HOWTO zur MySQL-Nutzung (Entwickler oder Anwender)

(C) 2006-2018 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>
OSTC Open Source Training and Consulting GmbH
http://www.ostc.de

$Id: mysql-user-HOWTO.txt,v 1.298 2025/02/23 20:14:55 tsbirn Exp $

Dieses Dokument beschreibt den MySQL-Einsatz auf Entwickler/Anwenderseite.

HINWEIS: MySQL-Spezialitäten sind durch "MY!" oder "MY!N.M" gekennzeichnet
(falls sie ab MySQL Version N.M (nicht mehr) verfügbar sind).

HINWEIS: Der Begriff "Datenbank" wird häufig "schwammig" verwendet. MySQL ist
ein "Datenbank-Managementsystem" (DBMS), das die Verwaltung vieler
"Datenbanken" gleichzeitig erlaubt. Jede Datenbank besteht aus
Tabellen und weiteren Datenbank-Objekten, die zur Abbildung eines
konkreten Sachverhalts verwendet werden. Statt der (langen) Begriffe
"Datenbank-Managementsystem" oder (abgekürzt) "Datenbank-System" wird
gerne der kurze Begriff "Datenbank" verwendet. Ein besserer Begriff
für eine eigentliche "Datenbank" ist daher "Schema", so besteht keine
Verwechslungsgefahr, was gemeint ist. Das DBMS Oracle z.B. vermeidet
den Begriff "Datenbank" und verwendet "Schema". Für "Datenbank" wird
in diesem Skript "DB" oder <Db> als Abkürzung verwendet.

INHALTSVERZEICHNIS
==================

1) MySQL-Clients
1a) Zusammenspiel der MySQL-Komponenten
1b) MySQL Web-Client "phpMyAdmin" einrichten und aufrufen
1c) MySQL-Clients: Konfigurations-Dateien
1d) MySQL-Clients: Allgemeine Optionen
1e) MySQL-Clients: Verbindung zum Server (SSL)
1f) Client "mysql" starten (Verbindung/Session)
1g) Client "mysql": Befehlsein- und ausgabe (Prompt, Strg-C, TEE)
1h) Client "mysql" effizient bedienen (USE, History, Strg-L, TAB, Hilfe, SOURCE)
1i) Client "mysql" verlassen
1j) Client "mysql" auf Kommandozeile nutzen (Batch-Modus)
1k) Client "mysql": Spezielle Optionen
1l) Client "mysql": Ausgabeformat-Optionen
1m) Client "mysql": Sonstige Optionen
1n) Client "mysql": Umgebungsvariablen
1o) Client "mysql": Interne Befehle (Escape-Sequenzen)
1p) Client "mysql": Hilfe anzeigen
1q) Client "mysql": Prompt-Definition
1r) Grafische MySQL-Programme (GUI)
2) Syntax
2a) Leerraum und Formatierung
2b) Zeichenketten (Strings) und Quotierung
2c) Escape-Sequenzen
2d) Zahlen
2e) Datenbank-Objekte
2f) Identifier (Bezeichner)
2g) GROSS/kleinschreibung
2h) Kommentare
3) Typische MySQL-Datenbankbefehle (Tutorial)
4) MySQL-Datentypen
4a) Datentyp-Optimierung (Performance/Speicherplatz)
4b) Fixe/Variable Record-Länge (Row-Format)
4c) AUTO_INCREMENT
5) MySQL-Operatoren
6) Boolesche Logik

7) Der Wert NULL
7a) Eigenschaften von NULL
7b) Prüfung auf Wert NULL
7c) Vergleiche mit NULL
7d) Boolesche Logik mit NULL (ternär/dreiwertig)
8) Reguläre Ausdrücke in MySQL (RegEx)
9) MySQL-Funktionen
9a) Aggregatfunktionen und Gruppierung (Aggregation)
10) Schlüssel (Key) und Index
10a) Eigenschaften
10b) Erstellen/entfernen
10c) FULLTEXT-Index
10d) SPATIAL-Index
10e) Index-Nutzung
10f) Index-Optimierung
10g) Fremdschlüssel (Foreign Keys) und Referenzielle Integrität
11) Joins
12) Mengen-Operationen
13) Unterabfragen (Subqueries/Subselect)
14) Transaktionen
15) Locking
16) Views (Sichten)
17) Variablen
18) Prepared Statements (Vorbereitete Anweisungen)
19) Stored Routines (Stored Procedures/Functions)
19a) Lokale Variablen
19b) Wertzuweisung an Variable
19c) Ausgabe von Variablen oder Daten
19d) Kontrollstrukturen (Block/Compound Statement)
19e) Kontrollstrukturen (Verzweigungen)
19f) Kontrollstrukturen (Schleifen)
19g) Prozeduren
19h) Funktionen
20) Trigger
21) Condition Handler (Error/Warning)
22) Cursor (Zeiger)
23) Table Handler
24) Events (Ereignisse)
25) Signale
25a) MySQL-Fehlercode / SQL-Status
26) Begrenzungen von MySQL (Limits)
27) MySQL testen

