datum im Format `'YYYY-MM-DD'` aus. `CURTIME()` gibt die aktuelle Zeit aus. `NOW()` gibt beides aus.
(alternativ `CURRENT_DATE()` bzw. `CURRENT_TIME()` bzw. `CURRENT_TIMESTAMP()`)

```
SELECT CURDATE(), CURTIME(), NOW(), CURRENT_TIMESTAMP()
```

```
+------------+-----------+----------------------+----------------------+
| CURDATE()  | CURTIME() | NOW()                | CURRENT_TIMESTAMP()  |
+------------+-----------+----------------------+----------------------+
| 2013-06-02 | 14:08:16  | 2013-06-02 14:08:16.0 | 2013-06-02 14:08:16.0 |
+------------+-----------+----------------------+----------------------+
```

ADDDATE(date,INTERVAL expr type), ADDDATE(expr,days)
Addiert zu einem Datum einen entsprechenden Wert.
(alternativ `DATE_ADD()`)
Das Gegenstück ist `SUBDATE()` bzw. `DATE_SUB()`.

```
SELECT CURDATE(), ADDDATE(CURDATE(), INTERVAL 1 WEEK)
```

```
+------------+-------------------------------------+
| CURDATE()  | ADDDATE(CURDATE(), INTERVAL 1 WEEK) |
+------------+-------------------------------------+
| 2013-06-02 | 2013-06-09                          |
+------------+-------------------------------------+
```

type Wert	*Erwartetes expr-Format*
MICROSECOND	MICROSECONDS
SECOND	SECONDS
MINUTE	MINUTES
HOUR	HOURS
DAY	DAYS
WEEK	WEEKS
MONTH	MONTHS
QUARTER	QUARTERS
YEAR	YEARS
SECOND_MICROSECOND	'SECONDS.MICROSECONDS'
MINUTE_MICROSECOND	'MINUTES.MICROSECONDS'

Tab. 5.1: `type` und `expr` für `INTERVAL`

ADDTIME(expr,expr2)
Addiert zu einer Uhrzeit einen entsprechenden Wert.
(alternativ `DATE_ADD()`)
CONVERT_TZ(dt,from_tz,to_tz)
Wandelt eine Zeitzone in eine andere um.
DATE(expr)
Extrahiert den Datumsteil aus dem `DATE`- oder `DATETIME`-Ausdruck.
DATEDIFF(expr,expr2)
Gibt die Anzahl der Tage zwischen dem Startdatum `expr` und dem Enddatum `expr2` zurück.

```
SELECT DATEDIFF('2013-01-10', '2013-01-01')
```

```
+--------------------------------------+
| DATEDIFF('2013-01-10', '2013-01-01') |
+--------------------------------------+
| 9                                    |
+--------------------------------------+
```

`DATE_FORMAT(date,format)`
Formatiert den Wert `date` entsprechend dem String `format`.
`STR_TO_DATE(str,format)`
Umkehrung zu `DATE_FORMAT()`.

format	*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) Datums- und Zeitfunktionen
`%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'

Tab. 5.2: Datumsformat

DAY(date)
Gibt den Tag im Monat zurück (alternativ `DAYOFMONTH()`).
DAYNAME(date)
Gibt den Wochentag zurück.

```
SELECT CURRENT_DATE(), DAY(CURRENT_DATE()), DAYNAME(CURRENT_DATE())
```

```
+----------------+--------------------+-------------------------+
| CURRENT_DATE() | DAY(CURRENT_DATE()) | DAYNAME(CURRENT_DATE()) |
+----------------+--------------------+-------------------------+
| 2013-06-02     | 2                  | Sunday                  |
+----------------+--------------------+-------------------------+
```

DAYOFMONTH(date)
Gibt den Tag im Monat zurück.
DAYOFWEEK(date)
Gibt den Wochentagindex zurück (1 = Sonntag, 2 = Montag, ...).
DAYOFYEAR(date)
Gibt den Tag im Jahr zurück.
FROM_UNIXTIME(unix_timestamp), FROM_UNIXTIME(unix_timestamp,format)
Gibt den Unix-Timestamp als Datum zurück.
HOUR(time)
Gibt die Stundenangabe zurück.
MINUTE(time)
Gibt die Minutenangabe zurück.
SECOND(time)
Gibt die Sekundenangabe zurück.
LAST_DAY(date)
Gibt den letzten Tag des Monats zurück.
MONTH(date)
Gibt den Monat (0 ... 12) zurück.
MONTHNAME(date)
Gibt den Monatsnamen zurück.
YEAR(date)
Gibt das Jahr zurück.
PERIOD_ADD(P,N)
Fügt `N` Monate zum Zeitraum `P` (im Format `YYMM` oder `YYYYMM`) hinzu.
PERIOD_DIFF(P1,P2)
Gibt die Anzahl der Monate zwischen den Zeiträumen `P1` und `P2` zurück.
QUARTER(date)
Gibt das Quartal zurück.
WEEK(date[,mode])
Gibt die Nummer der Woche zurück.

Modus	*Erster Wochentag*	*Bereich*	*Woche 1 ist die erste Woche ...*
0	Sonntag	0–53	mit einem Sonntag in diesem Jahr
1	Montag	0–53	mit mehr als drei Tagen innerhalb dieses Jahres
2	Sonntag	1–53	mit einem Sonntag in diesem Jahr
3	Montag	1–53	mit mehr als drei Tagen innerhalb dieses Jahres
4	Sonntag	0–53	mit mehr als drei Tagen innerhalb dieses Jahres
5	Montag	0–53	mit einem Montag in diesem Jahr
6	Sonntag	1–53	mit mehr als drei Tagen innerhalb dieses Jahres
7	Montag	1–53	mit einem Montag in diesem Jahr

Tab. 5.3: `mode` für die Woche

YEARWEEK(date), YEARWEEK(date,start)
Gibt Jahr und Woche für ein Datum zurück.

```
SELECT CURRENT_DATE(), WEEK(CURRENT_DATE(), 3)
```

```
+----------------+-------------------------+
| CURRENT_DATE() | WEEK(CURRENT_DATE(), 3) |
+----------------+-------------------------+
| 2013-06-02     | 22                      |
+----------------+-------------------------+
```

WEEKDAY(date)
Gibt den Wochentagindex zurück (0 = Montag, 1 = Dienstag, ...).
WEEKOFYEAR(date)
Gibt die Kalenderwoche zurück.
TIME(expr)
Gibt die Zeit zurück.
TIMEDIFF(expr,expr2)
Gibt die Zeitdifferenz zurück.
TIME_FORMAT(time,format)
Gibt die Uhrzeit formatiert zurück.
TIME_TO_SEC(time)
Gibt die Sekunden zu einer Zeit zurück.
TO_DAYS(date)
Gibt die Tage eines Datums zurück (seit Jahr 0).

```
SELECT CURRENT_DATE(), TO_DAYS(CURRENT_DATE()),  DATEDIFF(CURRENT_DATE(),'0000-01-01')+1
```

```
+----------------+-------------------------+----------------------------------------+
| CURRENT_DATE() | TO_DAYS(CURRENT_DATE()) | DATEDIFF(CURRENT_DATE(),'0000-01-01')+1 |
+----------------+-------------------------+----------------------------------------+
| 2013-06-02     | 735386                  | 735386                                 |
+----------------+-------------------------+----------------------------------------+
```

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(date)
Gibt die Unix-Timestamp (aktuell bzw. von) zurück.
UTC_DATE()
Gibt das UTC-Datum zurück.
UTC_TIME()
Gibt die UTC-Uhrzeit zurück.
UTC_TIMESTAMP()
Gibt den UTC-Timestamp zurück.

```
SELECT UNIX_TIMESTAMP(), UTC_DATE(), UTC_TIME(), UTC_TIMESTAMP()
```

```
+------------------+------------+------------+-----------------------+
| UNIX_TIMESTAMP() | UTC_DATE() | UTC_TIME() | UTC_TIMESTAMP()       |
+------------------+------------+------------+-----------------------+
| 1370174896       | 2013-06-02 | 12:08:16   | 2013-06-02 12:08:16.0 |
+------------------+------------+------------+-----------------------+
```

Alter berechnen

Nachfolgend wird gezeigt, wie man das Alter (Geburtsdatum) berechnen kann.

Die Beispielswerte werden in folgender Tabelle gespeichert.

```
CREATE TABLE IF NOT EXISTS DATETEST (
  date1  DATE,
  date2 DATE
)
```

```
INSERT INTO DATETEST VALUES
   ('2013-01-05', '2010-01-04'),
   ('2013-01-05', '2010-01-05'),
   ('2013-01-05', '2010-01-06'),
   (CURRENT_DATE(), '1999-08-21'),
   (CURRENT_DATE(), '1970-01-01');
```

Bei der Berechnung des Alters kann man nicht nur einfach die Differenz aus den Jahren bilden. Ist der Monat und Tag von `date2` nach dem Monat und Tag von `date1`, so muss ein Jahr abgezogen werden. Dies kann über den booleschen Wert (1 = `TRUE`, 0 = `FALSE`) erfolgen.

Die Berechnung sieht dabei wie folgt aus:

```
SELECT date1, date2,
      YEAR(date1) - YEAR(date2) as 'Jahre',
      (DATE_FORMAT(date1, '%m%d') < DATE_FORMAT(date2, '%m%d')) as 'Korrektur',
      YEAR(date1) - YEAR(date2) -
      (DATE_FORMAT(date1, '%m%d') < DATE_FORMAT(date2, '%m%d')) as 'Alter'
   FROM DATETEST;
```

```
+------------+------------+-------+-----------+-------+
| date1      | date2      | Jahre | Korrektur | Alter |
+------------+------------+-------+-----------+-------+
| 2013-01-05 | 2010-01-04 | 3     | 0         | 3     |
| 2013-01-05 | 2010-01-05 | 3     | 0         | 3     |
| 2013-01-05 | 2010-01-06 | 3     | 1         | 2     |
| 2013-06-02 | 1999-08-21 | 14    | 1         | 13    |
| 2013-06-02 | 1970-01-01 | 43    | 0         | 43    |
+------------+------------+-------+-----------+-------+
```

Eine einfache Methode ist auch (meist ausreichend genau), einfach die Differenz der Tage durch 365,25 zu Teilen (alle vier Jahre ist ein Schaltjahr) und den Nachkommaanteil abzuschneiden.

```
SELECT date1, date2, TRUNCATE(DATEDIFF(date1,date2) / 365.25,0) as 'Alter'
   FROM DATETEST;
```

```
+------------+------------+-------+
| date1      | date2      | Alter |
+------------+------------+-------+
| 2013-01-05 | 2010-01-04 | 3     |
| 2013-01-05 | 2010-01-05 | 3     |
| 2013-01-05 | 2010-01-06 | 2     |
| 2013-06-02 | 1999-08-21 | 13    |
| 2013-06-02 | 1970-01-01 | 43    |
+------------+------------+-------+
```

Aufgaben: **f_SQL_FUNC**

1. Geben Sie alle Spieler nach Geburtsdatum sortiert aus.
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat
2. Geben Sie alle männlichen Mitglieder aus.
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat, Geschlecht
3. Geben Sie alle Spieler und ihr Alter aus (Sortierung nach Alter).
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat, Alter
4. Geben Sie alle Spieler aus, die vor dem '1990-01-01' geboren sind (Sortierung nach Alter absteigend).
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat, Alter
5. Geben Sie alle Spieler aus, die mindestens 20 Jahre alt sind (Sortierung nach Alter absteigend).
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat, Alter
6. Geben Sie alle Spieler aus, die im August geboren sind (nach Geburtstag aufsteigend und nach Alter absteigend sortiert).
   Dabei sollen folgende Felder angezeigt werden:
   S_ID, Name, Vorname, GebDat, Alter, Tag

7a. Geben Sie alle Spieler nach Mannschaften sortiert aus (mit `LEFT JOIN`).
   Dabei sollen folgende Felder angezeigt werden:
   MannschName, Name, Vorname

7b. Geben Sie alle Spieler nach Mannschaften sortiert aus (mit `INNER JOIN`).
   Dabei sollen folgende Felder angezeigt werden:
   MannschName, Name, Vorname

8. Geben Sie alle Spiele und die dazugehörigen Mannschaften aus.
   Dabei sollen folgende Felder angezeigt werden:
   MannschName, Name, Vorname

5.1.3 Gruppierung

Die Gruppierung (mit `GROUP BY`) dient zum Zusammenfassen von Datensätzen für die Auswertung durch Aggregatsfunktionen für beispielsweise statistische Auswertungen.

Für die Beispiele wird einerseits die FB-Datenbank verwendet und eine einfache Tabelle, gerade für die Aggregatsfunktionen.

```
CREATE TABLE IF NOT EXISTS ARTIKEL (
  AID       INTEGER      NOT NULL,
  L_ID      CHAR(1)      NOT NULL,
  Preis     DECIMAL(4,2)
)
```

```
INSERT INTO ARTIKEL VALUES
   (1,'A', 2.45),
   (1,'B', 2.99),
   (2,'B', 2.99),
   (3,'A', 10.50),
   (4,'B', 1.45),
   (4,'C', 1.35),
   (5,'C', 1.99),
   (6,'D', 2.45),
   (6,'A', 2.45),
   (7,'D', 1.25),
   (8,'D', 21.45),
   (9,'E', 2.45),
   (10,'F', 1.25);
```

```
SELECT * FROM ARTIKEL
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 1   | A    | 2.45  |
| 1   | B    | 2.99  |
| 2   | B    | 2.99  |
| 3   | A    | 10.50 |
| 4   | B    | 1.45  |
| 4   | C    | 1.35  |
| 5   | C    | 1.99  |
| 6   | D    | 2.45  |
| 6   | A    | 2.45  |
| 7   | D    | 1.25  |
| 8   | D    | 21.45 |
| 9   | E    | 2.45  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Funktionen zur Benutzung in GROUP BY-Klauseln

Nachfolgende Funktionen stellen eine Auswahl dar. Alle Funktionen finden sie im MySQL-Handbuch.

AVG([DISTINCT] expr)
Gibt den Durchschnittswert von `expr` zurück.

DISTINCT

`DISTINCT` sorgt dafür, dass bei der Auswahl doppelte Einträge nicht berücksichtigt werden (Damit die Ausgabe nicht zu lang wird, wird diese mit `LIMIT` begrenzt).

```
SELECT Ort FROM ORT LIMIT 5
```

```
+-----------------+
| Ort             |
+-----------------+
| München         |
| München         |
| Mönchengladbach |
| Mönchengladbach |
| Regensburg      |
+-----------------+
```

```
SELECT DISTINCT Ort FROM ORT LIMIT 3
```

```
+-----------------+
| Ort             |
+-----------------+
| München         |
| Mönchengladbach |
| Regensburg      |
+-----------------+
```

COUNT(expr)
Gibt die Anzahl der Nicht-`NULL`-Werte zurück. `COUNT(*)` verwendet alle abgerufenen Datensätze unabhängig davon, ob diese `NULL`-Werte enthalten oder nicht.
`COUNT(DISTINCT expr,[expr...])` gibt die Anzahl unterschiedlicher Nicht-`NULL`-Werte zurück.

```
SELECT COUNT(Ort) FROM ORT
```

```
+------------+
| COUNT(Ort) |
+------------+
| 36         |
+------------+
```

```
SELECT Ort, COUNT(Ort) AS 'Anzahl'
   FROM ORT
   GROUP BY Ort
   ORDER BY 'Anzahl'
   LIMIT 5
```

```
+-----------------+--------+
| Ort             | Anzahl |
+-----------------+--------+
| München         | 11     |
| Mönchengladbach | 2      |
| Regensburg      | 1      |
| Freiburg        | 2      |
| Leverkusen      | 1      |
+-----------------+--------+
```

GROUP_CONCAT(expr)
Dieser gibt die verketteten Nicht-NULL-Werte einer Gruppe als String zurück.

```
GROUP_CONCAT(
          [DISTINCT] expr [,expr ...]
          [ORDER BY unsigned_integer | col_name | expr
          [ASC | DESC] [,col_name ...]]
          [SEPARATOR str_val])
```

DISTINCT entfernt dabei Dubletten, ORDER BY sortiert und SEPARATOR legt das Trennzeichen fest.

Vertikale Ausgabe des Ergebnisses:

```
SELECT GROUP_CONCAT(DISTINCT Ort ORDER BY Ort) AS 'Liste' FROM ORT
```

```
Liste=Augsburg,Bremen,Dortmund,Düsseldorf,Frankfurt am Main,Freiburg,Fürth,Gelsenkirchen,
Hamburg,Hannover,Leverkusen,Mainz,Mönchengladbach,München,Nürnberg,Odelshausen,Regensburg,
Stuttgart,Testhausen,Wolfsburg,Zunzenhausen
```

MIN([DISTINCT] expr), MAX([DISTINCT] expr)
Gibt den Mindest- bzw. den Höchstwert von expr zurück.

SUM([DISTINCT] expr)
Gibt die Summe von expr zurück.

HAVING-Klausel

Möchte man bei der GROUP BY-Klausel eine Bedingung verwenden, so scheidet die WHERE-Klausel aus, da hier keine Aggregatsfunktionen erlaubt sind. HAVING schafft hier Abhilfe.