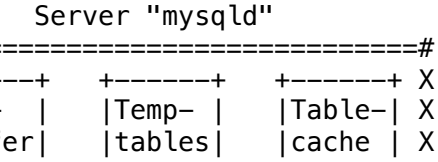
---

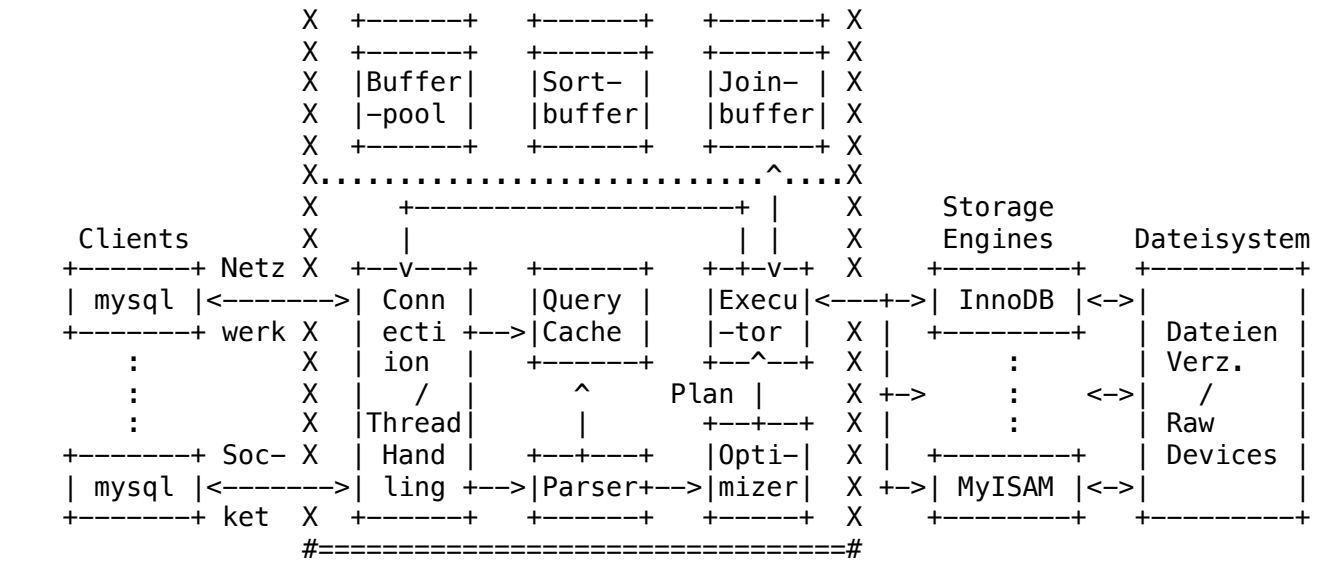
1) MySQL-Clients

1a) Zusammenspiel der MySQL-Komponenten

Architektur:

* N MySQL-Clients + 1 MySQL-Server (auf anderem oder gleichem Rechner)
  + Client: mysql (Kommandoschnittstelle, eine von vielen)
  + Server: mysqld (Daemon, Instanz)
* MySQL-Server-Schichten
  + Oben: Benutzer-Schnittstelle (API, SQL, DB-Management)
  + Mitte: Storage Engine
  + Unten: Dateisystem





Die Verbindung zwischen MySQL-Client und -Server erfolgt per Socket, Named
Pipe, Memory oder TCP/IP (siehe Optionen "-h/--host" und "--protocol").

1b) MySQL Web-Client "phpMyAdmin" einrichten und aufrufen

A) Einrichtung per Paket "phpmyadmin" (Administrator "phpmyadmin"):

```
sudo apt-get install phpmyadmin         # Paket + abhängig. Pakete install.
/etc/phpmyadmin/config.inc.php          # Konfiguration
/usr/share/phpmyadmin/*                 # PHP-Code
/usr/share/doc/phpmyadmin/*             # Dokumentation
http://localhost/phpmyadmin/index.php   # Anmeldung an Web-Oberfläche
```

B) Apache Webserver mit PHP-Unterstützung installieren und phpMyAdmin-Quellcode
kopieren nach:

```
/srv/www/htdocs/phpMyAdmin   # Variante A
/var/www/htdocs/phpMyAdmin   # Variante B
```

Dazu vorher in Konfig-Datei (inc=include)

```
/srv/www/htdocs/phpMyAdmin/config.inc.php      # Variante A
/var/www/htdocs/phpMyAdmin/config.inc.php      # Variante B
```

einen gültigen MySQL-Benutzer mit gültigem Passwort eintragen,
der MySQL-Administrationsrechte (d.h. ALL PRIVILEGES + GRANT OPTION) besitzt:

```
$cfg['Servers'][$i]['auth_type'] = 'config';   # Auth.meth. (config/http/cookie)
$cfg['Servers'][$i]['user']      = 'root';     # MySQL Benutzer (Administrator)
$cfg['Servers'][$i]['password']  = 'GEHEIM';   # MySQL Passwort (wg. 'config')
```

Den Apache-Webserver starten:

```
/etc/init.d/apache2 start   # Temporär (init)
service apache2 start       # Temporär (systemd)
rcapache2 start             # Temporär (OpenSUSE)
chkconfig -a apache2        # Permanent (OpenSUSE)
```

Im Web-Browser dann über folgende URL die Oberfläche "phpMyAdmin" aufrufen
und mit den in "config.inc.php" eingetragenen Benutzer-Daten anmelden:

```
http://localhost/phpMyAdmin/index.php
```

1c) MySQL-Clients: Konfigurations-Dateien

Die Einstellung der MySQL-Variablen und -Optionen stammen aus dem
MySQL-Kommando selbst sowie einigen Konfig-Dateien in der hier angegebenen
Reihenfolge (sofern vorhanden, die Einstellungen der zuletzt angegebenen
überschreiben die Einstellungen der ersten):

N	Quelle (Datei)	Bedeutung	
0a	Client-Programm	Global (zentral, fest eingebaut)	
0b	Server "mysqld"	Global (zentral, fest eingebaut)	
1	/etc/my.cnf	Global (zentral)	
2	/etc/mysql/my.cnf	Global (zentral)	steigender
3	/usr/etc/my.cnf	Global (zentral)	Vorrang
4	~/.my.cnf	Lokal (benutzerbezogen)	
5	Umgebung	Lokal (Umgebungsvariablen)	
6	Kommandozeile	Lokal (direkt beim Kommandoaufruf)	v

HINWEIS: Optionen auf der Kommandozeile haben immer das "letzte Wort"!