```
SELECT Ort, COUNT(Ort)
   FROM ORT
   GROUP BY Ort
   HAVING COUNT(Ort) > 1
   ORDER BY Ort
```

```
+-----------------+------------+
| Ort             | COUNT(Ort) |
+-----------------+------------+
| Dortmund        | 2          |
| Freiburg        | 2          |
| Hamburg         | 2          |
| Mainz           | 2          |
| Mönchengladbach | 2          |
| München         | 11         |
+-----------------+------------+
```

Aufgaben: **f_SQL_GROUP**

1a. Geben Sie alle Orte und die Anzahl an Mannschaften aus (Sortiert nach der Anzahl absteigend).
Verwenden Sie dazu einen INNER JOIN.
Dabei sollen folgende Datenfelder gezeigt werden:
Ort, Anzahl Mannschaften

1b. Geben Sie alle Orte und die Anzahl an Mannschaften aus (Sortiert nach der Anzahl absteigend).
Verwenden Sie dazu einen LEFT JOIN.
Dabei sollen folgende Datenfelder gezeigt werden:
Ort, Anzahl Mannschaften

2. Geben Sie alle Orte und das Durchschnittsalter der Spieler aus (Sortiert nach der Durchschnittsalter absteigend).
Dabei sollen folgende Datenfelder gezeigt werden:
Ort, Durchschnittsalter

3. Erstellen Sie ein Abfrage, die alle Vereine und die Anzahl ihrer Mannschaften ausgibt!

4. Erstellen Sie ein Abfrage, die nur Vereine ausgibt, die mehr als eine Mannschaft gemeldet haben!

5.1.4 Subselect

Mit Subselects lassen sich Unterabfragen in `SELECT`-Anweisungen erstellen. Für den Einsatz muss berücksichtigt werden, ob der Subselect einen Wert (Skalarunterabfrage), eine Spalte (Spaltenunterabfrage), eine Liste von Werten (Datensatzabfrage) oder eine ganze Tabelle (Tabellenunterabfrage) zurückliefert. Der Subselect wird dabei immer in runden Klammern gesetzt.

in der WHERE-Klausel

Will man alle Preise anzeigen, die billiger sind als der Durchschnittspreis, so wird der Subselect in der `WHERE`-Klausel verwendet und liefert hier genau einen Wert für den Vergleich zurück.

```
SELECT * FROM ARTIKEL WHERE Preis < (SELECT AVG(Preis) FROM ARTIKEL)
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 1   | A    | 2.45  |
| 1   | B    | 2.99  |
| 2   | B    | 2.99  |
| 4   | B    | 1.45  |
| 4   | C    | 1.35  |
| 5   | C    | 1.99  |
| 6   | D    | 2.45  |
| 6   | A    | 2.45  |
| 7   | D    | 1.25  |
| 9   | E    | 2.45  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Zum Vergleich, der Durchschnittspreis ist hierbei:

```
SELECT AVG(Preis) AS 'Durchschnittspreis' FROM ARTIKEL
```

```
+--------------------+
| Durchschnittspreis |
+--------------------+
| 4.232308           |
+--------------------+
```

Liefert die Unterabfrage eine Liste, so kann dies beispielsweise mit `IN` abgefragt werden.

```
SELECT m.MannschName, o.Ort
   FROM MANNSCHAFT m, VEREIN v, ORT o
   WHERE m.V_ID=v.V_ID AND v.O_ID=o.O_ID AND
      o.Ort IN (SELECT Ort FROM ORT WHERE Ort LIKE 'M%')
```

```
+----------------------------------------+-----------------+
| MannschName                            | Ort             |
+----------------------------------------+-----------------+
| FC Bayern Profis Herren                | München         |
| FC Bayern Profis Damen                 | München         |
| FC Bayern Profis Herren 2. Mannschaft  | München         |
| FC Bayern C-Jugend w, 1. Mannschaft    | München         |
| FC Bayern D-Juniorinnen                | München         |
| 1860 München Profis Herren             | München         |
| Borussia Mönchengladbach Profis Herren | Mönchengladbach |
| Mainz 05 Profis Herren                 | Mainz           |
| FC Herta München 1. Mannschaft         | München         |
| FC Neuhadern 1. Mannschaft             | München         |
| FC Olympia Moosach 1. Mannschaft       | München         |
| FC Stern München 1. Mannschaft         | München         |
| MTV-München 1. Mannschaft              | München         |
| Münchner SportVgg 1906 1. Mannschaft   | München         |
| Rot-Weiß Oberföhring 1. Mannschaft     | München         |
| SC Bogenhausen 1. Mannschaft           | München         |
| SC München 50 1. Mannschaft            | München         |
+----------------------------------------+-----------------+
```

in der FROM-Klausel

Auch ein Subselect in der FROM-Klausel ist möglich. Dabei muss aber zwingend ein Alias verwendet werden.

```
SELECT x.Vorname, count(x.Vorname) AS 'Anzahl'
   FROM (SELECT Vorname FROM SPIELER s, MANNSCHAFT m WHERE s.M_ID=m.M_ID) x
   GROUP BY 1 ORDER BY 2 DESC LIMIT 4;
```

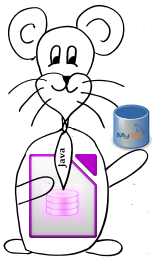
```
+-----------+--------+
| Vorname   | Anzahl |
+-----------+--------+
| Jan       | 3      |
| Dennis    | 2      |
| Sven      | 2      |
| Alexander | 2      |
+-----------+--------+
```

als Spalte

Für jeden Spaltenwert kann auch ein Subselect verwendet werden. Zu beachten ist aber, dass dieser Subselect für jeden Datensatz in der Haupttabelle einmal ausgeführt wird. Wird eine Tabelle in der Haupt- und Unterabfrage verwendet, so müssen zwei verschiedene Aliase verwendet werden.

```
SELECT o.Ort,
   (SELECT COUNT(*) FROM SPIELER s, MANNSCHAFT m, VEREIN v, ORT oo
      WHERE s.M_ID=m.M_ID AND m.V_ID=v.V_ID AND v.O_ID=oo.O_ID AND
            o.Ort=oo.Ort AND s.Geschlecht LIKE 'w%'
   ) AS 'w',
   (SELECT COUNT(*) FROM SPIELER s, MANNSCHAFT m, VEREIN v, ORT oo
      WHERE s.M_ID=m.M_ID AND m.V_ID=v.V_ID AND v.O_ID=oo.O_ID AND
            o.Ort=oo.Ort AND s.Geschlecht LIKE 'm%'
   ) AS 'm'
   FROM ORT o
   GROUP BY o.Ort
   ORDER BY 1;
```

```
+-------------------+---+----+
| Ort               | w | m  |
+-------------------+---+----+
| Augsburg          | 0 | 0  |
| Bremen            | 0 | 0  |
| Dortmund          | 0 | 3  |
| Düsseldorf        | 0 | 0  |
| Frankfurt am Main | 0 | 3  |
| Freiburg          | 0 | 2  |
| Fürth             | 0 | 0  |
| Gelsenkirchen     | 0 | 4  |
| Hamburg           | 0 | 3  |
| Hannover          | 0 | 11 |
| Leverkusen        | 0 | 3  |
| Mainz             | 0 | 9  |
| Mönchengladbach   | 0 | 3  |
| München           | 3 | 4  |
| Nürnberg          | 0 | 0  |
| Odelshausen       | 0 | 0  |
| Regensburg        | 0 | 0  |
| Stuttgart         | 0 | 13 |
| Testhausen        | 0 | 0  |
| Wolfsburg         | 0 | 13 |
| Zunzenhausen      | 0 | 0  |
+-------------------+---+----+
```



Geschwindigkeit

Da der Subselect einen erheblichen Aufwand für die Datenbank darstellt, sollte man immer überlegen, ob nicht auch ein einfacher JOIN möglich ist, der wesentlich schneller ist.

Aufgaben: **f_SQL_SUBS**

1a. Geben Sie den/die jüngsten Spieler/in aus.
Verwenden Sie dazu `LIMIT`. Dabei sollen folgende Datenfelder gezeigt werden:
Name, Alter

1b. Geben Sie den/die jüngsten Spieler/in aus.
Verwenden Sie dazu einen Subselect. Dabei sollen folgende Datenfelder gezeigt werden:
Name, Alter

2. In welcher Liga spielen Mannschaften mit der höchsten Spielerzahl. Geben Sie diese Mannschaften aus (Beschreibung(Liga), MannschName, Spielerzahl)!

3. Geben Sie die Tabelle der 1. Bundesliga nach Rang aus. Verwenden Sie dafür eine Variable, die den Rang speichert und jeweils um eins erhöht (Rang, Mannschaft, Anzahl Spieltage, Punkte, Tore)! Die Mannschaft mit den meisten Punkt ist im Rang ganz oben. Bei Punktgleichheit entscheidet die Anzahl der Tore.

5.1.5 Gebräuchliche Abfragen

Welches ist die höchste Artikelnummer?

```
SELECT MAX(AID) FROM ARTIKEL
```

```
+----------+
| MAX(AID) |
+----------+
| 10       |
+----------+
```

Welches ist der teuerste Artikel?

```
SELECT AID, L_ID, Preis
   FROM ARTIKEL
   WHERE Preis=(SELECT MAX(Preis) FROM ARTIKEL);
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 8   | D    | 21.45 |
+-----+------+-------+
```

Welches ist der billigste Artikel (einer oder mehrere)?

```
SELECT AID, L_ID, Preis
   FROM ARTIKEL
   WHERE Preis=(SELECT MIN(Preis) FROM ARTIKEL);
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 7   | D    | 1.25  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Eine andere Lösung arbeitet mit `LIMIT`, zeigt aber nur einen Wert an!

```
SELECT AID, L_ID, Preis
   FROM ARTIKEL
   ORDER BY Preis ASC
   LIMIT 1;
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 10  | F    | 1.25  |
+-----+------+-------+
```

Höchster Preis je Lieferant?

```
SELECT L_ID, MAX(Preis)
   FROM ARTIKEL
   GROUP BY L_ID
   ORDER BY MAX(Preis) DESC;
```

```
+------+------------+
| L_ID | MAX(Preis) |
+------+------------+
| D    | 21.45      |
| A    | 10.50      |
| B    | 2.99       |
| E    | 2.45       |
| C    | 1.99       |
| F    | 1.25       |
+------+------------+
```

Niedrigster Preis je Artikel/Lieferant?

```
SELECT AID, L_ID, Preis
   FROM ARTIKEL a
   WHERE Preis=(SELECT MIN(a2.preis)
                FROM ARTIKEL a2
                WHERE a.AID = a2.AID);
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 1   | A    | 2.45  |
| 2   | B    | 2.99  |
| 3   | A    | 10.50 |
| 4   | C    | 1.35  |
| 5   | C    | 1.99  |
| 6   | D    | 2.45  |
| 6   | A    | 2.45  |
| 7   | D    | 1.25  |
| 8   | D    | 21.45 |
| 9   | E    | 2.45  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

7 Views, Trigger, Index, ...

In diesem Kapitel finden sich die erweiterten Funktion von SQL, wie beispielsweise das Anlegen von Views, die Verwendung von selbst erstellten Funktionen usw.

Inhalt

7.1 Views 114
7.2 Index 117
7.3 Funktionen, Prozeduren 118
7.3.1 Arbeiten mit Parametern 119
7.3.2 Arbeiten mit lokalen Variablen und Kontrollstrukturen 120
7.4 Prepared Statements 124
7.5 Trigger 125
7.6 Transaktionen 127

7.1 Views

Views sind Abfragen, die in der Datenbank fest gespeichert werden. Sie stellen somit virtuelle Tabellen dar. Dabei wird ein entsprechender `SELECT`-Befehl gespeichert (meist schon in kompilierter Form), der dann bei jedem Zugriff ausgeführt wird.

Diese werden mit `CREATE VIEW` angelegt. Dabei sieht der vereinfachte Aufruf wie folgt aus:

```
CREATE
   [OR REPLACE]
   [DEFINER = { user | CURRENT_USER }]
   VIEW view_name [(column_list)]
   AS select_statement
```

CREATE ... VIEW Erstellt eine neue View. Dabei gelten die selben Bedingungen, wie beim Erstellen einer Tabelle.

OR REPLACE Ist eine View schon vorhanden, so wird diese ersetzt.

DEFINER Legt fest, unter welchen Rechten die View ausgeführt wird. Nur mit entsprechenden Admin-Rechten bei der Definition kann eine View mit anderen Rechten ausgeführt werden.

view_name Legt den Namen der View fest. Da Tabellen und Views im selben Namensraum gespeichert werden, darf beim Anlegen der View keine Tabelle mit dem selben Namen existieren.

column_list Legt die Spaltennamen fest. Wird hier nichts angegeben, werden die Spaltennamen der `SELECT`-Anweisung übernommen.

AS Hier folgt das `SELECT`-Statement.

Die wichtigsten Beschränkungen sind dabei:

- Die `SELECT`-Anweisung darf in der `FROM`-Klausel keine Unterabfrage enthalten.
- Die `SELECT`-Anweisung darf keine System- oder Benutzervariablen verwenden.
- Die in der View-Definition verwendeten Tabellen müssen vorhanden sein.
- Eine View darf keinen Trigger verwenden.
- Eine `ORDER BY`-Klausel in der `SELECT`-Klausel überschreibt eine evtl. vorhandene Definition der View.
- Der Ausdruck darf nicht zu komplex werden, z. B. mehrere verknüpfte Tabellen mit zusätzlicher `WHERE`-Bedingung (Abhängig von der MySQL-Versionnummer).

Mit `DROP VIEW` wird eine View gelöscht, mit `ALTER VIEW` geändert. Eine View-Definition wird mit `SHOW CREATE VIEW` angezeigt.

Verwendet man sehr häufig bestimmte Abfragen auf eine Tabelle, evtl. sogar über Verknüpfungen, so macht eine View Sinn.

Beispiel

Nachfolgendes Beispiel legt eine View an.

```
CREATE OR REPLACE VIEW WSPIELER AS
   SELECT S_ID, M_ID, Name, Vorname FROM SPIELER WHERE Geschlecht='weiblich';
```

Ruft man `SHOW TABLES` auf, so wird die View als Tabelle dargestellt.

```
SHOW TABLES;
```

```
+---------------------+
| Tables_in_FB        |
+---------------------+
| GROESSE_SPIELFELD   |
| KATEGORIE           |
| LIGA                |
| MANNSCHAFT          |
| ORT                 |
| PLATZ_STADION       |
| SPIEL               |
| SPIELER             |
| VEREIN              |
| WSPIELER            |
| ZT_SPIEL_MANNSCHAFT |
| ZT_SPIEL_SPIELER    |
+---------------------+
```

Auch ein `DESC WSPIELER` funktioniert.

```
DESC WSPIELER;
```

```
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| S_ID    | int(11)     | NO   |     | 0       |       |
| M_ID    | int(11)     | YES  |     | null    |       |
| Name    | varchar(30) | NO   |     | null    |       |
| Vorname | varchar(30) | YES  |     | null    |       |
+---------+-------------+------+-----+---------+-------+
```

Die Definition erhält man dann über `SHOW CREATE VIEW` (Ausgabe vertikal).

```
SHOW CREATE VIEW WSPIELER;
```

```
View=WSPIELER
Create View=CREATE ALGORITHM=UNDEFINED DEFINER=`test`@`localhost` SQL SECURITY DEFINER ⤶
VIEW `WSPIELER` AS select `SPIELER`.`S_ID` AS `S_ID`,`SPIELER`.`M_ID` AS `M_ID`,`SPIELER⤶
`.`Name` AS `Name`,`SPIELER`.`Vorname` AS `Vorname` from `SPIELER` where (`SPIELER`.`⤶
Geschlecht` = 'weiblich')
character_set_client=latin1
collation_connection=latin1_swedish_ci
```

Ein `SELECT` funktioniert genau so, als wäre die View eine echte Tabelle.

```
SELECT * FROM WSPIELER LIMIT 5;
```

```
+------+------+----------------+---------+
| S_ID | M_ID | Name           | Vorname |
+------+------+----------------+---------+
| 8    | 5    | Putigam        | Sarah   |
| 9    | 5    | Cross          | Niki    |
| 10   | 5    | Hartmannsegger | Sandra  |
+------+------+----------------+---------+
```

Aufgaben: **f_SQL_VIEW**

1. Welche Beschränkungen gelten für eine View?
   - ☐ Keine, eine View kann beliebige `SELECT`-Klauseln enthalten.
   - ☐ Es dürfen keine Subselects verwendet werden.
   - ☐ Es dürfen keine Subselects in der `WHERE`-Klausel verwendet werden.
   - ☐ Es dürfen keine Subselects in der `FROM`-Klausel verwendet werden.
   - ☐ Eine View darf keine Trigger verwenden.

2a. Erstellen Sie eine View (Name `MANNSCHAFT_SPIELER_COUNT`), welche alle Mannschaften und die Anzahl an Spielern ausgibt (sortiert nach Mannschaften).

2b. Erstellen Sie eine entsprechende Abfrage, um die View `MANNSCHAFT_SPIELER_COUNT` zu testen.

3a. Erstellen Sie eine View (Name `MANNSCHAFT_SPIELER_COUNT_MW`), welche alle Mannschaften und die Anzahl an Spielern gesamt und nach Geschlecht getrennt ausgibt (sortiert nach Mannschaften).

3b. Erstellen Sie eine entsprechende Abfrage, um die View `MANNSCHAFT_SPIELER_COUNT_MW` zu testen.

7.2 Index

Ein Index sortiert eine Tabelle nach einem bestimmten Kriterium und ermöglicht es so, Abfragen zu beschleunigen. Viele DBMS legen automatisch einen Index auf den Primärschlüssel, um so die Performance zu erhöhen und den gesuchten Datensatz schneller zu finden.

Der Index wird mit `CREATE INDEX` angelegt. Dabei sieht der vereinfachte Aufruf wie folgt aus:

```
CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX index_name
   [USING index_type]
   ON tbl_name (index_col_name,...)
```

CREATE ... INDEX Erstellt einen neuen Index.

- **UNIQUE** Es wird ein eindeutiger Index ohne Dubletten erstellt (siehe hierzu `CREATE TABLE`).
- **FULLTEXT** Der Index unterstützt auch Volltextsuche.
- **SPATIAL** Nur für MyISAM-Tabellen mit raumbezogenen Spalten.

index_name Legt den Namen des Index fest.

ON tbl_name Legt die Tabelle für den Index fest.

(index_col_name,...) Legt die Spalten für den Index fest. Mit der Syntax `col_name(N)` kann der Index auf die ersten N Zeichen einer String-Spalte beschränkt werden.

Die maximale Anzahl von Indizes pro Tabelle und die maximale Indexlänge bestimmt die verwendete Engine.

MySQL verwendet den Index u. a.:

- Zum schnellen Auffinden von Datensätzen, die der `WHERE`-Klausel entsprechen
- Zum Abrufen von Datensätzen bei Joins
- Zum Auffinden von `MIN()` - oder `MAX()` -Werten
- Zum Sortieren oder Gruppieren einer Tabelle

Mit `DROP INDEX` wird der Index gelöscht, mit `SHOW INDEX FROM <table>` werden die Indizes einer Tabelle angezeigt.

MySQL erstellt beispielsweise für die Tabelle `SPIELER` zwei Indizes. Einen für die Spalte mit dem Primärschlüssel und einen für die Spalte mit dem Fremdschlüssel (siehe Auszug).

```
mysql> SHOW INDEX FROM SPIELER;
+---------+------------+----------+-------------+------+------------+
| Table   | Non_unique | Key_name | Column_name | Null | Index_type |
+---------+------------+----------+-------------+------+------------+
| SPIELER |          0 | PRIMARY  | S_ID        |      | BTREE      |
| SPIELER |          1 | M_ID     | M_ID        | YES  | BTREE      |
+---------+------------+----------+-------------+------+------------+
```

Beispiel

Nachfolgendes Beispiel erzeugt einen Index.

```
CREATE INDEX SPIELER_RNR ON SPIELER (Ruecken_Nr);
```

7.3 Funktionen, Prozeduren

MySQL ermöglicht es, benutzerdefinierte Funktionen bzw. Prozeduren zu erstellen.

Eine Funktion wird mit `CREATE FUNCTION`, eine Prozedur mit `CREATE PROCEDURE` erstellt. Dabei sieht der vereinfachte Aufruf wie folgt aus:

```
CREATE PROCEDURE name ([parameter[,...]])
    [characteristic ...] routine_body

CREATE FUNCTION name ([parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body
```

CREATE ... name Erstellt eine neue Funktion oder Prozedur. Der wesentliche Unterschied dabei ist, dass eine Prozedur keinen Rückgabewert hat. Dabei wird die Funktion bzw. Prozedur in der Standarddatenbank gespeichert und steht allen zur Verfügung. Soll diese nur in einer Datenbank verwendet werden, so muss der DB-Name voran gesetzt werden (`dbname.fktname`).

parameter Legt die Parameter fest. Wenn keine Parameter übergeben werden, muss eine leere Parameterliste mit Klammern `()` verwendet werden. Die Parameter bei Prozeduren können als `IN` (Standard), `OUT` und `INOUT` definiert werden. Funktionsparameter werden immer als `IN`-Parameter übergeben.

characteristic Legt die Art des Codes fest, beispielsweise `LANGUAGE SQL`. Mehr dazu im MySQL-Handbuch.

routine_body Der eigentliche Code. Besteht dieser aus mehr als einer Anweisung, so muss der Code mit `BEGIN` und `END` eingeschlossen werden.
Da die äußere Definition mit ';' abgeschlossen wird und die innere Definition im Normalfall auch, führt dies zu Problemen. Daher muss der Befehls-Trenner beispielsweise mit `DELIMITER//` umdefiniert und am Ende wieder auf den Standardwert gesetzt werden.

```
DELIMITER //
CREATE PROCEDURE myprocedure()
   BEGIN
     SELECT 1+1;
   END
   //
DELIMITER ;
```

Eine gespeicherte Funktion bzw. Prozedur wird mit `ALTER PROCEDURE` bzw. `ALTER FUNCTION` geändert. Gelöscht mit `DROP PROCEDURE` bzw. `DROP FUNCTION`. Der Status wird mit `SHOW PROCEDURE STATUS` bzw. `SHOW FUNCTION STATUS` angezeigt, der Code mit `SHOW PROCEDURE CODE` bzw. `SHOW FUNCTION CODE`[1].

Beispiel

Nachfolgendes Beispiel definiert eine Prozedur, die den ältesten Spieler ausgibt. Zur Sicherheit wird diese gelöscht, falls sie vorher existiert.

```
DROP PROCEDURE IF EXISTS FB.aeltesterSpieler
```

```
CREATE PROCEDURE FB.aeltesterSpieler()
   BEGIN
      SELECT * FROM SPIELER ORDER BY GebDat ASC LIMIT 1;
   END
```

Der Aufruf erfolgt mit `CALL`:

```
CALL FB.aeltesterSpieler()
```

```
+------+------+--------+---------+------------+------------+------------+---------------+
| S_ID | M_ID | Name   | Vorname | Ruecken_Nr | GebDat     | Geschlecht | Gehalt_monatl |
+------+------+--------+---------+------------+------------+------------+---------------+
| 35   | null | Kobold | Pumukel | 99         | 1962-02-21 | männlich   | 0.00          |
+------+------+--------+---------+------------+------------+------------+---------------+
```

[1] Diese Funktion muss im MySQL-Server aktiviert sein!

7.3.1 Arbeiten mit Parametern

Parameter können per `IN`, `OUT` und `INOUT` definiert werden. Diese werden wie normale Variablen verwendet. Bei `IN` gibt es keine Möglichkeit, Werte zurückzugeben, bei den anderen beiden schon.

Im Abschnitt 5.1.5 auf Seite 92 wurde ein Subselect verwendet, um beispielsweise den billigsten Artikel zu ermitteln.

```
SELECT AID, L_ID, Preis
   FROM ARTIKEL
   WHERE Preis=(SELECT MIN(Preis) FROM ARTIKEL);
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 7   | D    | 1.25  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Wird der Wert für den billigsten Artikel öfters gebraucht, macht es Sinn, diesen in einer Variablen zu speichern.

Mit nachfolgender Definition, wird der billigste Preis in der Variablen `bilPreis` gespeichert. Dazu wird `INTO` im `SELECT`-Statement verwendet.

```
DROP PROCEDURE IF EXISTS BEISPIEL.billigsterPreis
```

```
CREATE PROCEDURE BEISPIEL.billigsterPreis(OUT bilPreis decimal(4,2))
   BEGIN
      SELECT MIN(Preis) INTO bilPreis FROM ARTIKEL;
   END
```

Die so definierte Variable kann dann wie folgt verwendet werden:

```
CALL BEISPIEL.billigsterPreis(@billigsterPreis);
SELECT @billigsterPreis;
```

```
+------------------+
| @billigsterPreis |
+------------------+
| 1.25             |
+------------------+
```

Jetzt kann man den Subselect durch die Variable ersetzen.

```
CALL BEISPIEL.billigsterPreis(@billigsterPreis);
SELECT AID, L_ID, Preis
   FROM ARTIKEL
   WHERE Preis=@billigsterPreis;
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 7   | D    | 1.25  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Man kann aber auch die Variable verwenden, um die Ausgabe zu nummerieren. Variablen werden dabei mit `SET` definiert und bekommen mit `:=` den Wert zugewiesen.

```
SET @counter:=0;
SELECT (@counter:=@counter+1) AS 'Position', AID, Preis
   FROM ARTIKEL LIMIT 5;
```

```
+----------+-----+-------+
| Position | AID | Preis |
+----------+-----+-------+
| 1        | 1   | 2.45  |
| 2        | 1   | 2.99  |
| 3        | 2   | 2.99  |
| 4        | 3   | 10.50 |
| 5        | 4   | 1.45  |
+----------+-----+-------+
```

7.3.2 Arbeiten mit lokalen Variablen und Kontrollstrukturen

In einer Prozedur oder Funktion ist es auch möglich, lokale Variablen zu definieren und diverse Kontrollstrukturen zu verwenden.

IF-Anweisung

Mit `IF` wird ein einfaches Bedingungskonstrukt zur Verfügung gestellt.

```
IF bedingung THEN
   sql-list
[ELSEIF bedingung THEN
   sql-list
] ...
[ELSE
   sql-list
]
END IF
```

Wenn die Bedingung zutrifft, wird die zugehörige SQL-Anweisungsliste (ein bzw. mehrere Befehle) ausgeführt. Trifft keine Bedingung zu, wird die Anweisungsliste aus der `ELSE`-Klausel ausgeführt.

CASE-Anweisung

Die `CASE`-Anweisung stellt eine Mehrfachauswahl zur Verfügung.

```
CASE variable
WHEN wert1 THEN
   sql-list
[WHEN wert2 THEN
   sql-list
] ...
[ELSE
   sql-list
]
END CASE
```

Oder:

```
CASE
WHEN bedingung THEN
   sql-list
[WHEN bedingung THEN
   sql-listt
] ...
[ELSE
   sql-listt
]
END CASE
```

Es werden die Anweisungen ausgeführt, bei denen der Wert dem Inhalt der Variablen entspricht bzw. die Bedingung zutrifft. Trifft nichts zu, werden die SQL-Anweisungen bei `ELSE` ausgeführt.

LOOP-Anweisung

Mit `LOOP` lässt sich eine einfache Schleife realisieren. Die Schleife wird solange durchlaufen, bis sie mit `LEAVE` verlassen wird. Mit `ITERATE` wird die entsprechende Schleife von oben erneut durchlaufen. Optional kann die `LOOP`-Anweisung auch beschriftet werden. Werden beide Label verwendet, so müssen diese identisch sein.

```
[begin_label:] LOOP
   sql-list
END LOOP [end_label]
```

REPEAT-Anweisung

Die `REPEAT`-Schleife ist eine fußgesteuerte Schleife, die solange wiederholt wird, bis die Bedingung bei `UNTIL` erfüllt ist. Auch hier kann mit `LEAVE` und `ITERATE` gearbeitet werden.

```
[begin_label:] REPEAT
   sql-list
UNTIL bedingung
END REPEAT [end_label]
```

WHILE-Anweisung

Die `WHILE-DO`-Schleife ist eine kopfgesteuerte Schleife, die solange wiederholt wird, bis die Bedingung erfüllt ist. Auch hier kann mit `LEAVE` und `ITERATE` gearbeitet werden.

```
[begin_label:] WHILE bedingung DO
    sql-list
END WHILE [end_label]
```

Beispiel

In nachfolgendem Beispiel wird eine Funktion erstellt, die jeweils den ersten Buchstaben eines Wort groß, alle anderen klein schreibt.

```
DROP FUNCTION IF EXISTS Capitalize
```

```
CREATE FUNCTION Capitalize (str VARCHAR(1024)) RETURNS VARCHAR(1024) CHARSET utf8
   BEGIN
      DECLARE result VARCHAR(1024) DEFAULT '';
      DECLARE part   VARCHAR(1024) DEFAULT '';

      -- solange, bis kein Leerzeichen mehr da ist
      WHILE INSTR(TRIM(str) , '␣') > 0 DO
         -- Aufteilen bis ' '
         SET part = SUBSTR(str, 1, INSTR(str , '␣'));

         -- erster Buchstabe groß, Rest klein
         SET result = CONCAT(result, UCASE(LEFT(part,1)),LOWER(SUBSTR(part,2)));

         -- Teil entfernen
         SET str = SUBSTR(str, INSTR(str , '␣') + 1);
      END WHILE;
      RETURN TRIM(CONCAT(result, UCASE(LEFT(str,1)),LOWER(SUBSTR(str,2))));
   END
```

Aufgerufen wird die Funktion dann wie folgt:

```
SELECT Capitalize('autoren:␣michael␣niedermair,␣martina␣pomper');
```

```
+-----------------------------------------------------------+
| Capitalize('autoren:␣michael␣niedermair,␣martina␣pomper') |
+-----------------------------------------------------------+
| Autoren: Michael Niedermair, Martina Pomper               |
+-----------------------------------------------------------+
```

```
SELECT Capitalize('aUTOREN:␣MICHAEL␣niedermair,␣martina␣POMPER');
```

```
+-----------------------------------------------------------+
| Capitalize('aUTOREN:␣MICHAEL␣niedermair,␣martina␣POMPER') |
+-----------------------------------------------------------+
| Autoren: Michael Niedermair, Martina Pomper               |
+-----------------------------------------------------------+
```

Aufgaben: **f_SQL_FUNCPROC**

1a. Erstellen Sie eine Funktion, die das Alter berechnet.

1b. Geben Sie mit obiger erstellten Funktion das Alter der Spieler aus (Sortiert nach Alter absteigend).

2. Geben Sie alle Orte und das Durchschnittsalter der Spieler aus (Sortiert nach der Durchschnittsalter absteigend). Verwenden Sie dabei obige Funktion.
   Dabei sollen folgende Datenfelder gezeigt werden:
   Ort, Durchschnittsalter

3. Geben Sie den/die jüngsten Spieler/in aus. Verwenden Sie dazu einen Subselect und obige Funktion. Dabei sollen folgende Datenfelder gezeigt werden:
   Name, Alter

Cursor

Ein Cursor ist ein Zeiger auf eine SQL-Abfrage, der es ermöglicht, in einer Prozedur/Funktion jeden ausgewählten Datensatz einzeln zu behandeln.

```
DECLARE cursorname CURSOR FOR sql-statement
```

Wenn der Cursor an das Ende des Abfrageergebnis kommt, wird der Status `NOT FOUND` (genaugenommen der Code '02000') gesetzt, der Durchlauf ist somit beendet und der Cursor kann wieder geschlossen werden. Dazu wird die boolesche Variable `done` definiert. Erreicht der Cursor das Ende der Abfrage, kann über einen `HANDLER` festgelegt werden, dass die Variable `done` auf `TRUE` gesetzt wird.

Die allgemeine Syntax lautet dabei:

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
    sql-statement
```

Für `handler_action` gilt:

CONTINUE Der Handler führt das SQL-Statement aus, bricht aber die Prozedur nicht ab.

EXIT Der Handler bricht die Prozedur sofort ab.

UNDO (wird nicht unterstützt)

Für `condition_value` gilt:

SQLSTATE Der Handler wird ausgeführt, wenn ein entsprechender Status eintritt. Mehr dazu im MySQL-Handbuch.

SQLWARNING Der Handler wird bei Warnungen ausgeführt.

NOT FOUND Der Handler wird ausgeführt, wenn Objekte etc. nicht gefunden werden.

SQLEXCEPTION Der Handler wird bei Fehlern ausgeführt.

Für obiges Beispiel gilt dann:

```
DECLARE done BOOLEAN DEFAULT FALSE;
DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=TRUE;
```

Im nächsten Schritt muss der Cursor geöffnet und nach Verwendung wieder geschlossen werden:

```
OPEN cursorname;
...
CLOSE cursorname;
```

Die einzelnen Einträge werden über `FETCH` ermittelt.

```
FETCH cursorname INTO var_name [, var_name] ...
```

Dies holt die nächste Zeile aus dem Cursor (falls vorhanden), speichert die Spalteneinträge in entsprechenden Variablen und setzt dann den Cursor um eins weiter.

Beispiel

In Abschnitt 5.1.3 auf Seite 86 wurde die Tabelle `ARTIKEL` verwendet.

```
SELECT * FROM ARTIKEL
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
| 1   | A    | 2.45  |
| 1   | B    | 2.99  |
| 2   | B    | 2.99  |
| 3   | A    | 10.50 |
| 4   | B    | 1.45  |
| 4   | C    | 1.35  |
| 5   | C    | 1.99  |
| 6   | D    | 2.45  |
| 6   | A    | 2.45  |
| 7   | D    | 1.25  |
| 8   | D    | 21.45 |
| 9   | E    | 2.45  |
| 10  | F    | 1.25  |
+-----+------+-------+
```

Nun sollen die drei Lieferanten, die Artikel über 2 Euro anbieten, in eine neue Tabelle kopiert werden. Dazu soll eine Prozedur erstellt werden.

```
DROP TABLE IF EXISTS LIEFER;
CREATE TABLE LIEFER (
  L_ID      CHAR(1)      NOT NULL,
  Preis     DECIMAL(4,2)
);
DROP PROCEDURE IF EXISTS BEISPIEL.copyLiefer;
```

```
CREATE PROCEDURE BEISPIEL.copyLiefer()
   BEGIN
      DECLARE l_l_id   CHAR(1);
      DECLARE l_preis  DECIMAL(4,2);
      DECLARE old_l_id CHAR(1) default '';
      DECLARE l_count INT default 0;
      DECLARE done BOOLEAN DEFAULT FALSE;
      DECLARE cur1 CURSOR FOR SELECT L_ID, Preis FROM ARTIKEL ORDER BY L_ID, Preis;
      DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=TRUE;
      OPEN cur1;
      schleife: WHILE NOT done DO
         FETCH cur1 INTO l_l_id, l_preis;
         -- am Ende angekommen?
         IF NOT done THEN
            -- Preis über 2 Euro?
            IF l_preis > 2 THEN
               -- anderer Lieferant?
               IF l_l_id <> old_l_id THEN
                  INSERT INTO LIEFER VALUES(l_l_id,l_preis);
                  SET old_l_id = l_l_id;
                  SET l_count = l_count + 1;
                  -- schon drei Lieferanten?
                  IF l_count >= 3 THEN
                     LEAVE schleife;
                  END IF;
               END IF;
            END IF;
         END IF;
         END WHILE;
      CLOSE cur1;
   END
```

Zu beachten ist, dass das `FETCH`-Statement, wenn es am Ende angekommen ist, also `done=TRUE` als Ergebnis `NULL` liefert. Es muss daher vor der Weiterverarbeitung der Wert von `done` geprüft werden.

In Tabelle `LIEFER` finden sich nach dem Prozeduraufruf folgende Einträge:

```
CALL BEISPIEL.copyLiefer();
SELECT * FROM LIEFER;
```

```
+------+-------+
| L_ID | Preis |
+------+-------+
| A    | 2.45  |
| B    | 2.99  |
| D    | 2.45  |
+------+-------+
```

7.4 Prepared Statements

Prepared Statements dienen dazu, SQL-Anweisung mit Platzhaltern vorzubereiten und diese dann nur noch mit entsprechenden Werten aufzurufen. Der Vorteil dabei ist, dass die SQL-Anweisung nur einmal analysiert werden muss. Anstelle eines Prepared Statements lässt sich aber auch eine Prozedur verwenden.

Die Definition lautet dabei:

```
PREPARE stmtname FROM sql-statement
```

Als SQL-Anweisungen können hier (mit MySQL 5.5) folgende Anweisungen verwendet werden:

```
ALTER TABLE
ANALYZE TABLE
CACHE INDEX
CALL
CHANGE MASTER
CHECKSUM {TABLE | TABLES}
COMMIT
{CREATE | RENAME | DROP} DATABASE
{CREATE | DROP} INDEX
{CREATE | RENAME | DROP} TABLE
{CREATE | RENAME | DROP} USER
DELETE
DO
FLUSH {TABLE | TABLES | TABLES WITH READ LOCK | HOSTS | PRIVILEGES
  | LOGS | STATUS | MASTER | SLAVE | DES_KEY_FILE | USER_RESOURCES}
GRANT
INSERT
INSTALL PLUGIN
KILL
LOAD INDEX INTO CACHE
OPTIMIZE TABLE
REPAIR TABLE
REPLACE
RESET {MASTER | SLAVE | QUERY CACHE}
REVOKE
SELECT
SET
SHOW {AUTHORS | CONTRIBUTORS | WARNINGS | ERRORS}
SHOW BINLOG EVENTS
SHOW CREATE {PROCEDURE | FUNCTION | EVENT | TABLE | VIEW}
SHOW {MASTER | BINARY} LOGS
SHOW {MASTER | SLAVE} STATUS
SLAVE {START | STOP}
TRUNCATE TABLE
UNINSTALL PLUGIN
UPDATE
```

Ausgeführt wird das Prepared Statement mit `EXECUTE USING`. Ein vorhandenes kann über `DEALLOCATE PREPARE` bzw. `DROP PREPARE` wieder gelöscht werden.

Beispiel

Es soll der billigste Lieferant eines Artikels ermittelt werden.