Konfig-Dateien sind durch in eckige Klammern gesetzte Programmnamen in
"GRUPPEN" (Abschnitte) eingeteilt. Berücksichtigt werden für den "mysql"-Client
Optionen folgender Gruppen (--> mysql-admin-HOWTO.txt --> 1a) Grundlegendes):

```
[mysql]    # Programmname
[client]   # Programmtyp
```

TIP: Mit folgenden Optionen wird das Einlesen der Konfig-Dateien
verändert bzw. der sich aus den Konfig-Dateien ergebende Default-Wert
der Optionen ausgegeben. Die ersten drei Optionen sind nur als 1. Argument nach
dem Kommandonamen (z.B. "mysql") erlaubt, --print-defaults muss direkt danach
stehen:

Option	Bedeutung
--no-defaults	Keine Konf.dateien lesen
--defaults-file=<Path>	Default-Optionen NUR aus Datei <Path> lesen
--defaults-extra-file=<Path>	Datei <Path> ZUSÄTZL. nach glob. Konf.dat.
--print-defaults	Programmargumente ausgeben und Abbruch

D.h. die in das Server-Programm "mysqld" bzw. das Client-Programm "mysql"
eingebauten Einstellungen erhält man per:

```
mysqld --no-defaults --print-defaults   # Server
mysql  --no-defaults --print-defaults   # Client
```

Die Einstellungen gemäß Konfig.Dateien von Server und Client erhält man per:

```
mysqld --print-defaults   # Server
mysql  --print-defaults   # Client
```

1d) MySQL-Clients: Allgemeine Optionen

Die Optionen der MySQL-Kommandozeilen-Programme sind in Kurzform "-X" sowie in
Langform "--XXX" angebbar, bei der Kurzform ist die GROSS/kleinschreibung
relevant, bei der Langform nicht. ALLE MySQL-Programme kennen folgende Optionen:

Kurzform	Langform	Bedeutung
-v	--verbose	Mehr Info ausgeben (mehrfach erlaubt)
-s	--silent	Weniger Info ausgeben (mehrfach erlaubt)
-V	--version	Versioninformation ausgeben
-?	--help	Hilfe zum Aufruf ausgeben (Syntax, Optionen)

ALLE MySQL-Kommandozeilen-Clients mit einer Verbindung zum MySQL-Server kennen folgende Optionen (legen Art und Weise der Verbindung fest):

Kurzform	Langform	Bedeutung
-h<Host>	--host=<Host>	Ziel-Host (STD: localhost)
-u<User>	--user=<User>	Benutzer festlegen (STD: OS-Anmeldung)
-p[<Pass>]	--password[=<Pass>]	Passwort abfragen bzw. direkt angeben
-D<Db>	--database=<Db>	Datenbank auswählen (STD: keine)
	--protocol=<Proto>	Verb.protokoll (tcp/socket/pipe/memory)
-P<Port>	--port=<Port>	Ziel-Port (STD: 3306)
-S<File>	--socket=<File>	Socket (STD: /var/lib/mysql/mysql.sock bzw. /var/run/mysqld/mysqld.sock)

Bedeutung von "<Proto>" bei Option --protocol:

<Proto>	Verbindung Client <-> Server erzwingen ...	Art
tcp	... per TCP (STD: Port 3306)	lokal+remote
socket	... per Socket (STD: /var/lib/mysql/mysql.sock)	nur lokal
pipe	... per Named Pipe (STD:	
memory	... per Shared Memory	nur lokal

HINWEIS: Standardmäßig wird lokal per "socket" und remote per "tcp" verbunden.

HINWEIS: Bei der Kurzform -X darf der Wert auch per Leerzeichen getrennt von der Option angegeben werden. AUSNAME: Das Passwort <Pass> ist DIREKT NACH der Option "-p" (ohne Leerzeichen dazwischen) anzugeben!

HINWEIS: Ohne Option "-p" wird eine Anmeldung OHNE Passwort versucht.

1e) MySQL-Clients: Verbindung zum Server (SSL)

Die MySQL-Clients versuchen je nach Aufruf auf folgende Arten mit dem MySQL-Server Verbindung aufzunehmen:

* Lokale Verbindung per Socket/Named Pipe/Shared Memory:
  Option "-h/--host" nicht oder mit Wert "localhost/127.0.0.1/::1" angegeben und Option "--protocol" nicht angegeben oder nicht auf Wert "tcp" gesetzt.
  MySQL-Client und -Server müssen dann natürlich auf dem gleichen Host laufen oder per VPN verbunden sein.

* Netzwerk-Verbindung per TCP/IP:
  Option "-h/--host" angegeben und bezeichnet fremden Rechner/fremde IP-Adresse oder Option "--protocol" auf Wert "tcp" gesetzt.
  Auch dann, wenn rein lokale Verbindung möglich wäre!