```
PREPARE artikel_billigLieferant FROM
   'SELECT␣AID,␣L_ID,␣Preis␣␣FROM␣ARTIKEL␣WHERE␣AID=?␣ORDER␣BY␣Preis␣LIMIT␣1';
```

```
SET @Artikel=4;
EXECUTE artikel_billigLieferant USING @Artikel;
```

```
+-----+------+-------+
| AID | L_ID | Preis |
+-----+------+-------+
|   4 | C    |  1.35 |
+-----+------+-------+
```

Aufgaben: **f_SQL_PREPSTMT**

1. Welche Vorteile bieten Prepared Statements?
2. Welche Vorteile bieten Prozeduren gegenüber Prepared Statements?
3. Erstellen Sie ein Prepared Statement, welches alle Lieferanten eines bestimmten Artikels liefert.

7.5 Trigger

Trigger sind SQL-Anweisungen, die bei Tabellen (keine Views oder temporäre Tabellen) bei bestimmten Anweisungen (INSERT, UPDATE oder DELETE) automatisch ausgeführt werden.

Der allgemeine Aufruf sieht dabei wie folgt aus:

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

DEFINER Legt fest, unter welchen Rechten der Trigger ausgeführt werden soll.

trigger_name Der Name des Triggers.

trigger_time Der Zeitpunkt, wann der Trigger ausgelöst wird. Mit BEFORE wird der Trigger vor der Anweisung und mit AFTER nach der Anweisung ausgeführt.

trigger_event Gibt an, welche Anweisung den Trigger aktiviert.

- **INSERT** Der Trigger wird aktiviert, wenn in der Tabelle eine neue Zeile eingefügt wird, beispielsweise bei INSERT..., LOAD DATA und REPLACE.
- **UPDATE** Der Trigger wird aktiviert, wenn eine Zeile geändert wird (nur UPDATE).
- **DELETE** Der Trigger wird aktiviert, wenn eine Zeile gelöscht wird, beispielsweise bei DELETE... und REPLACE.

tbl_name Für welche Tabelle der Trigger bestimmt ist.

sql-statement Die SQL-Anweisung(en) (mit BEGIN...END) für den Trigger.

Mit dem Prefix OLD kann man auf Spalten der vorhandenen Zeile zugreifen, bevor diese geändert oder gelöscht worden sind, mit NEW auf die neuen oder geänderten Spalten der Zeile.

BEFORE/AFTER UPDATE Alte Werte mit OLD verfügbar.
Neue Werte mit NEW verfügbar.

BEFORE/AFTER DELETE Alte Werte mit OLD verfügbar.

BEFORE/AFTER INSERT Neue Werte mit NEW verfügbar.

Mit SHOW TRIGGERS lassen sich erstellte Trigger anzeigen, mit DROP TRIGGER wieder löschen.

Beispiel

Im nachfolgenden Beispiel sollen das Änderungsdatum und die Änderungszeit protokolliert werden. Dazu wird die Tabelle ARTIKEL angepasst (wer für den aktuellen Benutzer und wann für die Änderungszeit) und als ARTIKEL_TR gespeichert.

```
DROP TABLE IF EXISTS ARTIKEL_TR;
CREATE TABLE ARTIKEL_TR (
  AID       INTEGER      NOT NULL,
  L_ID      CHAR(1)      NOT NULL,
  Preis     DECIMAL(4,2),
  wer       VARCHAR(30),
  wann      DATETIME
);
```

Der Trigger ARTIKEL_TR_INSERT wird beim Einfügen verwendet.

```
DROP TRIGGER IF EXISTS ARTIKEL_TR_INSERT;
CREATE TRIGGER ARTIKEL_TR_INSERT BEFORE INSERT ON ARTIKEL_TR FOR EACH ROW
   BEGIN
      SET NEW.wer  = CURRENT_USER();
      SET NEW.wann = NOW();
   END;
```

Der Trigger ARTIKEL_TR_UPDATE wird beim Ändern verwendet.

```
DROP TRIGGER IF EXISTS ARTIKEL_TR_UPDATE;
CREATE TRIGGER ARTIKEL_TR_UPDATE BEFORE UPDATE ON ARTIKEL_TR FOR EACH ROW
   BEGIN
      SET NEW.wer  = CURRENT_USER();
      SET NEW.wann = NOW();
   END;
```

Fügt man Datensätze hinzu, wird jetzt automatisch der Anwender und das aktuelle Datum gesetzt.

```
INSERT INTO ARTIKEL_TR (AID,L_ID,Preis) VALUES
   (1,'A', 2.45),(1,'B', 2.99),(2,'B', 2.99),(3,'A', 10.50),(4,'B', 1.45),
   (4,'C', 1.35),(5,'C', 1.99),(6,'D', 2.45),(6,'A', 2.45),(7,'D', 1.25),
   (8,'D', 21.45),(9,'E', 2.45),(10,'F', 1.25);
SELECT * FROM ARTIKEL_TR;
```

```
+-----+------+-------+----------------+-----------------------+
| AID | L_ID | Preis | wer            | wann                  |
+-----+------+-------+----------------+-----------------------+
| 1   | A    | 2.45  | test@localhost | 2013-06-02 14:08:16.0 |
| 1   | B    | 2.99  | test@localhost | 2013-06-02 14:08:16.0 |
| 2   | B    | 2.99  | test@localhost | 2013-06-02 14:08:16.0 |
| 3   | A    | 10.50 | test@localhost | 2013-06-02 14:08:16.0 |
| 4   | B    | 1.45  | test@localhost | 2013-06-02 14:08:16.0 |
| 4   | C    | 1.35  | test@localhost | 2013-06-02 14:08:16.0 |
| 5   | C    | 1.99  | test@localhost | 2013-06-02 14:08:16.0 |
| 6   | D    | 2.45  | test@localhost | 2013-06-02 14:08:16.0 |
| 6   | A    | 2.45  | test@localhost | 2013-06-02 14:08:16.0 |
| 7   | D    | 1.25  | test@localhost | 2013-06-02 14:08:16.0 |
| 8   | D    | 21.45 | test@localhost | 2013-06-02 14:08:16.0 |
| 9   | E    | 2.45  | test@localhost | 2013-06-02 14:08:16.0 |
| 10  | F    | 1.25  | test@localhost | 2013-06-02 14:08:16.0 |
+-----+------+-------+----------------+-----------------------+
```

Historisierung

Unter Historisierung (auch Temporale Datenhaltung) genannt, versteht man einen Mechanismus, der dafür sorgt, dass Änderungen nachvollzogen werden können und jeder Stand wieder hergestellt werden kann. Dazu gibt es viele verschiedene Verfahren und Methoden. Eine Übersicht bietet beispielsweise `https://de.wikipedia.org/wiki/Temporale_Datenhaltung`.

Eine sehr einfache Variante ist, dass alle Änderungen in einer Tabelle, quasi als LOG, gespeichert werden.

Übung:

1. Erstellen Sie in der FB-Datenbank eine Tabelle mit dem Namen `LOG`.
2. Erstellen Sie folgende Spalten mit entsprechenden sinnvollen Spaltentypen:

   Tabelle Der Name der Tabelle, bei der die Änderung erfolgt ist.

   PK Der Wert des Primärschlüssels der geänderten Tabelle.

   Name Der Name der Spalte, die geändert worden ist.

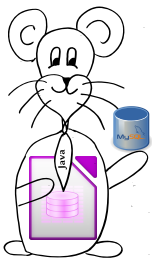
   old Der vorherige Wert.

   new Der neue Wert.

   Aktion delete, update, insert, ...

   wer Geändert durch ...

   wann Geändert am ...
3. Erstellen Sie entsprechende Trigger, die bei jeder Tabelle die Änderungen in der `LOG`-Tabelle protokollieren.
4. Wird beispielsweise ein neuer Datensatz eingefügt, so muss für jede Spalte ein Eintrag erfolgen.



Automatische Historisierung

Eine automatische Historisierung mit Triggern auf Datenbankebene ist nicht einfach zu realisieren. Ein weiteres Problem ist, dass bei DB-Historisierung der Zeitstempel verwendet wird. Will man auf einen "alten Stand" zurückgreifen, so muss man bei vielen Änderungen einen Zeitbereich verwenden, was aber nicht immer zu einem exakten Ergebnis führt.

Eine andere Variante ist beispielsweise die Historisierung mit dem Hibernate-Projekt Envas (Entity Versioning). Dabei wird immer ein Stand (komplett) mit einer Revisionsnummer verwaltet und man muss selbst auf Datenbank-Ebene nichts machen, alles wird dabei automatisch erledigt.

7.6 Transaktionen

Damit auch bei vielen SQL-Anweisungen, die zusammengehören, die Integrität der Daten gewahrt bleibt, sollen Änderungen (`DELETE`-, `UPDATE`-, `REPLACE`- oder `INSERT`), die zusammengehören, auch in einem Stück gespeichert werden. Bei einem Fehler einer einzelnen Anweisungen müssen alle zusammengehörenden Anweisungen rückgängig gemacht werden.

Unter einer Transaktion versteht man dabei den Wechsel eines konsistenten Zustand in einen anderen konsistenten Zustand. Dabei ist z. B. das ACID-Prinzip (deutsch AKID) zu beachten:

Atomarität Bei einer atomaren Operation gilt, dass die Operationen ganz oder gar nicht ausgeführt werden (Alles-oder-nichts-Eigenschaft).

Konsistenzerhaltung Dies bedeutet, dass Daten-Operationen nach Beendigung immer einen konsistenten Datenzustand hinterlassen, also alle Integritätsbedingungen, insbesondere von Schlüssel- und Fremdschlüsselbedingungen, erfüllt sind.

Isolation Durch Isolation wird verhindert, dass Operationen beispielsweise von anderen Anwendern, die eigenen Operationen beinflussen.

Dauerhaftigkeit Dauerhaftigkeit garantiert, dass nach erfolgreichem Abschluss einer Transaktion dauerhaft die Daten in der Datenbank gespeichert sind.

Dazu stehen folgende Befehle zur Verfügung (zusätzliche Parameter finden sich in der MySQL-Beschreibung):

```
START TRANSACTION | BEGIN
COMMIT
ROLLBACK
SET AUTOCOMMIT = {0 | 1}
```

Eine Transaktion wird mit `START TRANSACTION` oder mit `BEGIN` eingeleitet. Mit `COMMIT` werden die Änderungen permanent gespeichert, dagegen werden mit `ROLLBACK` alle Änderungen seit dem Start der Transaktion rückgängig gemacht. Bis zu einem `COMMIT` sind die Änderungen für andere Anwender nicht sichtbar.

Die Anweisung `SET AUTOCOMMIT` aktiviert oder deaktiviert den standardmäßigen Autocommit-Modus für die aktuelle Verbindung. Ein `START TRANSACTION` oder `BEGIN` deaktiviert für den Block den Autocommit-Modus und aktiviert diesen danach wieder.

Transaktionen werden im Normalfall nur sehr selten auf der MySQL-Konsole verwendet, sondern meist nur immer in einer Applikation, beispielsweise in Java.

8 Erweiterungen

In diesem Kapitel wird das Datenbankmodell zweimal erweitert (Stufe II & III) und es stehen viele Aufgaben zum Üben bereit.

Inhalt

8.1 Datenmodell Stufe II 130
8.2 Datenmodell Stufe III 139

8.1 Datenmodell Stufe II

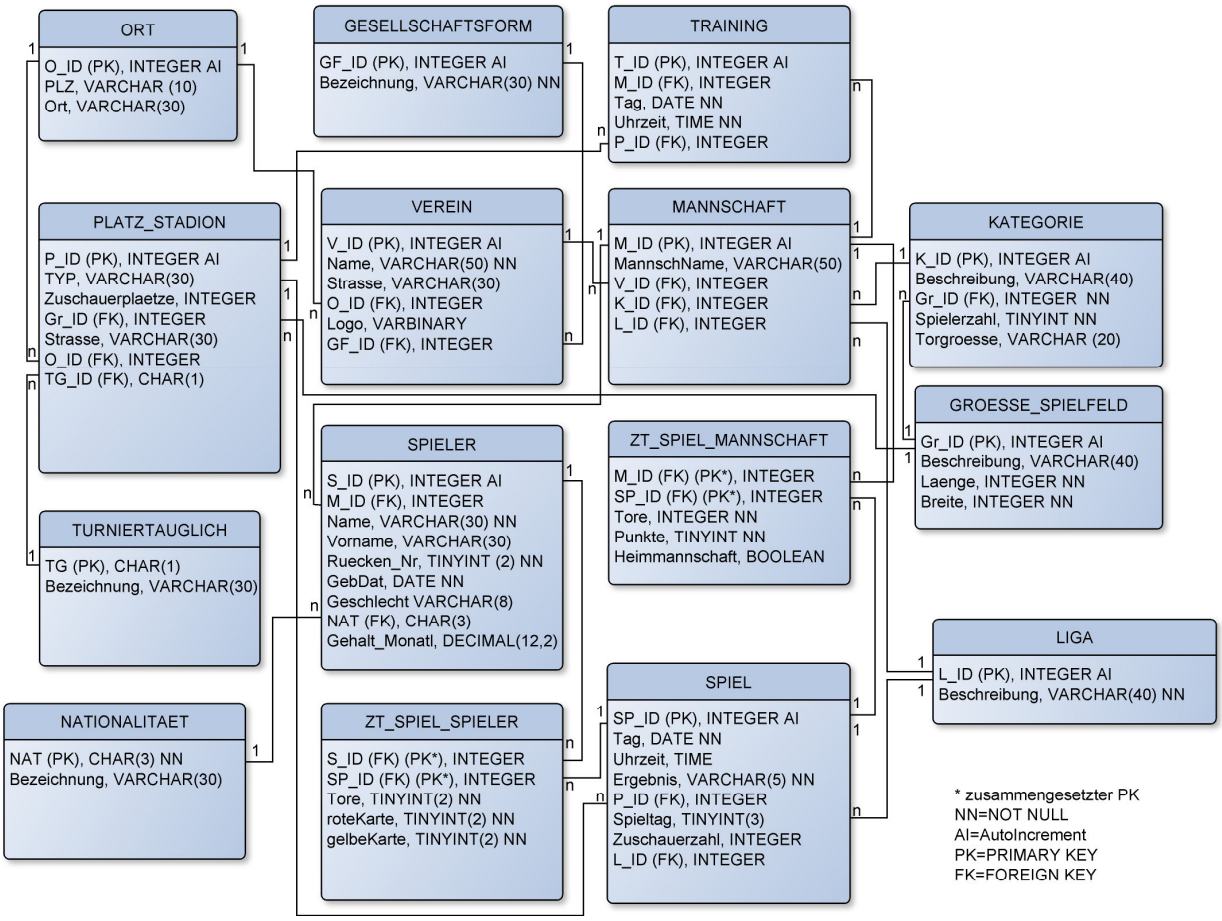


Abb. 8.1: Budesliga-Modell – Stufe II

Hinweis:

Lösen Sie nachfolgende Aufgaben und dokumentieren Sie diese selbst, z. B. mitschreiben etc.

Aufgabenblock 1 – einfache Abfragen – nur eine Tabelle

Aufgaben: **f_SQL_FB2_01**

1. Erstellen Sie eine Abfrage, die die Tabelle `SPIELER` ausgibt! Dabei sollen folgende Spalten ausgegeben werden: `S_ID`, `M_ID`, `Name`, `Vorname`, `Geschlecht` und `Gehalt_monatl`
2. Erstellen Sie eine Abfrage, die aus Tabelle `SPIELER` alle Datensätze der Spalten `Name`, `Vorname` und `GebDat` ausgibt! Die Ergebnistabelle soll nach dem Namen und dann nach dem Vornamen aufsteigend (a,b,...) sortiert sein.
3. Erstellen Sie eine Abfrage, die aus Tabelle `SPIELER` alle Datensätze der Spalten `Name`, `Vorname` und `GebDat` ausgibt!

   Der Name und der Vorname sollen mit einem Komma getrennt in einer Spalte stehen.

   Die Ergebnistabelle soll nach dem Namen und dann nach dem Vornamen aufsteigend (a,b,...) sortiert sein.
4. Erstellen Sie eine Abfrage, die aus Tabelle `Spieler` die Zeichenlänge der Spielernamen ausgibt!

   Dabei wird Name und Vorname (sollen mit einem Leerzeichen getrennt werden) als kompletter Spielername betrachtet.
5. Erstellen Sie eine Abfrage, die alle Orte der Tabelle `ORT` ausgibt, die mit einer 8 in der Spalte `PLZ` beginnen!
6. Erstellen Sie eine Abfrage, die alle Orte mit dem Anfangsbuchstaben K bis N aus der Tabelle `ORT` ausgibt! Die Ergebnistabelle soll absteigend (n,m,l,...) sortiert sein.
7. Erstellen Sie eine Abfrage, die aus der Tabelle SPIELER alle Spieler mit einem Gehalt von mindestens 600.000 Euro ausgibt! Die Ergebnistabelle soll absteigend sortiert sein und die Spalten `Name`, `Vorname` und `Gehalt_monatl` ausgeben!
8. Erstellen Sie eine Abfrage, die aus der Tabelle SPIELER alle Spieler mit der Rückennummer 1 ausgibt! Es sollen nur die Spalten `Name`, `Vorname` und `Ruecken_Nr` ausgegeben werden.
9. Erstellen Sie eine Abfrage, die aus der Tabelle `VEREIN` Vereine mit 'FC' im Vereinsnamen ausgibt! Es soll nur die Spalte `Name` angezeigt werden. Wählen Sie eine aussagekräftige Spaltenüberschrift!

Aufgabenblock 2 – Funktionen – nur eine Tabelle

Aufgaben: **f_SQL_FB2_02**

1. Erstellen Sie eine Abfrage, die alle Datensätze der Tabelle `VEREIN` zählt! Wählen Sie eine aussagekräftige Spaltenüberschrift!
2. Erstellen Sie eine Abfrage, die alle Spieler mit der Rückennummer 1 aus der Tabelle `SPIELER` zählt! Wählen Sie eine aussagekräftige Spaltenüberschrift!
3. Erstellen Sie eine Abfrage, die das Gehalt aller Spieler der Tabelle `SPIELER` summiert! Wählen Sie eine aussagekräftige Spaltenüberschrift!
4. Erstellen Sie eine Abfrage, die das Gehalt aller Spieler mit der Rückennummer 1 der Tabelle `SPIELER` summiert! Wählen Sie eine aussagekräftige Spaltenüberschrift!
5. Erstellen Sie eine Abfrage, die das maximale, das minimale und das Durchschnittsgehalt aller Spieler der Tabelle `SPIELER` ermittelt! Das durchschnittliche Gehalt soll auf 2 Stellen nach dem Komma gerundet sein. Wählen Sie aussagekräftige Spaltenüberschriften!

6a. Erstellen Sie eine Abfrage, die das Alter aller Spieler der Tabelle `SPIELER` ermittelt (Ungefähre Lösung)! Geben Sie Namen, Vornamen, Geburtsdatum und Alter aus (Sortiert nach Alter).

6b. Erstellen Sie eine Abfrage, die das Alter aller Spieler der Tabelle `SPIELER` ermittelt (genaue Lösung)! Geben Sie Namen, Vornamen, Geburtsdatum und Alter aus (Sortiert nach Alter).

7. Erstellen Sie eine Abfrage, die das Durchschnittsalter (auf 2 Nachkommastellen gerundet) aller Spieler der Tabelle `SPIELER` ermittelt!
8. Erstellen Sie nun eine Abfrage, die mittels Sub-Select alle Spieler ermittelt, die jünger sind als das Durchschnittsalter.
9. Ermitteln Sie mit Hilfe einer `IF THEN ELSE` Abfrage, ob ein Spieler in diesem Jahr bereits Geburtstag hatte oder nicht. Geben Sie zu dem Namen, Vornamen und Geburtsdatum aus.
10. Erstellen Sie eine Abfrage, die in der Tabelle `GROESSE_SPIELFELD` in einer zusätzlichen Spalte die Fläche in Quadratmeter berechnet! Wählen Sie eine aussagekräftige Spaltenüberschrift!

Aufgabenblock 3 – Verknüpfung von Tabellen

Aufgaben: **f_SQL_FB2_03**

1. Erstellen Sie eine Abfrage, die die Tabelle `KATEGORIE` mit der Tabelle `GROESSE_SPIELFELD` über das Schlüsselattribut verknüpft. Es sollen folgende Attribute ausgegeben werden: `Beschreibung`, `Spielerzahl` und `Torgroesse` aus der Tabelle `KATEGORIE` sowie `Beschreibung`, `Laenge` und `Breite` verknüpft als '<l> x <b>' aus der Tabelle `GROESSE_SPIELFELD`.
2. Welches Ergebnis bringt eine Verknüpfung der obigen Tabellen mit `UNION` bzw. `CROSS JOIN`?
3. Erstellen Sie eine Abfrage, die alle Mannschaften des 1. FC Bayern (=Verein) ausgibt.
4. Erstellen Sie eine Abfrage, die alle Spieler der 1. Herrenmannschaft des FC Bayern mit Name, Vorname und Rückennummer ausgibt.
5. Erstellen Sie eine Abfrage, die alle Spieler ausgibt, die in keiner Mannschaft spielen.
6. Erstellen Sie eine Abfrage, die alle Mannschaften mit jeweiliger Anzahl der Spieler ausgibt. Die Mannschaften ohne Spieler sollen auch gelistet sein.
7. Sind Mannschaften in einer Kategorie mit 7 Spielern gemeldet? Geben Sie diese aus! MannschName, Spielerzahl, Beschreibung (Kategorie)
8. Erstellen Sie eine Abfrage, welche die Anzahl an Aktiengesellschaften bei den Vereinen ermittelt.
9. Welche Mannschaften spielten bei Heimspielen 2:1?

Aufgabenblock 4 – GROUP BY, HAVING

Aufgaben: **f_SQL_FB2_04**

1. Erstellen Sie eine Abfrage, die pro Mannschaft die Anzahl der Spieler zählt.
2. Erstellen Sie eine Abfrage, die pro Mannschaft die Anzahl der Spieler aus Brasilien zählt.
3. Erstellen Sie eine Abfrage, die pro Gesellschaftsform die Anzahl der Vereine zählt.
4. Wie oft spielten Mannschaften bei Heimspielen 2:1?

Aufgabenblock 5 – SUB-SELECT

Aufgaben: **f_SQL_FB2_05**

1. Listen sie alle Spieler auf, deren Rückennummer der Anzahl an österreichischen Spieler in der Mannschaft 'Mainz 05 Profis Herren' entspricht!
2. Listen Sie alle Spieler auf, deren Alter der Länge eines Minispielfeldes entspricht oder größer ist!
3. Kombination aus Aufgabe 1 & 2:

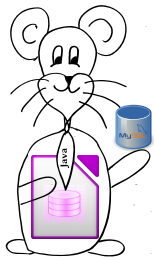
   Listen Sie alle Spieler auf, deren Rückennummer der Anzahl an brasilianischen Spieler in der Mannschaft 'VfL Woflsburg Profis Herren' entspricht und deren Alter der Länge eines Minispielfeldes entspricht oder größer ist!

Aufgabenblock 6 – VIEWS

Aufgaben: **f_SQL_FB2_06**

1. Erstellen Sie eine VIEW `BLMannschaft` mit allen Mannschaften der 1. Bundesliga Herren.
2. Erstellen Sie eine VIEW `Spieler_MLN`, die Spieler, Mannschaft, Liga und Nationalität verknüpft (Spalten: S_ID, Name, Vorname, Beschreibung, Bezeichnung). Geben Sie dann mit Hilfe der View alle Spieler aus Österreich, die in einer Mannschaft der 1. Bundesliga Herren spielen, aus.

Aufgabenblock 7 – STORED FUNCTION



Delimiter

Bei Funktionen, Prozeduren und Triggern ist es wichtig, den Delimiter entsprechend anzupassen. Mehr dazu auf Seite 118.

Aufgaben: **f_SQL_FB2_07**

1a. Löschen Sie die Prozedur `FB2.hoechstesGehalt`, wenn diese existiert.

1b. Erstellen Sie eine Prozedur `FB2.hoechstesGehalt`, die den Spieler mit dem höchsten Gehalt ausgibt (Einfache Variante mit Limit). Spalten: Name, Vorname, Gehalt_Monatl

1c. Testen Sie die Prozedur `FB2.hoechstesGehalt`.

2a. Löschen Sie die Prozedur `FB2.hoechstesGehalt2`, wenn diese existiert.

2b. Erstellen Sie eine Prozedur `FB2.hoechstesGehalt2`, die das höchste Gehalt aller Spieler ausgibt. Verwenden Sie dabei einen Parameter.

2c. Testen Sie die Prozedur `FB2.hoechstesGehalt2`.

Aufgabenblock 8 – gemischt (anspruchsvoll!)

In diesem Aufgabenblock wird schrittweise die Ausgabe der Tabelle der 1. Bundesliga zum aktuellen Zeitpunkt entwickelt.

Aufgaben: **f_SQL_FB2_08**

1. Erstellen Sie eine Abfrage, die Mannschaft der 1. Bundesliga, die Summe der Spieltage, die Summe der Punkte und die Summe der eigenen Tore ausgibt. Sortiert nach Anzahl der Punkte (absteigend), dann nach Anzahl der Tore (absteigend)
2. Nun soll in die obige Abfrage die Spalte `Rang` hinzugefügt werden, welche die derzeitige Platzierung der Mannschaft beschreibt. Hinweis: Hierfür kann eine Variable als Zähler deklariert werden: `SET @counter:=0;`

3a. Nun fehlt noch die letzte Spalte der Tabelle – die Tordifferenz. Die Tordifferenz soll anstelle der Spalte Tore erscheinen. Dazu müssen zunächst die Gegentore je Mannschaft ermittelt werden, welche dann von den Toren abgezogen werden. Die Gegentore werden jedoch in der Tabelle `ZT_SPIEL_MANNNSCHAFT` nicht erfasst, dies wären redundante Daten. D. h. man ermittelt die Gegentore, indem man Tore der Gegner-Mannschaft ermittelt. Dazu müssen zunächst die Spiel-Paarungen mit den jeweiligen Toren ermittelt werden.

3b. Geben Sie für jedes Spiel die Spielnummer `SP_ID` und beide Mannschaften nebeneinander aus (Spalte 1: ID, Spalte 2: Heimmannschaft, Spalte 3: ID, Spalte 4: Gastmannschaft)!

Hinweis: Spalte 3 und 4 kann mittels Subselect eingefügt werden. Die Tabelle Mannschaft wird hierbei zwei Mal mit einem unterschiedlichen Alias verwendet.

4. Ergänzen Sie nun die Abfrage aus Aufgabe 3 um die jeweils erzielten Tore der Mannschaften im entsprechenden Spiel. Spalte 1: ID, Spalte 2: Heimmannschaft, Spalte 3: Tore, Spalte 4: ID, Spalte 5: Gastmannschaft, Spalte 6: Tore.

5a. Als nächstes werden die Gegentore jeder Mannschaft ermittelt und summiert. D. h. es wird die Abfrage aus Aufgabe 4 in 2 Abfragen zerlegt:

Erstellen Sie eine Abfrage, die nur die Heimmannschaft und die Tore der Gegenmannschaft ausgibt (Spalte 2 und 6 aus Aufgabe 4).

5b. Erstellen Sie eine Abfrage, die nur die Gastmannschaft und die Tore der Heimmannschaft ausgibt (Spalte 5 und 3 aus Aufgabe 4).

6a. Diese beiden Ergebnistabellen sollen Grundlage für alle weiteren Abfragen dienen. Darum sollen Sie als View definiert werden:

Erstellen Sie jeweils eine VIEW für die Abfragen 5a mit dem Titel GegenT_HeimMannsch

Hinweis: SUB-SELECTS sind in VIEWS nicht zulässig, deshalb muss gewünschte View über 2 Schritte erzeugt werden!

6b. Erstellen Sie jeweils eine VIEW für die Abfragen 5b mit dem Titel GegenT_GastMannsch

7a. Ermitteln Sie die Summe aller Gegentore je Mannschaft!

Die Summe aller Gegentore je Mannschaft bei Heimspielen (aus der Tabelle/View: GegenT_HeimMannsch).

7b. Die Summe aller Gegentore je Mannschaft bei Auswärtsspielen (aus der Tabelle/View: GegenT_GastMannsch).

8. Erstellen Sie eine Abfrage, worin die Gegentore aus Heim- und Auswärtsspielen summiert werden!
9. Geben Sie nun die Tabelle der 1. Bundesliga aus!

Spalten: Rang, Mannschaft, Anzahl Spieltage, Tore, Gegentore, Tordifferenz

10a. Löschen Sie die STORED PROCEDURE `SpieleMannsch`, wenn diese existiert.

10b. Erstellen Sie ein STORED PROCEDURE `SpieleMannsch`, welche für Tabelle aus Aufgabe 4 (Ausgabe der Heim- und Gastmannschaft je Spiel mit jeweiligen Toren) mittels `IF`-Funktion die Spalten `Gew_Verl` (gew/verl/ue) und Punkte (3/0/1) aus Sicht der Heimmannschaft ergänzt.