Für eine per SSL verschlüsselte Verbindung zwischen Client und Server sind folgende Optionen anzugeben:

Option (nur Langform)	Bedeutung
--ssl	SSL für Verbindung wz. Cl. und S. einschalten (automatisch von anderen --ssl-Opt. aktiviert)
--skip-ssl	Abschalten der SSL-Verbindungsverschlüsselung
--ssl-ca=<File>	CA-Datei im PEM-Format (siehe OpenSSL-Doku)
--ssl-capath=<Dir>	CA-Verz. (siehe OpenSSL-Doku)
--ssl-cert=<File>	X509-Zertifikat in PEM-Format
--ssl-cipher=<Typ>	SSL-Verschlüsselungstyp (...)
--ssl-key=<File>	X509-Schlüssel im PEM-Format
--ssl-verify-server-cert	"Common Name" des Servers in seinem Zertifikat mit dem beim Verb.aufbau benutzten Hostnamen vergleichen (STD: aus)

Ob ein Verbindung per SSL verschlüssel werden MUSS, wird pro Benutzer beim Anlegen durch Angabe der Option REQUIRE <SSLOpt> festgelegt:

Option <SSLOpt>	Bedeutung
NONE	Unverschlüss. Verbindung erlaubt (STD), verschl. auch
SSL	Nur SSL-verschlüsselte Verbindungen erlaubt
X509	Gültiges Zertifikat notwendig, Issuer + Subject egal
CIPHER <Spec>	Verschlüsselung und Schlüssel müssen best. Bed. genügen (z.B. CIPHER "EDH-RSA-DES-CBC3-SHA")
ISSUER <String>	Zertifikat muss von CA <String> ausgestellt sein
SUBJECT <String>	Zertifikat muss Subject <String> enthalten

1f) Client "mysql" starten (Verbindung/Session)

Der Client "mysql" bietet eine einfache KOMMANDOZEILENBASIERTE Schnittstelle zum Absetzen von SQL-Anweisungen an einen MySQL-Server und zur Ausgabe ihrer Ergebnisse (MySQL-"Terminal/Monitor"). Trotzdem ist er sehr leistungsfähig, weil MySQL (nahezu) vollständig über SQL-Anweisungen steuerbar ist. Um diese Schnittstelle nutzen zu können, sind allerdings gute Kenntnisse der SQL-Anweisungen und ihrer Syntax notwendig.

Auf der Kommandozeile wird zunächst per Client "mysql" eine VERBINDUNG mit dem MySQL-Server "mysqld" aufgenommen (eine SESSION gestartet), indem eine Anmeldung mit gültigem Benutzernamen + Passwort durchgeführt wird (-u=user, -p=password, -h=host, -D=database, -P=Port, -S=Socket). Anschließend können solange SQL-Anweisungen abgesetzt werden, bis durch Verlassen des "mysql"-Clients die Verbindung zum MySQL-Server "mysqld" wieder beendet wird:

```
mysql -utom -pgeheim
```

Geht die Verbindung zum MySQL-Server verloren (z.B. durch Timeout), dann öffnet "mysql" beim nächsten Kommando AUTOMATISCH eine neue Verbindung (reconnect). Da dies fast unmerklich geschieht und beim Verbindungsabbruch Sitzungsdaten wie lokale Variablen und sonstige Einstellungen verloren gehen, ist dies auch verhinderbar durch:

```
mysql --skip-reconnect ...
```

Typische Aufrufe des "mysql"-Clients:

Kommando	Bedeutung

```
  +------------------------------+-------------------------------------------+
  | mysql                        | Als angemeldeter OS-Benutzer aufrufen     |
  | mysql -utom                  | Benutzer "tom" ohne Passwort (unsinnig!)  |
  | mysql -utom -p               | Benutzer "tom" (Passwort interaktiv eingeb)|
  | mysql -utom -pgeheim         | Passwort "geheim" gleich mitgeben (ungut!)|
  | mysql -utom -pgeheim -htest  | Mit Rechner "test" verbinden (host)       |
  | mysql -utom -pgeheim -P1234  | Mit Port "1234" verbinden (STD: 3306)     |
  | mysql -utom -pgeheim -S/tmp/x| Mit Socket verbinden                      |
  | mysql -utom -pgeheim -Dprod  | Datenbank "prod" auswählen (Variante A)   |
  | mysql -utom -pgeheim prod    | Datenbank "prod" auswählen (Variante B)   |
  +------------------------------+-------------------------------------------+
```

Hilfe zum Aufruf des "mysql"-Client anzeigen (Optionen, Reihenfolge der Konfig-Dateien, Werte von Variablen und Optionen):

```
  mysql --help
  mysql -?
```

Weitere Einstellungen zur Verbindung: !!!

```
  --connect-timeout=<N>      # Sek. bis autom. Verbindungstrenn. (STD: 0=nie)
  --max-allowed-packet=<N>   # Max. Länge transf. Befehle/Ergebnisse (STD: 16MB)
  --net-buffer-length=<N>    # Puffergröße TCP/IP+Socket-Komm. (STD: 16KB)
  --skip-reconnect           # Keine automat. Neuverbindung nach Trennung
```

Durch Setzen des Timeout wird der Client bei längerer Nichtbenutzung automatisch verlassen (STD: 0 = kein Timeout):

TIP: Aliase für "mysql"-Anmeldung + Umschalten auf aktuell relevante Datenbank definieren und in "~/.alias" oder "~/.bash_aliases" ablegen (-D=Standard-DB, -t=ASCII-Rahmen um Ausgaben zeichnen, -e=Befehl ausführen):

```
  alias my="mysql -utom -pgeheim -Dtest -t"       # Interaktiv/Batch per Umlenk.
  alias mye="mysql -utom -pgeheim -Dtest -t -e"   # SQL direkt auf Kommandozeile
```