10c. Probieren sie die STORED PROCEDURE `SpieleMannsch` aus.

8.2 Datenmodell Stufe III

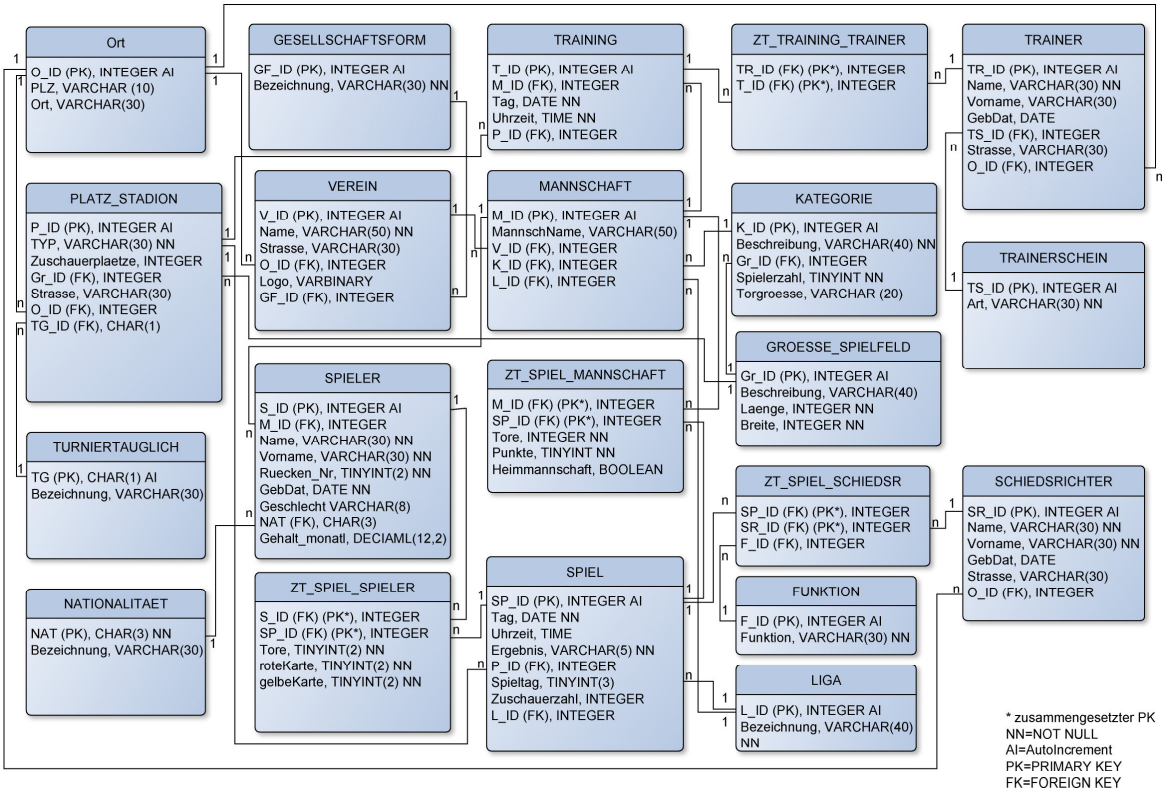


Abb. 8.2: Budesliga-Modell – Stufe III

In der Praxis kommt es oft vor, dass ein bestehendes Datenbankmodell angepasst, erweitert und geändert werden muss. Der Kunde will aber seine „Altdaten“ behalten (oft sind dies viele tausend oder mehr Datensätze) und weiter abrufbar haben, da dies oft sein Kapital darstellt. Daher ist es wichtig, sich Gedanken zu machen, wie ein „altes“ Modell auf ein „neues“ Modell umgestellt wird und dabei die Daten erhalten bleiben.

Im Datenmodell der Stufe II sind schon Daten erhalten. Ziel es es, diese „alten“ Daten in das Modell der Stufe III zu transferieren.

Aufgabe

Bauen Sie das Datenmodell der Stufe II um, so dass das Modell III entsteht.

Beachten Sie dabei folgendes:

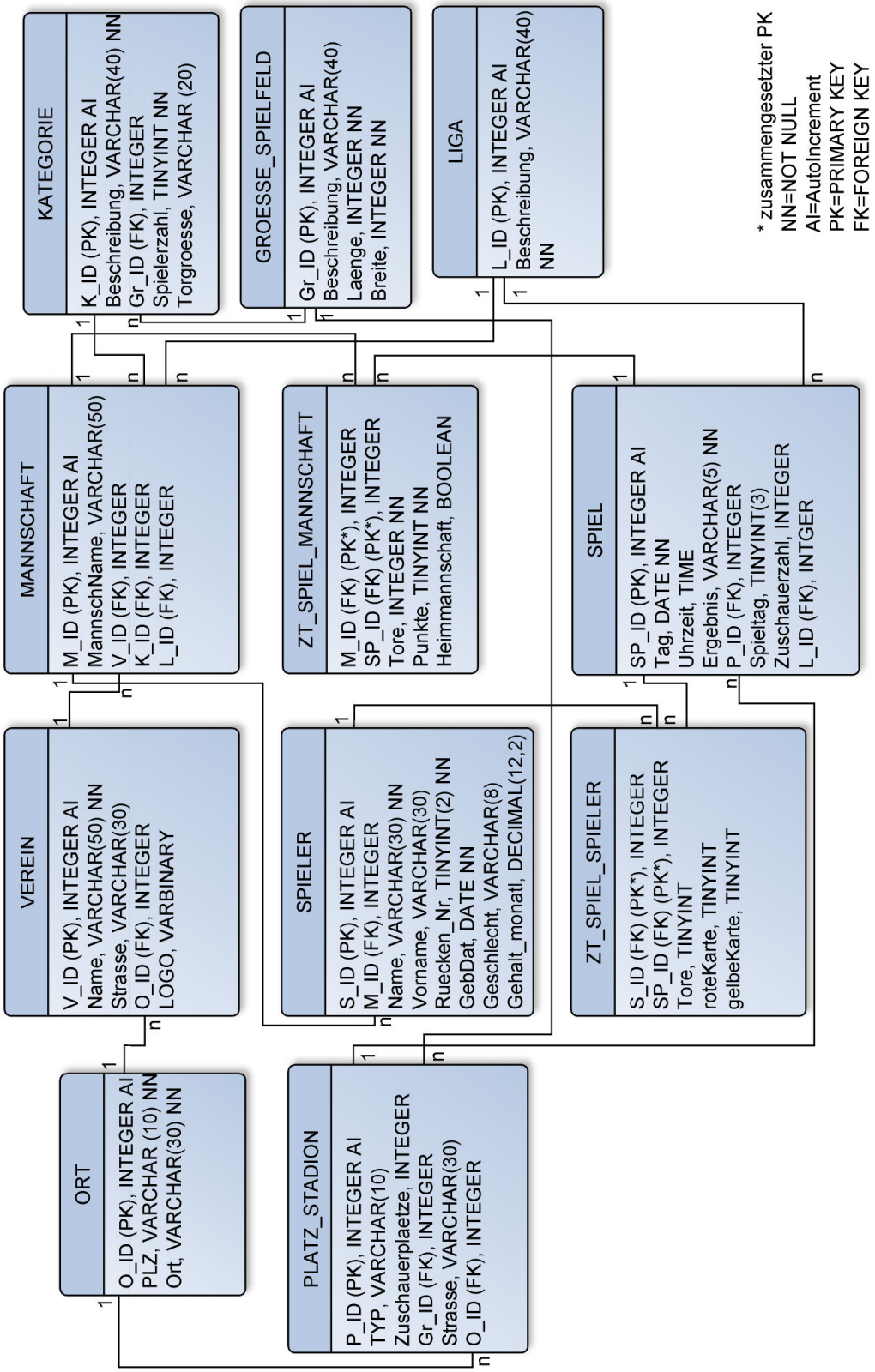
- Erstellen Sie dazu eine neue Datenbank mit dem Namen `FB3`.
- Kopieren Sie die Tabellenstruktur und die Daten aus `FB2`.
- Alle Daten aus Modell II müssen erhalten bleiben, fehlende Daten müssen entsprechend sinnvoll ergänzt werden.
- Verwendet Sie für die Umstellung keinen SQL-Dump und ändern diesen, sondern verwenden Sie nur `ALTER TABLE`, `INSERT INTO` usw.

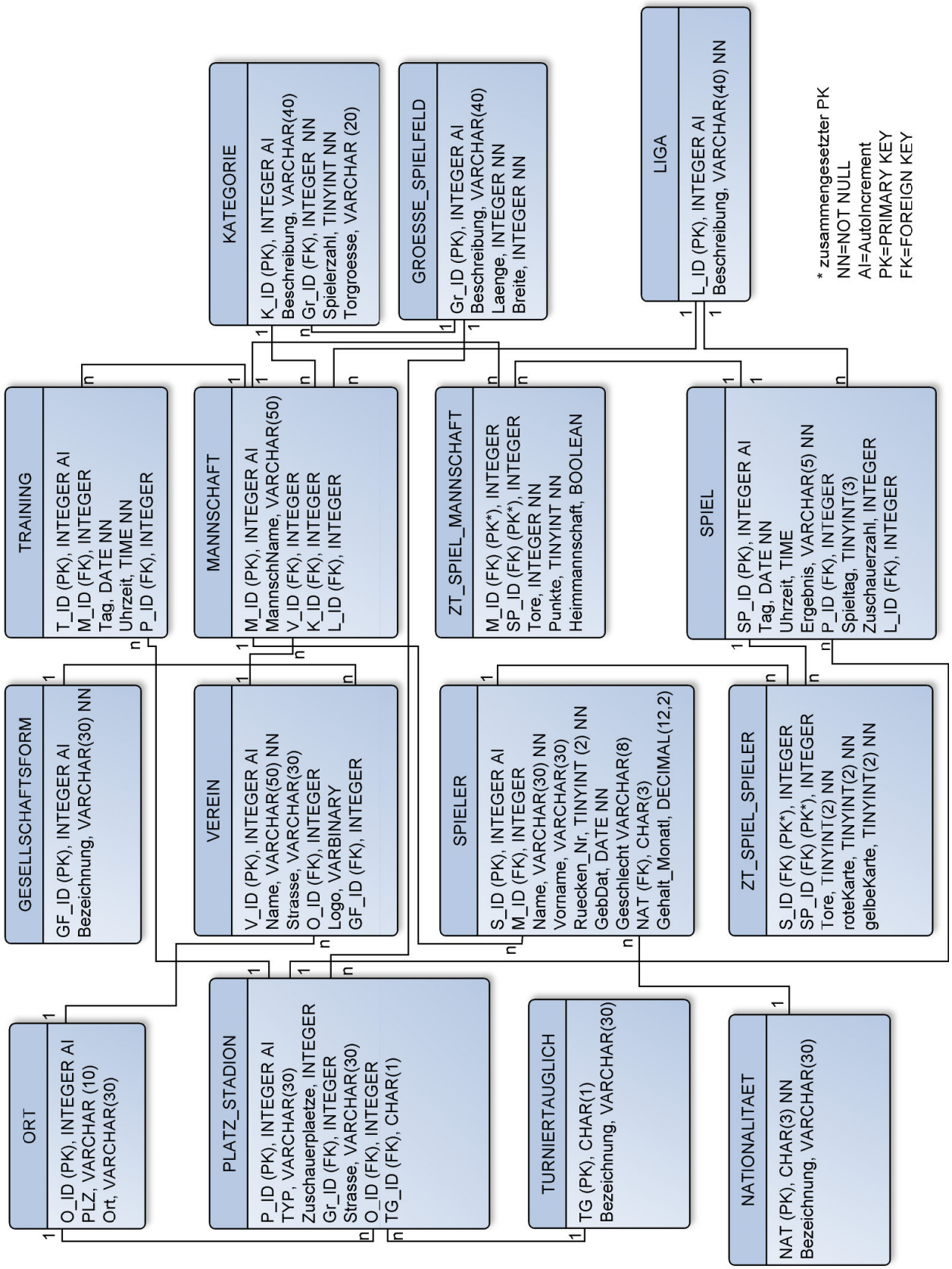
9 Anhang

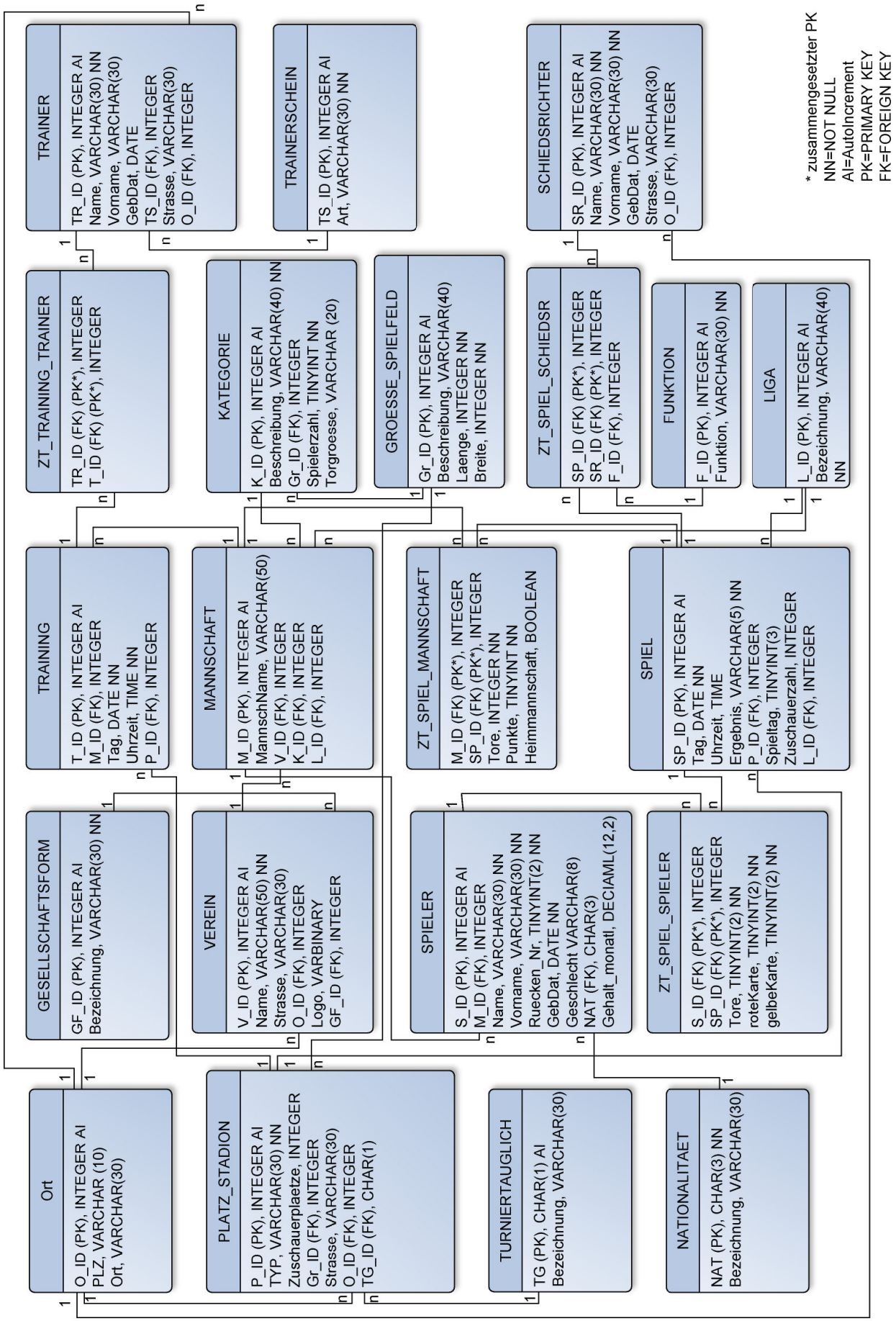
Inhalt

9.1 Datenmodelle 142
9.2 Datentypen 145
9.2.1 Ganzzahlen 145
9.2.2 Fließkommazahlen 145
9.2.3 Text 146
9.2.4 Datum und Zeit 147
9.2.5 Sonstige 147
9.3 Tools 148
9.3.1 mysql 148
9.3.2 mysqladmin 149
9.3.3 mysqlcheck 149
9.3.4 mysqldump 150
9.3.5 mysqlimport 151
9.3.6 mysqlshow 151

9.1 Datenmodelle




KATEGORIE
K_ID (PK), INTEGER AI
Beschreibung, VARCHAR(40)
Gr_ID (FK), INTEGER NN
Spielerzahl, TINYINT NN
Torgroesse, VARCHAR (20)
GROESSE_SPIELFELD
Gr_ID (PK), INTEGER AI
Beschreibung, VARCHAR(40)
Laenge, INTEGER NN
Breite, INTEGER NN
LIGA
L_ID (PK), INTEGER AI
Beschreibung, VARCHAR(40) NN
* zusammengesetzter PK
NN=NOT NULL
AI=AutoIncrement
PK=PRIMARY KEY
FK=FOREIGN KEY
TRAINING
T_ID (PK), INTEGER AI
M_ID (FK), INTEGER
Tag, DATE NN
Uhrzeit, TIME NN
P_ID (FK), INTEGER
MANNSCHAFT
M_ID (PK), INTEGER AI
MannschName, VARCHAR(50)
V_ID (FK), INTEGER
K_ID (FK), INTEGER
L_ID (FK), INTEGER
ZT_SPIEL_MANNSCHAFT
M_ID (FK) (PK*), INTEGER
SP_ID (FK) (PK*), INTEGER
Tore, INTEGER NN
Punkte, TINYINT NN
Heimmannschaft, BOOLEAN
SPIEL
SP_ID (PK), INTEGER AI
Tag, DATE NN
Uhrzeit, TIME
Ergebnis, VARCHAR(5) NN
P_ID (FK), INTEGER
Spieltag, TINYINT(3)
Zuschauerzahl, INTEGER
L_ID (FK), INTEGER
GESELLSCHAFTSFORM
GF_ID (PK), INTEGER AI
Bezeichnung, VARCHAR(30) NN
VEREIN
V_ID (PK), INTEGER AI
Name, VARCHAR(50) NN
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
Logo, VARBINARY
GF_ID (FK), INTEGER
SPIELER
S_ID (PK), INTEGER AI
M_ID (FK), INTEGER
Name, VARCHAR(30) NN
Vorname, VARCHAR(30)
Ruecken_Nr, TINYINT (2) NN
GebDat, DATE NN
Geschlecht VARCHAR(8)
NAT (FK), CHAR(3)
Gehalt_Monatl, DECIMAL(12,2)
ZT_SPIEL_SPIELER
S_ID (FK) (PK*), INTEGER
SP_ID (FK) (PK*), INTEGER
Tore, TINYINT(2) NN
roteKarte, TINYINT(2) NN
gelbeKarte, TINYINT(2) NN
ORT
O_ID (PK), INTEGER AI
PLZ, VARCHAR (10)
Ort, VARCHAR(30)
PLATZ_STADION
P_ID (PK), INTEGER AI
TYP, VARCHAR(30)
Zuschauerplaetze, INTEGER
Gr_ID (FK), INTEGER
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
TG_ID (FK), CHAR(1)
TURNIERTAUGLICH
TG (PK), CHAR(1)
Bezeichnung, VARCHAR(30)
NATIONALITAET
NAT (PK), CHAR(3) NN
Bezeichnung, VARCHAR(30)


TRAINER
TR_ID (PK), INTEGER AI
Name, VARCHAR(30) NN
Vorname, VARCHAR(30)
GebDat, DATE
TS_ID (FK), INTEGER
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
TRAINERSCHEIN
TS_ID (PK), INTEGER AI
Art, VARCHAR(30) NN
SCHIEDSRICHTER
SR_ID (PK), INTEGER AI
Name, VARCHAR(30) NN
Vorname, VARCHAR(30) NN
GebDat, DATE
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
* zusammengesetzter PK
NN=NOT NULL
AI=AutoIncrement
PK=PRIMARY KEY
FK=FOREIGN KEY
ZT_TRAINING_TRAINER
TR_ID (FK) (PK*), INTEGER
T_ID (FK) (PK*), INTEGER
KATEGORIE
K_ID (PK), INTEGER AI
Beschreibung, VARCHAR(40) NN
Gr_ID (FK), INTEGER
Spielerzahl, TINYINT NN
Torgroesse, VARCHAR (20)
GROESSE_SPIELFELD
Gr_ID (PK), INTEGER AI
Beschreibung, VARCHAR(40)
Laenge, INTEGER NN
Breite, INTEGER NN
ZT_SPIEL_SCHIEDSR
SP_ID (FK) (PK*), INTEGER
SR_ID (FK) (PK*), INTEGER
F_ID (FK), INTEGER
FUNKTION
F_ID (PK), INTEGER AI
Funktion, VARCHAR(30) NN
LIGA
L_ID (PK), INTEGER AI
Bezeichnung, VARCHAR(40)
NN
TRAINING
T_ID (PK), INTEGER AI
M_ID (FK), INTEGER
Tag, DATE NN
Uhrzeit, TIME NN
P_ID (FK), INTEGER
MANNSCHAFT
M_ID (PK), INTEGER AI
MannschName, VARCHAR(50)
V_ID (FK), INTEGER
K_ID (FK), INTEGER
L_ID (FK), INTEGER
ZT_SPIEL_MANNSCHAFT
M_ID (FK) (PK*), INTEGER
SP_ID (FK) (PK*), INTEGER
Tore, INTEGER NN
Punkte, TINYINT NN
Heimmannschaft, BOOLEAN
SPIEL
SP_ID (PK), INTEGER AI
Tag, DATE NN
Uhrzeit, TIME
Ergebnis, VARCHAR(5) NN
P_ID (FK), INTEGER
Spieltag, TINYINT(3)
Zuschauerzahl, INTEGER
L_ID (FK), INTEGER
GESELLSCHAFTSFORM
GF_ID (PK), INTEGER AI
Bezeichnung, VARCHAR(30) NN
VEREIN
V_ID (PK), INTEGER AI
Name, VARCHAR(50) NN
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
Logo, VARBINARY
GF_ID (FK), INTEGER
SPIELER
S_ID (PK), INTEGER AI
M_ID (FK), INTEGER
Name, VARCHAR(30) NN
Vorname, VARCHAR(30) NN
Ruecken_Nr, TINYINT(2) NN
GebDat, DATE NN
Geschlecht VARCHAR(8)
NAT (FK), CHAR(3)
Gehalt_monatl, DECIAML(12,2)
ZT_SPIEL_SPIELER
S_ID (FK) (PK*), INTEGER
SP_ID (FK) (PK*), INTEGER
Tore, TINYINT(2) NN
roteKarte, TINYINT(2) NN
gelbeKarte, TINYINT(2) NN
Ort
O_ID (PK), INTEGER AI
PLZ, VARCHAR (10)
Ort, VARCHAR(30)
PLATZ_STADION
P_ID (PK), INTEGER AI
TYP, VARCHAR(30) NN
Zuschauerplaetze, INTEGER
Gr_ID (FK), INTEGER
Strasse, VARCHAR(30)
O_ID (FK), INTEGER
TG_ID (FK), CHAR(1)
TURNIERTAUGLICH
TG (PK), CHAR(1) AI
Bezeichnung, VARCHAR(30)
NATIONALITAET
NAT (PK), CHAR(3) NN
Bezeichnung, VARCHAR(30)

9.2 Datentypen

MySQL unterstützt eine Vielzahl von Datentypen verschiedener Kategorien: numerische Typen, Typen für Datum und Uhrzeit und auch String-Typen (zeichenbasierte Typen). Jedoch werden nicht alle von LibreOffice-Base und Java unterstützt.

9.2.1 Ganzzahlen

Tiny Integer
`TINYINT[(M)] [UNSIGNED] [ZEROFILL]`
Der vorzeichenbehaftete Bereich liegt zwischen -128 und 127.
Der vorzeichenlose Bereich liegt zwischen 0 und 255.
Java→ `byte`

Small Integer
`SMALLINT[(M)] [UNSIGNED] [ZEROFILL]`
Der vorzeichenbehaftete Bereich liegt zwischen -32768 und 32767.
Der vorzeichenlose Bereich liegt zwischen 0 und 65535.
Java→ `short`

Medium Integer
`MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]`
Der vorzeichenbehaftete Bereich liegt zwischen -8388608 und 8388607.
Der vorzeichenlose Bereich liegt zwischen 0 und 16777215.
Java→ (–)

Integer
`INT[(M)] [UNSIGNED] [ZEROFILL]`
`INTEGER[(M)] [UNSIGNED] [ZEROFILL]`
Der vorzeichenbehaftete Bereich liegt zwischen -2147483648 und 2147483647.
Der vorzeichenlose Bereich liegt zwischen 0 und 4294967295.
Java→ `int`

Big Integer
`BIGINT[(M)] [UNSIGNED] [ZEROFILL]`
Der vorzeichenbehaftete Bereich liegt zwischen -9223372036854775808 und 9223372036854775807.
Der vorzeichenlose Bereich liegt zwischen 0 und 18446744073709551615.
Java→ `long`

9.2.2 Fließkommazahlen

Float
`FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]`
Kleine Fließkommazahl (mit einfacher Genauigkeit). Zulässige Werte sind der Bereich zwischen -3.402823466E+38 und -1.175494351E-38, 0 und der Bereich zwischen 1.175494351E-38 und 3.402823466E+38 (nach IEEE-Standard). M ist die Gesamtzahl von Dezimalstellen, D die Anzahl der Stellen hinter dem Dezimalpunkt.
Java→ `float`

Double
`DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]`
`DOUBLE PRECISION[(M,D)] [UNSIGNED] [ZEROFILL]`
`REAL[(M,D)] [UNSIGNED] [ZEROFILL]`
Fließkommazahl normaler Größe (mit doppelter Genauigkeit).
Zulässige Werte:
-1.7976931348623157E+308 bis -2.2250738585072014E-308,
0 und 2.2250738585072014E-308 bis 1.7976931348623157E+308
nach IEEE-Standard
Java→ `double`

Fließkommazahl
`FLOAT(p) [UNSIGNED] [ZEROFILL]`
p steht für die Genauigkeit in Bit, aber MySQL verwendet diesen Wert nur zur Bestimmung, ob als Datentyp für das Ergebnis FLOAT ($0 \leq p \leq 24$) oder DOUBLE (25 $\leq$p $\leq$53) zugewiesen werden soll.
Java→ (–)

„exakte“ Festkommazahl
`DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]`
`DEC[(M[,D])] [UNSIGNED] [ZEROFILL]`
`NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]`
`FIXED[(M[,D])] [UNSIGNED] [ZEROFILL]`
M ist die Gesamtzahl von Dezimalstellen (Genauigkeit), D die Anzahl der Stellen hinter dem Dezimalpunkt. ($M \leq 65$ und $D \leq 30$)
Java→ `BigDecimal`

9.2.3 Text

Text (fix)
`[NATIONAL] CHAR(M) [BINARY | ASCII | UNICODE]`
`NCHAR(M) [BINARY | ASCII | UNICODE]`
Eine Zeichenkette fester Länge, die bei der Speicherung mit Leerzeichen auf die angegebene Länge aufgefüllt wird. M steht für die Spaltenlänge. M liegt in einem Bereich zwischen 0 und 255 Zeichen. `NATIONAL` bzw. `NCHAR` legt fest, dass eine CHAR-Spalte einen bestimmten vordefinierten Zeichensatz verwendet. Ansonsten wird der Zeichensatz der Tabelle bzw. Datenbank verwendet. `BINARY` legt eine binären Sortierung fest, so dass der Vergleich auf numerischen Zeichenwerten erfolgt. `ASCII` sortiert nach `latin1`, `UNICODE` nach der Unicodetabelle. `CHAR BYTE` ist ein Alias auf `BINARY`.
Java→ `~String`

Text
`[NATIONAL] VARCHAR(M) [BINARY]`
`NVARCHAR(M) [BINARY]`
`TEXT[(M)]`
Eine Zeichenkette variabler Länge. ($M \leq 65.535$)
Java→ `String`

Text
`MEDIUMTEXT`
Eine `TEXT`-Spalte mit einer Maximallänge von $16.777.215 = 2^{24-1}$ Zeichen.

Memo
`LONGTEXT`
`LONGVARCHAR`
Eine `TEXT`-Spalte mit einer Maximallänge von $4.294.967.295 = 2^{32-1}$ (4 GiB) Zeichen.

Aufzählung
`ENUM('value1','value2',...)`
Es handelt sich dabei um ein Zeichenketten-Objekt, das aus der Liste der Werte 'value1', 'value2', ..., NULL und einem speziellen Fehlerwert ” genau einen Wert auswählt (max. 65.535 Einträge). ENUM-Werte werden intern als Integers dargestellt.
java→ `enum`

Menge
`SET('value1','value2',...)`
Es handelt sich um ein Zeichenketten-Objekt, dass null oder mehr Werte haben kann, die jeweils aus der Werteliste 'value1', 'value2', ... stammen (max. 64 Einträge). `SET`-Werte werden intern als Integers dargestellt.

9.2.4 Datum und Zeit

Datum
`DATE`
Der unterstützte Bereich liegt zwischen '1000-01-01' und '9999-12-31' (Format 'YYYY-MM-DD').
Java→ `Date`

Datum/Zeit
`DATETIME`
Kombination aus Datum und Uhrzeit (Format 'YYYY-MM-DD HH:MM:SS').
Java→ `Date`

Zeit
`TIME`
Eine Zeitangabe (Format 'HH:MM:SS')
Java→ `Time`

Jahr
`YEAR[(2|4)]`
Ein Jahr im zwei- oder vierstelligen Format.

Zeitstempel
`TIMESTAMP[(M)]`
Der Bereich liegt zwischen '1970-01-01 00:00:00' und einem Zeitpunkt irgendwann im Jahr 2037.
Java→ `Timestamp`

9.2.5 Sonstige

Bit
`BIT[(M)]`
M gibt die Anzahl von Bits pro Wert in einem Bereich zwischen 1 und 64 an.

Ja/Nein
`BOOL`
`BOOLEAN`
`TINYINT(1)`
Der Wert Null wird als falsch ausgewertet, Werte ungleich null als wahr.
Java→ `boolean`

Objekte
`TINYBLOB`
Binary Large Objects (BLOBs) sind binäre Objekte wie Bild- oder Audiodateien. Eine BLOB-Spalte mit einer Maximallänge von $255 = 2^{8-1}$ Byte.
Java→ Array

Objekte
`BLOB[(M)]`
Eine BLOB-Spalte mit einer Maximallänge von $65.535 = 2^{16-1}$ Byte.
Java→ Array

Objekte
`MEDIUMBLOB`
Eine BLOB-Spalte mit einer Maximallänge von $16.777.215 = 2^{24-1}$ Byte.

Objekte
`LONGBLOB`
Eine BLOB-Spalte mit einer Maximallänge von $4.294.967.295 = 2^{32-1} = 4\,\text{GiB}$ Byte.

Binärfeld (fix)
`BINARY(M)`
Java→ Array

Binärfeld
`VARBINARY(M)`
Java→ Array

9.3 Tools

Im MySQL-Paket sind einige Tools enthalten, hier ein kurze Beschreibung der wichtigsten davon.

9.3.1 mysql

Der MySQL-Client dient dazu, über die Konsole SQL-Statements an den MySQL-Server zu schicken und das Ergebnis entsprechend den Einstellungen auszugeben.

```
mysql [Parameter] [database]
```

Die wichtigsten Parameter für den MySQL-Client sind:

`-u <user>`	mit welcher Kennung soll gearbeitet werden
`-p[pw]`	Aufforderung, das PW einzugeben, bzw. optional gleich in den Aufruf mit einzubinden
`-h <host>`	gibt an, auf welchem Server MySQL installiert ist
`-P <port>`	gibt an, welcher Port verwendet werden soll (Standard: 3306)
`-C`	benutzte Kompression bei der Übertragung Client - Server
`-e '<command>'`	Kommando ausführen und dann beenden
`-H`	produziere als Ausgabeformat HTML
`-X`	produziere als Ausgabeformat XML
`-t`	Ausgabe im Tabellenformat

Beispiel

```
mysql -u test -X -e 'SELECT * FROM TBEITRAG' BEISPIEL
```

```
<?xml version="1.0"?>

<resultset statement="SELECT * FROM TBEITRAG
" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
        <field name="ID">E</field>
        <field name="Bezeichnung">Erwachsener</field>
        <field name="Beitrag">100.00</field>
  </row>

  <row>
        <field name="ID">F</field>
        <field name="Bezeichnung">Familienbeitrag</field>
        <field name="Beitrag">200.00</field>
  </row>

  <row>
        <field name="ID">K</field>
        <field name="Bezeichnung">Kind / Student</field>
        <field name="Beitrag">60.00</field>
  </row>

  <row>
        <field name="ID">M</field>
        <field name="Bezeichnung">Familienmitglied</field>
        <field name="Beitrag">0.00</field>
  </row>
</resultset>
```

9.3.2 mysqladmin

Der MySQL-Adminclient dient zur Durchführung administrativer Aufgaben. Man kann damit den Status und die Einstellungen des Servers überprüfen, Datenbanken erstellen oder löschen.

```
mysqladmin [Parameter] command [command-arg] [command [command-arg]] ...
```

Die wichtigsten Parameter sind dabei:

`create`	erstellt eine neue Datenbank
`drop`	löscht eine Datenbank
`extended-status`	zeigt die Serverstatusvariablen an
`flush-hosts`	synchronisiert alle Daten in den Host-Cache
`flush-logs`	schreibt alle Logs auf Festplatte
`flush-privileges`	lädt die Rechte-Tabelle (GRANT) neu
`password new-pw`	ändert das Passwort
`ping`	überprüft, ob der Server ausgeführt wird
`refresh`	synchronisiert alle Tabellen und schließt und öffnet die Logdateien
`shutdown`	der Server wird heruntergefahren und beendet
`status`	gibt eine Meldung über den Serverstatus aus
`variables`	zeigt die Serversystemvariablen an

Beispiel

```
mysqladmin ping
```

```
mysqld is alive
```

9.3.3 mysqlcheck

Das MySQL-Checkprogramm prüft, repariert, optimiert und analysiert Tabellen.

```
mysqlcheck [Parameter] db_name [tables]
```

Die wichtigsten Parameter sind dabei:

`--all-databases`	überprüft alle Tabellen
`--analyze`	analysiert die Tabellen
`--auto-repair`	Beschädigungen werden automatisch zu reparieren versucht
`--check`	überprüft die Tabellen auf Fehler
`--databases`	verarbeitet alle Tabellen der angegebenen Datenbank
`--extended`	es wird auf 100% Konsistenz der Datenbank geprüft (dauert sehr lange!)
`--optimize`	optimiert die Tabellen
`--repair`	repariert fast alle Fehler

Beispiel

```
mysqlcheck -u test BEISPIEL
```

```
BEISPIEL.ARTIKEL                                    OK
BEISPIEL.ARTIKEL_TR                                 OK
BEISPIEL.DATETEST                                   OK
BEISPIEL.KUNDE                                      OK
BEISPIEL.LIEFER                                     OK
BEISPIEL.TBEITRAG                                   OK
BEISPIEL.TMITGLIED                                  OK
```

9.3.4 mysqldump

Der MySQL-Dump-Client ist ein Sicherungsprogramm, welches einen Speicherauszug einer Datenbank mit entsprechenden SQL-Statements erstellt.

```
mysqldump [Parameter] db_name [tables]
```

Die wichtigsten Parameter sind dabei:

`--add-drop-database`	fügt ein `DROP DATABASE`-Statement vor jede `CREATE DATABASE`-Anweisung ein
`--add-drop-table`	fügt ein `DROP TABLE`-Statement vor jede `CREATE TABLE`-Anweisung ein
`--complete-insert`	verwendet vollständige `INSERT`-Statements (mit Spaltennamen)
`--create-options`	fügt alle MySQL-spezifischen Tabellenoptionen in die `CREATE TABLE`-Anweisungen ein
`--hex-blob`	erstellt den Speicherauszug unter Verwendung der Hexadezimalnotation für `BINARY`, `VARBINARY`, `BLOB` und `BIT`
`--routines`	gibt auch gespeicherte Routinen (Funktionen und Prozeduren) aus
`--triggers`	gibt auch gespeicherte Trigger aus
`--xml`	die Ausgabe erfolgt im XML-Format

Beispiel

Hier wird die Ausgabe in eine Datei umgeleitet und nur die ersten 40 Zeilen werden angezeigt.

```
mysqldump -u test BEISPIEL >dump.txt ; head -n 40 dump.txt
```

```
-- MySQL dump 10.13  Distrib 5.5.31, for debian-linux-gnu (x86_64)
--
-- Host: localhost    Database: BEISPIEL
-- ------------------------------------------------------
-- Server version       5.5.31-0ubuntu0.12.10.1

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `ARTIKEL`
--

DROP TABLE IF EXISTS `ARTIKEL`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `ARTIKEL` (
  `AID` int(11) NOT NULL,
  `L_ID` char(1) NOT NULL,
  `Preis` decimal(4,2) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `ARTIKEL`
--

LOCK TABLES `ARTIKEL` WRITE;
/*!40000 ALTER TABLE `ARTIKEL` DISABLE KEYS */;
INSERT INTO `ARTIKEL` VALUES (1,'A',2.45),(1,'B',2.99),(2,'B',2.99),(3,'A',10.50),(4,'B
',1.45),(4,'C',1.35),(5,'C',1.99),(6,'D',2.45),(6,'A',2.45),(7,'D',1.25),(8,'D',21.45)
,(9,'E',2.45),(10,'F',1.25);
/*!40000 ALTER TABLE `ARTIKEL` ENABLE KEYS */;
UNLOCK TABLES;
```

9.3.5 mysqlimport

Der MySQL-Import-Client ist das Zeilenäquivalent zu `LOAD DATA INFILE`.

```
mysqlimport [Parameter] db_name textfile1 [textfile2 ...]
```

Mehr dazu im Abschnitt 4.6.1 auf Seite 37.

9.3.6 mysqlshow

Der MySQL-Show-Client zeigt Informationen zu den vorhandenen Datenbanken etc. an.

```
mysqlshow [Parameter] [db_name [tbl_name [col_name]]]
```

Beispiel

Hier wird die Ausgabe in eine Datei umgeleitet und nur die ersten 40 Zeilen werden angezeigt.