Aufruf durch:

```
  my                         # Interaktiv-Modus starten
  my < <SqlFile>             # Batch-Datei mit SQL-Anweisungen einlesen
  mye 'SELECT * FROM pers'   # SQL-Anweisung direkt ausführen
```

1g) Client "mysql": Befehlsein- und ausgabe (Prompt, Strg-C, TEE)

Nach der Anmeldung am "mysql-Client" erscheint ein PROMPT folgender Form, der die interaktive Eingabe beliebiger SQL-Anweisungen (Statements) erwartet. Abschluss der Eingabe per <RETURN> übergibt diese dem MySQL-Server zur Ausführung und zeigt das Ergebnis sowie statistische Informationen an.

```
  mysql> ...   # Wartet auf Kommando-Eingabe + <RETURN>
```

HINWEIS: Echte SQL-Anweisungen (werden an MySQL-Server geschickt) sind mit ";" abzuschließen! Client-spezifische Befehle ("USE <Db>" oder "QUIT") dürfen ohne abschließenden ";" geschrieben werden (besser angeben).

Um z.B. alle Datenbanken anzuzeigen, ist folgender Dialog zu führen (die SQL-Anweisung ist hier GROSS geschrieben, GROSS/kleinschreibung ist aber eigentlich egal, siehe auch --> 2g) GROSS/kleinschreibung):

```
  mysql> SHOW DATABASES;<RETURN>     # Prompt + Benutzer-EINGABE (";" notwendig)
  +----------------------------+   # ASCII-Rahmen           (AUSGABE)
  | Database                   |   # Spaltenüberschrift    (AUSGABE)
  +----------------------------+   # ASCII-Rahmen           (AUSGABE)
  | INFORMATION_SCHEMA         |   # 1. Ergebniszeile      (AUSGABE)
  | mysql                      |   # 2. Ergebniszeile      (AUSGABE)
```

```
  | PERFORMANCE_SCHEMA           |   # 3. Ergebniszeile       (AUSGABE)
  | phpmyadmin                   |   # 4. Ergebniszeile       (AUSGABE)
  | ...                          |   # ...                    (AUSGABE)
  +------------------------------+   # ASCII-Rahmen           (AUSGABE)
  15 rows in set (0.01 sec)          # Statistik (15 Ergeb.zeilen + Laufzeit)
                                     # (Leerzeile)
  mysql>                             # Prompt nächste Benutzer-EINGABE
```

oder (leeres Ergebnis):

```
  mysql> SHOW DATABASES LIKE "xyz";  # Prompt + Benutzer-EINGABE + <RETURN>
  Empty set (0.00 sec)               # Statistik (leeres Ergebnis + Laufzeit)
                                     # (Leerzeile)
  mysql>                             # Prompt nächste Benutzer-EINGABE
```

oder (Fehler):

```
  mysql> SHOW TABLES;                          # Prompt + Benutzer-EINGABE
  ERROR 1046 (3D000): No database selected     # Fehlermeldung
  mysql>                                       # Prompt nächste Benutzer-EINGABE
```

Fehlermeldungen, Warnungen und Bemerkungen (Notes) nach einer SQL-Anweisung ausgeben (bezieht sich immer auf die vorhergehende SQL-Anweisung):

```
  SHOW ERRORS;     # Nur Fehler ausgeben
  SHOW WARNINGS;   # Fehler, Warnungen, Bemerkungen (Notes) ausgeben
```

Nach dem <RETURN> sendet der Client "mysql" das Kommando an den MySQL-Server "mysqld", mit dem die Verbindung besteht, wartet auf das Ergebnis und gibt es auf dem Bildschirm aus. Die Ausgabe erfolgt in tabellarischer Form (Zeilen + Spalten). Die erste Zeile enthält die Spaltenüberschrift. Am Ende wird die Anzahl der erhaltenen Datensätze (N rows) und die Dauer der Abfrage (N.NN sec) ausgegeben. Anschließend wird erneut der Prompt angezeigt und auf die Eingabe des nächsten Kommandos gewartet. Hier noch ein Beispiel:

```
  mysql> SELECT SIN(PI()/2), (4+1)*5;   # Prompt + Benutzer-Eingabe
  +-------------+---------+             # Ausgabe (ASCII-Rahmen)
  | SIN(PI()/2) | (4+1)*5 |             # Ausgabe (Überschrift)
  +-------------+---------+             # Ausgabe (ASCII-Rahmen)
  |           1 |      25 |             # Ausgabe (1. Datensatz)
  +-------------+---------+             # Ausgabe (ASCII-Rahmen)
  1 row in set (0.00 sec)               # Statistik (eine Ergeb.zl. + Laufzeit)
                                        # (Leerzeile)
  mysql>                                # Prompt nächste Benutzer-EINGABE
```

Fortsetzungs-Prompt "->": SQL-Kmdos müssen nicht auf einer Zeile stehen, sie können mehrere Zeilen verteilt sein. MySQL erkennt das Kommando-Ende am abschließenden ";" und gibt bis zum abschließenden ";" den Fortsetzungs-Prompt "->" aus:

```
   mysql> SELECT user,          # Kmdo-Beginn
       ->        password,      # Kmdo-Teil
       ->        host           # Kmdo-Teil
       -> FROM mysql.user       # Kmdo-Teil
       -> ;                     # Kmdo-Ende ";"
```