```
mysqlshow -u test BEISPIEL TMITGLIED
```

```
Database: BEISPIEL  Table: TMITGLIED
+------------+-------------+-----------------+------+-----+---------+-------+---------------------------------+---------+
| Field      | Type        | Collation       | Null | Key | Default | Extra | Privileges                      | Comment |
+------------+-------------+-----------------+------+-----+---------+-------+---------------------------------+---------+
| ID         | int(11)     |                 | NO   | PRI |         |       | select,insert,update,references |         |
| Vorname    | varchar(20) | utf8_general_ci | YES  |     |         |       | select,insert,update,references |         |
| Nachname   | varchar(20) | utf8_general_ci | YES  |     |         |       | select,insert,update,references |         |
| Beitrag_ID | char(1)     | utf8_general_ci | YES  |     |         |       | select,insert,update,references |         |
| Betreuer   | int(11)     |                 | YES  |     |         |       | select,insert,update,references |         |
+------------+-------------+-----------------+------+-----+---------+-------+---------------------------------+---------+
```

SQL-Befehle, Funktionen, ...

Symbole

! ... 68
< ... 67
<= ... 67
<=> ... 67
<> ... 67
= ... 67
> ... 67
&& ... 68

A

ABS() ... 78
ACOS() ... 78
ADD ... 25
ADDDATE() ... 80
ADDTIME() ... 80
AFTER ... 125
ALTER ... 25
ALTER FUNCTION ... 118
ALTER PROCEDURE ... 118
ALTER TABLE ... 24, **25**
ALTER VIEW ... 114
AND ... 68
ASCII ... 146
ASCII() ... 71
ASIN() ... 78
ATAN() ... 78
ATAN2() ... 78
AUTO_INCREMENT ... 22
AVG() ... 87

B

BEFORE ... **25**, 125
BEGIN ... **118**, 127
BETWEEN ... 67
BIGINT ... 145
BIN() ... 71
BINARY ... 146, 147
BIT ... 147
BIT_LENGTH() ... 71
BLOB ... 147
BOOL ... 147
BOOLEAN ... 147

C

CALL ... 118
CASCADE ... 24
CASE ... 70, 120
CEIL() ... 78
CEILING() ... 78
CHAR ... 146
CHAR BYTE ... 146
CHAR() ... 71
CHAR_LENGTH() ... 71
CHARACTER_LENGTH() ... 71
CHECK ... 24
CHECKS ... 30
CLOSE ... 122
COALESCE() ... 67
COLUMN ... 25
COMMENT ... 22
COMMIT ... 127
CONCAT() ... 72
CONCAT_WS() ... 72
CONNECT ... 18
CONSTRAINT ... 23
CONV() ... 72
CONVERT_TZ() ... 80
COS() ... 78
COT() ... 78
COUNT() ... 87, 97
CRC32() ... 78
CREATE ... TRIGGER ... 125
CREATE DATABASE ... **18**, 150
CREATE FUNCTION ... 118
CREATE INDEX ... 117
CREATE PROCEDURE ... 118
CREATE TABLE ... **22**, 32, 150
CREATE VIEW ... 114
CROSS JOIN ... 63
CURDATE() ... 80
CURRENT_DATE() ... 80
CURRENT_TIME() ... 80
CURRENT_TIMESTAMP() ... 80
CURTIME() ... 80

D

DATE ... 147
DATE() ... 80
DATE_ADD() ... 80
DATE_FORMAT() ... 81
DATE_SUB() ... 80
DATEDIFF() ... 80
DATETIME ... 147
DAY() ... 82
DAYNAME() ... 82
DAYOFMONTH() ... 82
DAYOFWEEK() ... 82
DAYOFYEAR() ... 82
DCL ... 17
DDL ... 17
DEALLOCATE PREPARE ... 124
DEC ... 146
DECIMAL ... 146
DECLARE ... HANDLER ... 122
DEFAULT ... 22, 36
DEGREES() ... 78
DELAYED ... 36
DELETE ... 18
DELIMITER ... 118
DESC ... 23
DESCRIBE ... 23
DISTINCT ... 87
DML ... 17

DO ... 121
DOUBLE ... 145
DOUBLE PRECISION ... 145
DQL ... 17
DROP ... 25
DROP DATABASE ... **18**, 150
DROP FUNCTION ... 118
DROP INDEX ... 117
DROP PREPARE ... 124
DROP PROCEDURE ... 118
DROP TABLE ... **24**, 150
DROP TRIGGER ... 125
DROP VIEW ... 114
DUAL ... 67

E
ELSE ... 70, 120
ELSEIF ... 120
ELT() ... 72
END ... 70, **118**
END CASE ... 120
END IF ... 120
ENGINE ... 23
ENUM ... 146
EXECUTE ... 124
EXP() ... 78

F
FETCH ... 122
FIELD() ... 72
FIXED ... 146
FLOAT ... 145
FLOOR() ... 78
FLUSH PRIVILEGES ... 19
FOREIGN KEY ... 24
FORMAT() ... 72
FROM ... 58
FROM_UNIXTIME() ... 82
FULL JOIN ... 62
FULL OUTER JOIN ... 62

G
GREATEST() ... 67
GROUP BY ... 58, **86**, 97
GROUP_CONCAT() ... 88

H
HANDLER ... 122
HAVING ... **58**, 88
HEX() ... 73
HIGH_PRIORITY ... 36
HOUR() ... 82

I
IF ... 120
IF EXISTS ... 24
IF NOT EXISTS ... 22
IF() ... 70
IFNULL() ... 70
IGNORE ... 36, 38
IN ... 67
INNER JOIN ... 60
INSERT ... 18, **36**, 38, 150
INSERT ... SELECT ... 36
INSERT ... SET ... 36
INSERT ... VALUES ... 36
INSERT() ... 73
INSTR() ... 73
INT ... 145
INTEGER ... 145
INTERVALL() ... 67
INTO ... 119
IS ... 67
IS NOT ... 67
IS NOT NULL ... 67
IS NULL ... 67
ISNULL() ... 67
ITERATE ... 120, 121

K
KEY ... 22

L
LANGUAGE SQL ... 118
LAST_DAY() ... 82
LAST_INSERT_ID() ... 22
LCASE() ... 73
LEAST() ... 67
LEAVE ... 120, 121
LEFT JOIN ... 61
LEFT OUTER JOIN ... 61
LEFT() ... 75
LENGTH() ... 71
LIKE ... 77
LIMIT ... 38, **58**, 87, 92
LN() ... 79
LOAD DATA INFILE ... 37
LOCATE() ... 73
LOG() ... 79
LOG10() ... 79
LOG2() ... 79
LONGBLOB ... 147
LONGTEXT ... 146
LONGVARCHAR ... 146
LOOP ... 120
LOW_PRIORITY ... 36, 38
LOWER() ... 73
LPAD() ... 74
LTRIM() ... 74

M
MAX() ... 88
MEDIUMBLOB ... 147
MEDIUMINT ... 145
MEDIUMTEXT ... 146
MID() ... 75
MIN() ... 88
MINUTE() ... 82
MOD() ... 79
MONTH() ... 82
MONTHNAME() ... 82

N
NCHAR ... 146
NOT ... 68
NOT BETWEEN ... 67
NOT IN ... 67
NOT NULL ... **22**, 32
NOW() ... 80
NULL ... 22
NULLIF() ... 70
NUMERIC ... 146
NVARCHAR ... 146

O
ON DELETE ... 24
ON DUPLICATE KEY UPDATE ... 36
ON UPDATE ... 24
OPEN ... 122
OR ... 68
ORDER BY ... 38, **58**, 88

P
PERIOD_ADD() ... 82
PERIOD_DIFF() ... 82
PI() ... 79
POW() ... 79
POWER() ... 79
PRIMARY KEY ... 22, 24

Q
QUARTER() ... 82
QUOTE() ... 74

R
RADIANS() ... 79
RAND() ... 79
REAL ... 145
REFERENCES ... 24
REGEXP ... 77
RENAME TO ... 25
REPEAT ... 120
REPEAT() ... 74
REPLACE() ... 75
RESTRICT ... 24
RESTRICT CASCADE ... 24
REVERSE() ... 75
RIGHT JOIN ... 61
RIGHT OUTER JOIN ... 61
RIGHT() ... 75
ROLLBACK ... 127
ROUND() ... 79
RPAD() ... 74
RTRIM() ... 74

S
SECOND() ... 82
SELECT ... 58
SEPARATOR ... 88
SET ... **30**, 36, 38, 119, 146
SET AUTOCOMMIT ... 127
SET DEFAULT ... 24, 25
SET NULL ... 24, 25
SHOW CREATE TABLE ... 23
SHOW CREATE VIEW ... 114
SHOW ENGINES ... 23
SHOW FUNCTION CODE ... 118
SHOW FUNCTION STATUS ... 118
SHOW INDEX ... 117
SHOW PROCEDURE CODE ... 118
SHOW PROCEDURE STATUS ... 118
SHOW TRIGGERS ... 125
SIGN() ... 79
SIN() ... 78
SMALLINT ... 145
SOUNDEX() ... 76
SOUNDS LIKE ... 76
SPACE() ... 75
SQRT() ... 79
START TRANSACTION ... 127
STR_TO_DATE() ... 81
STRCMP() ... 77
SUBDATE() ... 80
SUBSTRING() ... 75
SUM() ... 88

T
TAN() ... 78
TCL ... 17
TEMPORARY ... 22
TEXT ... 146
THEN ... 70
TIME ... 147
TIME() ... 83
TIME_FORMAT() ... 83
TIME_TO_SEC() ... 83
TIMEDIFF() ... 83
TIMESTAMP ... 147
TINYINT ... 145
TO_DAYS() ... 83
TRIM() ... 74
TRUNCATE() ... 79

U
UCASE() ... 73
UNHEX() ... 73
UNICODE ... 146
UNION ... 65
UNIQUE KEY ... 22
UNIX_TIMESTAMP() ... 83
UNSIGNED ... 145
UNTIL ... 120
UPDATE ... 36, 38
UPPER() ... 73
USING ... 71, 124
UTC_DATE() ... 83
UTC_TIME() ... 83
UTC_TIMESTAMP() ... 83

V
VALUES ... 36
VARBINARY ... 147
VARCHAR ... 146

W
WEEK() ... 82
WEEKDAY() ... 83
WHEN ... 70, 120
WHERE ... 38, 58
WHILE ... 121

X
XOR ... 68

Y
YEAR ... 147
YEARWEEK() ... 83

Z
ZEROFILL ... 145

Stichwortverzeichnis

A
Abfrage erstellen
  mit Base ... 94
  mit SQL ... 58
ACID ... 127
Aggregatsfunktionen ... 86
AKID ... 127
Alias ... 95
Alles-oder-nichts-Eigenschaft ... 127
ANSI ... 16
Anzahl ... 97
Arkuskosinus ... 78
Arkussinus ... 78
Arkustangens ... 78
Atomarität ... 127
AutoIncrement ... 42
AutoWert ... 40, 42

B
Bedingungsdefinitionen ... 23
Benutzervariable ... 30
Berichte ... 99
Berkeley ... 16
Binary Large Objects ... 147

C
case sensitive ... 13
Connector/J ... 13
Constraints ... 23
CRC ... 78
CSV ... 37
Cursor ... 122

D
Datenbank ... 9
  Modelle ... 27
  Modellierung ... 5
  Theorie ... 5, 9
Datencontainer ... 10
Datensatzabfrage ... 89
Datentypen ... 145
Datum ... **80**, 147
  Alter berechnen ... 84
Dauerhaftigkeit ... 127
DBA ... 27
DBMS ... 10, 11
Dubletten ... 88

E
Encoding ... 18
Entity Versioning ... 126
Entwurfsansicht ... 32
Envas ... 126
Equivalent Join ... 60

F
Fachinformatiker
  Anwendungsentwicklung ... 5
  Systemintegration ... 5
Feldeigenschaft ... 32
Fließkommazahlen ... 145
Fremdschlüssel ... **24**, 30
  definieren ... 25
  Überprüfung abschalten ... 24, 30
Funktionen ... 67, 118
  Ablaufsteuerungsfunktionen ... 70
  Datums- und Zeitfunktionen ... 80
  Logische Operatoren ... 68
  Numerische Funktionen ... 78
  String-Funktionen ... 71
  String-Vergleichsfunktionen ... 77
  Vergleichsoperatoren ... 67

G
Ganzzahlen ... 145
Gruppierung ... 97

H
Historisierung ... 126
HSQLDB ... 9, 11

I
Identifizierungsschlüssel ... 32
Index ... 117
Informatikkaufmann/-frau ... 5
Integrität ... 127
ISO ... 16
Isolation ... 127
IT–Systemelektroniker/-in ... 5
IT–Systemkaufmann/-frau ... 5
IT-Berufe ... 5

J
JDBC ... 11
  com.mysql.jdbc.Driver ... 13
  MySQL-Treiber ... 12
  Treiber ... 11
JOIN ... *siehe* SQL–JOIN
Jokerzeichen ... 77

K
Kontrollstruktur ... 120
  CASE ... 120
  IF ... 120
  LOOP ... 120
  REPEAT ... 120
  WHILE ... 121
Kosinus ... 78
Kotangens ... 78

L
LibreOffice
  Base
    Abfragen ... 10
    Arbeitsbereiche ... 10
    Berichte ... 10
    Datenbankverbindung ... 11
    Dateneingabe ... 40
    Datensätze löschen ... 41
    Datensätze ändern ... 41
    Einführung ... 10
    Formulare ... 10
    Formulare erstellen ... 42
    interne DB ... 11
    MySQL ... 12
    Tabellen ... 10
  Calc ... 15
  Writer ... 15
Lizenz ... **2**, 5
  CC-BY-NC-ND ... 5

M
Mehrbenutzermodus ... 11
Mehrfachauswahl ... 120
Multibytezeichen ... 71
MySQL ... 9
  Adminclient ... 149
  Benutzervariable ... 30
  Checkprogramm ... 149
  Client ... 148
  Datensätze aus Textdatei laden ... 37
  Download ... 12
  Engines ... 23
  Fremdschlüssel
    Überprüfung abschalten ... 24, 30
  Handbuch ... 12
  Kommentare
    spezielle Befehle ... 18
  mysqlimport ... 38
  Speichersysteme ... 23
  Systemvariable ... 30
  Tabellen reparieren ... 149
  URL ... 13
  Verbindungsinformationen ... 12
  Workbench ... 27
    Data Modeling ... 27
    Server Administration ... 27
    SQL Development ... 27
  XAMPP ... 12
  Zugriff auf Datei ... 13
  Zugriff über Java ... 13

N
Namensregel ... 22

O
ODBC ... 14
ODF ... 11
Open Database Connectivity ... 14
Operatoren ... 67
Oracle ... 9, 12

P
Parameter ... 119

Prepared Statement 124
Primärschlüssel 22, **24**, 32
  in Base festlegen 32
Prozedure 118
Prüfsummenwert 78

R
RDBMS 9, 21
Referentielle Integrität 24
regex 77
Reguläre Ausdrücke 77

S
Schleife
  abbrechen 120
  LOOP 120
  REPEAT 120
  WHILE 121
  wiederholen 120
SEQUEL 16
Sinus 78
Skalarunterabfrage 89
Spaltenunterabfrage 89
SQL
  Abfrage mit der GUI 94
  Abfrage von Daten 58
  Alias für Tabellen festlegen 60
  Cursor 122
  Datenbank erstellen 18
  DCL 17
  DDL 17
  DML 17
  DQL 17
  Eingabe von Daten 36
  FULL JOIN 60
  Funktionen 118
  Geschichte 16
  geänderte Rechte aktivieren ... 19
  Grundlagen 16
  Index anlegen 117
  INNER JOIN 60
  JOIN 59
    CROSS 63
    FULL 62
    INNER 60
    LEFT 61
    LEFT OUTER 61
    RIGHT 61
    RIGHT OUTER 61
    SELF 64
  Kommentare 18
  LEFT JOIN 60
  OUTER JOIN 60
  Prozeduren 118
  Reihenfolge der Abarbeitung ... 58
  RIGHT JOIN 60
  SELF JOIN 60
  Systemdatenbank 18
  Tabellen
    anlegen 22
    löschen 24
    Namensregel 22
    Temporäre 22
    verknüpfen 59
    ändern 25
  TCL 17
  Transaktion 127
  Trigger 125
  Views anlegen 114
  XML 16
  Ändern von Daten 38
  Übersicht 16
  92 16
  99 16
  2003 16
  2008 16
  2011 16
Statement
  prepared 124
Subselect 89
Suche
  phonetisch 76
Systemvariable 30

T
Tabelle 21
Tabellenunterabfrage 89
Tangens 78
Temporale Datenhaltung 126
Text 146
Transaktion 127
Trigger 125

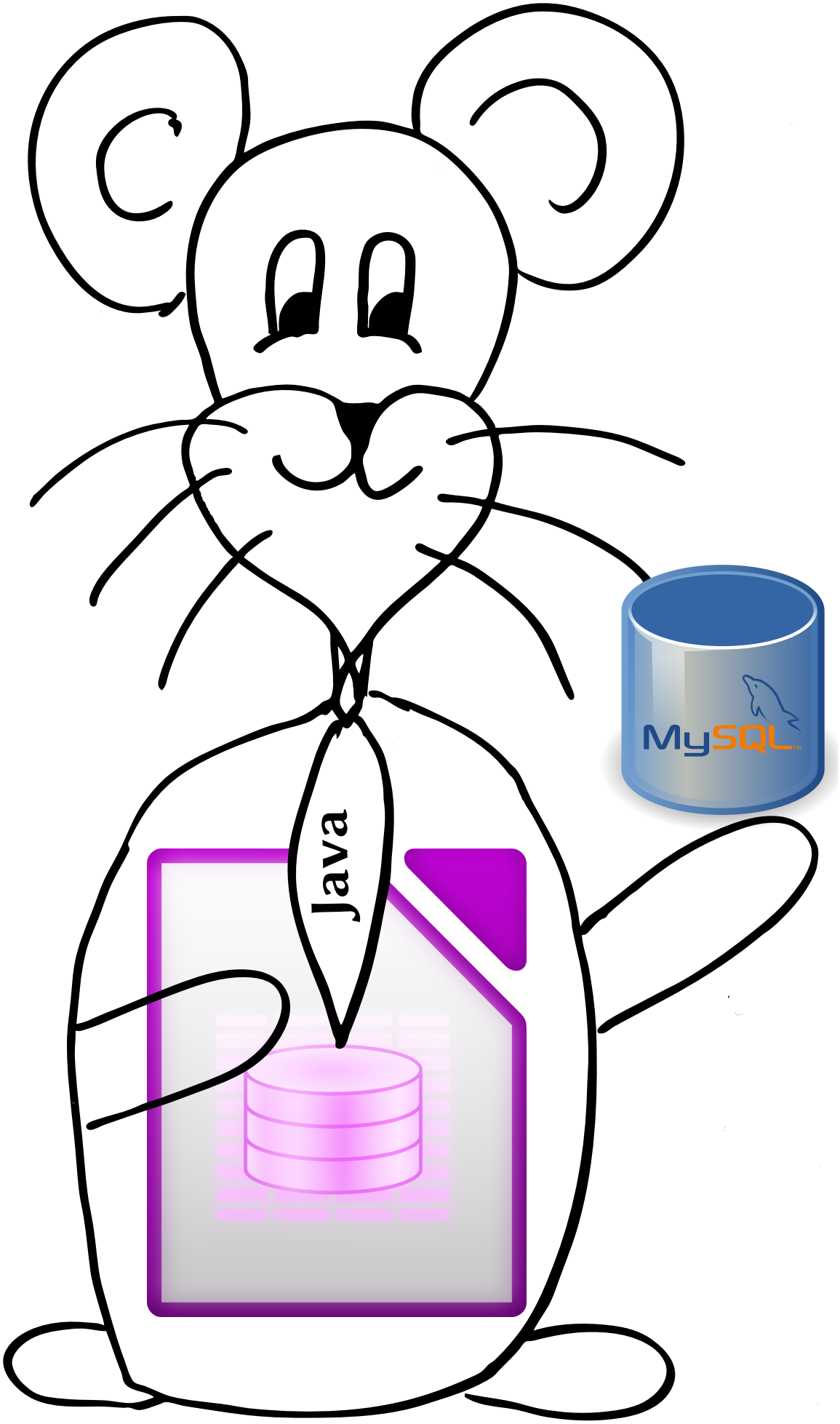
U
UTF-8 18

V
Variable 119, 120
  lokal 120
Verbund 59
View 114

W
Workbench 27

X
XAMPP 12

Z
Zeichenkodierung 18
Zeit **80**, 147


Java
MySQL