Unvollständige SQL-Anweisungen (z.B. wird der ";" gerne vergessen) führen zur Anzeige eines der folgenden Fortsetzungs-Prompts (dann einfach restliche Kommandoteile oder fehlendes ";" tippen und <RETURN> drücken):

```
  +--------+------------+-----------------------------------------+
  | Prompt | Element    | Bedeutung                               |
  +--------+------------+-----------------------------------------+
  | ->     | Kommando   | Unvollständig bzw. ";" vergessen        |
  | ">     | String     | "..."   begonnen aber noch nicht beendet |
  | '>     | String     | '...'   begonnen aber noch nicht beendet |
```

```
  | `>     | Bezeichner | `...`   begonnen aber noch nicht beendet |
  | /*>    | Kommentar  | /*...*/ begonnen aber noch nicht beendet |
  +--------+------------+-----------------------------------------+
```

Ausgaben des "mysql"-Client in einer Datei <OutFile> aufzeichnen:

```
  mysql ... --tee=<OutFile>
```

Die Ausgabe auf Datei kann auch interaktiv im "mysql"-Client aktiviert und wieder deaktiviert werden durch:

```
  mysql> TEE <OutFile>   # Aufzeichnung starten
  mysql> ...             # Ergebnisse aufzeichnen
  mysql> NOTEE           # Aufzeichnung beenden
  mysql> ...             # Ergebnisse NICHT aufzeichnen
  mysql> \T <OutFile>    # Aufzeichnung starten (Escape-Sequenz)
  mysql> ...             # Ergebnisse aufzeichnen
  mysql> \t              # Aufzeichnung beenden (Escape-Sequenz)
```

HINWEIS: Vorteil dieser Art der Ausgabesteuerung ist, dass die Ausgabe sowohl auf dem Bildschirm als auch in eine Datei <OutFile> erfolgt.

1h) Client "mysql" effizient bedienen (USE, History, Strg-L, TAB, Hilfe, SOURCE)

Standard-Datenbank per USE: Als 1. Kommando immer folgendes verwenden, um eine der vorhandenen Datenbanken als DEFAULT/STANDARD-DATENBANK auszuwählen. Die Objekte darin sind dann leicht über ihre Namen (ohne Qualifizierung mit dem Datenbanknamen) erreichbar. Der ";" am Ende darf bei "USE" weggelassen werden (besser erst gar nicht daran gewöhnen!):

```
  USE <Db>    # Variante A
  USE <Db>;   # Variante B (besser)
```

Name der Standard-Datenbank ausgeben ("NULL" falls noch keine ausgewählt):

```
  SELECT DATABASE();   # Variante A
  SELECT SCHEMA();     # Variante B
```

MySQL-History: Alte Befehle können per Cursortasten durchgeblättert, editiert und erneut mit <RETURN> ausgeführt werden. Mehrzeilig Befehle werden dabei zu einer (langen) Zeile zusammengezogen, die evtl. schwer zu editieren ist. Die Befehle stehen auch nach Verlassen und erneutem Aufrufen des Clients "mysql" wieder zur Verfügung, da sie PRO BENUTZER in seinem Heimatverz. in folgender Datei gespeichert werden (d.h. root <-> normaler Benutzer getrennt):

```
  ~/.mysql_history
```

Da ALLE im "mysql"-Client eingegebenen Befehle in der MySQL-History landen, können darin evtl. Passworte oder andere sicherheitsrelevante Daten stehen. Die Zugriffsrechte der Datei sollten daher 600 lauten ("rw" nur für Besitzer). Um den Datei-Inhalt schnell zu leeren, folgendes Kommando absetzen:

```
  > ~/.mysql_history   # Leere Datei umlenken
```

Um die MySQL-History NICHT aufzuzeichen, folgendes einstellen:

```
  export MYSQL_HISTFILE="/dev/null"                          # Variante A
  rm ~/.mysql_histfile && ln -s /dev/null ~/.mysql_histfile  # Variante B
```

Befehle oder Befehlsteile können auch per MAUS (linke Taste selektiert + mittlere Taste kopiert) in die Kommandozeile des "mysql"-Clients kopiert werden. Wird bis zum rechten Terminalrand markiert, ist der Befehlsabschluss <RETURN> gleich mit in der Kopie enthalten (unter Linux).

TAB-Completion: Datenbank-, Tabellen- und Spalten-Namen (aber keine anderen SQL-Syntax-Elemente) können per <TAB>-Taste AUTOMATISCH VERVOLLSTÄNDIGT werden (Tab-Completion). Dazu per "-D <Db>" oder "USE <Db>;" eine Default/Standard-Datenbank auswählen und beim Aufruf von "mysql" die Option "--auto-rehash" angeben oder in "mysql" den Befehl "REHASH" eingeben (verlangsamt den Start etwas), um den Datenbankinhalt einzulesen.

```
mysql -utom -pgeheim --auto-rehash mysql      # Variante A
mysql -utom -pgeheim --auto-rehash -D mysql   # Variante B
mysql -utom -pgeheim                          # Variante C
  USE mysql;                                  # Variante C
  REHASH                                      # Variante C
```

Bildschirm LEEREN im "mysql"-Client (Prompt "mysql> " steht dann ganz oben):

```
<Strg-L>   # TIP: L=Leeren (eigentlich Steuerzeichen FF=FormFeed ;-)
```

HILFE zu Kommandos im "mysql"-Client durch folgende Befehle anzeigen (die Hilfetexte stehen direkt in der Datenbank in den Tabellen "mysql.help_..."):

Befehl	Bedeutung
HELP	Liste der internen "mysql"-Client-Befehle ausgeben
HELP HELP	Hilfe zur Hilfe selbst ausgeben
HELP <SqlCmd>...	Hilfe zu SQL-Anweisung ausgeben (eindeut. Präfix langt)
HELP CONTENTS	Liste der Hilfe-Themen ausgeben
HELP <Topic>	Hilfe zu einem Thema aus Themenliste ausgeben

TIP: UPDATE- und DELETE-Anweisungen gegen versehentliches Weglassen der WHERE- oder LIMIT-Klausel schützen (d.h. versehentliches Ändern/Löschen ALLER Datensätze verhindern) und Anzahl der Ergebniszeilen bzw. Verknüpfungszeilen beschränken:

	Langform	Bedeutung
-U	--safe-updates	UPDATE- und DELETE-Anweisungen gegen
	--i-am-a-dummy	Weglassen von WHERE und LIMIT schützen
	--select-limit=<N>	Abbruch Erg.ausgabe ab N Zl. (STD: 1000)
	--max-join-size=<N>	Abbruch Erg.ausg. ab N Zl.komb. (STD: 1Mio)

TIP: SQL-Anweisungen aus externer Datei einlesen (Include):

```
SOURCE <SqlFile>;              # Kein "..." um <SqlFile>, vollst. Pfad!
```

1i) Client "mysql" verlassen (Strg-C)

Durch folgende Eingaben kann der "mysql"-Client verlassen werden:

Befehl	Bedeutung
QUIT	Quit-Befehl
EXIT	Exit-Befehl
\q	Quit (Escape-Sequenz)
Strg-C	Abbruch (Cancel, deaktivieren mit "--sigint-ignore")
Strg-D	D=Dateiende (nur Linux)
Strg-Z	Z=Dateiende (nur Windows)

Drücken von Strg-C zum Abbrechen der aktuellen SQL-Anweisung sollte man sich abgewöhnen. In der Shell-Kommandozeile ist man daran gewöhnt, dass die Shell dabei nicht abgebrochen und man nicht abgemeldet wird. Aus dem Client "mysql" fliegt man hingegen sofort raus und muss sich mühsam wieder anmelden.

Ruft man den "mysql"-Client mit Option "--sigint-ignore" auf, so beendet Strg-C nur noch den aktuellen Befehl, bricht aber "mysql" nicht mehr ab:

```
mysql -utom -pgeheim --sigint-ignore -Dtest
```

TIP: Am besten in Konfig-Datei "~/.my.cnf" eintragen.

1j) Client "mysql" auf Kommandozeile nutzen (Batch-Modus)

Neben der interaktiven Nutzung kann der Client "mysql" auch im Batch-Modus benutzt werden. Dazu die SQL-Anweisung direkt nach der Option "-e/--execute" angeben. Mehrere SQL-Anweisungen durch ";" trennen, der letzte ";" darf fehlen (d.h. einzelne SQL-Anweisung muss nicht mit ";" abgeschlossen werden):

```
mysql -utom -pgeheim -e "USE mysql; SELECT user, host, password FROM user"
```

Alternativ können die SQL-Anweisungen per Eingabe-Umleitung aus einer Datei <SqlFile> eingelesen werden (analog dem Kommando "SOURCE" im "mysql"-Client). Ebenso können Ergebnisse von SQL-Anweisungen statt auf dem Bildschirm per Ausgabe-Umleitung auch auf eine Datei <OutFile> ausgegeben werden.

Kommando	Bedeutung
`mysql -u<User> -p -e "<SqlCmd>"`	execute
`... -e "USE first; SELECT * FROM pers"`	execute

`... -e "USE first;`
`INSERT INTO pers VALUES`
`(100, 'Hans', 'Dampf');`
`INSERT INTO pers VALUES`
`(101, 'Hänschen', 'Klein')"`
	execute (mehrere Kommandos durch ";" trennen!)
	(Statement in "..." setzen, Strings darin in '...')

`mysql -u<User> -p < <SqlFile>`	Eingabe von Datei <SqlFile>
`..... -u<User> -p -D<Db> < <SqlFile>`	(USE <Db> nicht vergessen!)
`..... -u<User> -p -D<Db> <<EOF` <Stmt>; ... EOF	Here-Dokument von EOF SQL-Statement ... bis EOF (End Of File)
`mysql ... -e "SOURCE <SqlFile>"`	Eingabe von Datei <SqlFile>
`mysql ... > <OutFile>`	Ausgabe auf Datei <Outfile>

Die Eingabedatei <SqlFile> wird oft manuell erstellt und kann neben der Definition von Tabellen auch Daten zum Füllen der Tabellen enthalten. Häufig entsteht die Eingabedatei auch durch einen teilweisen/vollständigen "Dump" einer MySQL-Datenbank per Kommando "mysqldump". Es erzeugt SQL-Anweisungen, die mit dem "mysql"-Client ausgeführt werden können, um Struktur + Inhalt der Datenbank wieder zu erstellen:

```
mysqldump -utom -pgeheim test > test-dump.sql   # Dump von DB "test" erzeugen
mysql -utom -pgeheim -e "DROP DATABASE test"    # DB "test" löschen (Inhalt!)
mysql -utom -pgeheim -e "CREATE DATABASE test"  # DB "test" anlegen (leer)
mysql -utom -pgeheim -Dtest < test-dump.sql     # DB-Dump "test" einspielen
```

Alternativ (Dump-Datei editieren) folgende Anweisungen am Anfang der Dumpdatei "test-dump.sql" hinzufügen:

```
  mysqldump -utom -pgeheim test > test-dump.sql   # Dump von DB "test" erzeugen
  edit test-dump.sql                              # SQL-Dumpdatei editieren:
    DROP DATABASE test;                           # - DB "test" löschen (Inhalt!
    CREATE DATABASE IF NOT EXISTS test;           # - DB "test" anlegen (leer)
    USE test;                                     # - Auf DB "test" umschalten
    ...                                           # - Restliche SQL-Anweisungen
```

Und dann die Sicherung der Datenbank einspielen:

```
  mysql -utom -pgeheim < test-dump.sql
```

HINWEIS: Bei einem SQL-Syntaxfehler wird die Verarbeitung sofort abgebrochen
(außer Option "-f/--force" ist gesetzt).

1k) Client "mysql": Spezielle Optionen

Die Optionen des "mysql"-Clients sind in Kurzform "-X" sowie in Langform "--XXX" angebbar, bei der Kurzform ist die GROSS/kleinschreibung relevant, bei der Langform nicht. Folgende Optionen des "mysql"-Client wurden in den vorherigen Abschnitten schon beschrieben:

Opt	Langform	Bedeutung
-e	--execute="<SqlCmd>"	SQL-Anweisung ausführen
	--auto-rehash	Tab.+Spaltennamen vervollst. (nach USE)
-A	--no-auto-rehash	Tab.+Spaltennamen NICHT vervollst. (alt)
-A	--skip-auto-rehash	Tab.+Spaltennamen NICHT vervollst. (alt)
-A	--disable-auto-rehash	Tab.+Spaltennamen NICHT vervollst. (neu)
	--sigint-ignore	Signal INT (Strg-C) ignorieren
	--connect-timeout=<N>	Sek. bis autom. Verbindungstrenn. (STD: 0)
	--reconnect	Bei Verb.verlust diese autom. wieder aufb.
	--skip-reconnect	Bei Verb.verlust diese NICHT autom. aufb.
	--max-allowed-packet=<N>	Max. Länge transf. Bef./Ergeb. (STD: 16MB)
	--net-buffer-length=<N>	Puffergröße TCP/IP+Socket-Komm. (STD: 16KB)
-U	--safe-updates	UPDATE- und DELETE-Anweisungen gegen
	--i-am-a-dummy	Weglassen von WHERE und LIMIT schützen
	--select-limit=<N>	Abbruch Erg.ausgabe ab N Zl. (STD: 1000)
	--max-join-size=<N>	Abbruch Erg.ausg. ab N Zl.komb. (STD: 1Mio)

Folgende Optionen legen das Verhalten bei fehlerhaften SQL-Anweisungen fest (STD: Bei SQL-Syntaxfehler Verarbeitung der restlichen Anw. abbrechen):

Kurzform	Langform	Bedeutung
-f	--force	Bei einem SQL-Fehler nicht abbrechen
	--show-warnings	Warnungen nach jeder Anweisung anzeigen

1l) Client "mysql": Ausgabeformat-Optionen

Folgende Optionen legen das AUSGABEFORMAT fest:

```
  +----------+------------+-----------------------------------------+
  | Kurzform | Langform   | Bedeutung                               |
  +----------+------------+-----------------------------------------+
  | -t       | --table    | ASCII-Rahmen (STD im Interaktiv-Modus)  |
  | -B       | --batch    | Spalten TAB-getrennt (STD im Batch-Modus) |
  | -E       | --vertical | vErtikale-Ausgabe (nützlich!)           |
  | -H       | --html     | HTML-Format (1 Zeile ohne NL!)          |
  | -X       | --xml      | XML-Format (mehrzeilig + eingerückt)    |
  +----------+------------+-----------------------------------------+
  | -r       | --raw      | Ohne Escape-Umwandlung ausgeben (*)     |
  +----------+------------+-----------------------------------------+
```

(*) HINWEIS: Der Client "mysql" wandelt beim Einlesen Escape-Sequenzen wie \n \r \t \b \a \0 \\ \" \' in echte (Steuer-)Zeichen um. Bei der Ausgabe erfolgt die umgekehrte Umwandlung Steuerzeichen nach Escape-Sequenzen nur dann, wenn KEINE der Optionen -t, -X, -E, --raw angegeben wird.

Standard-Ausgabeformate:

* Interaktiver Modus (Eingabe-Gerät ist EIN Terminal) --> Option -t/--table
  Werte jeder Ausgabespalte werden auf grösste notwendige Breite formatiert
  und Ausgabe wird mit einem "ASCII-Rahmen" versehen.

* Batch-Modus (Eingabe-Gerät ist KEIN Terminal) --> Option -B/--batch
  Werte jeder Ausgabespalte werden nicht auf einheitliche Breite
  formatiert, sondern durch je einen TABULATOR getrennt.

TIP: Für das menschliche Auge ist das 1. Format angenehmer zu lesen,
     für den Computer ist das 2. Format besser zu verarbeiten.

Bei breiten Tabellen ist das per Option "-E" aktivierte "vErtikale Format" sinnvoll. Die Spaltenwerte eines Datensatzes werden dann zeilenweise mit den Spaltennamen als Präfix ausgegeben und jeder Datensatz eingeleitet durch eine Zeile "*** N. row ***".

```
  *************************** 1. row ***************************
       nr: 77
  vorname: heinz
     name: bayer
  *************************** 2. row ***************************
       nr: 88
  vorname: Andrea
     name: Bayer
  *************************** 3. row ***************************
       nr: 99
  vorname: Richard
     name: Seiler
  3 rows in set (0.00 sec)
```

TIP: Im "mysql"-Client durch Stmt-Abschluss "\G" (go) statt ";" auslösen.

1m) Client "mysql": Sonstige Optionen

```
  +----+---------------------------+-----------------------------------------+
  | Opt| Langform                  |  Bedeutung                              |
  +----+---------------------------+-----------------------------------------+
  |    | --column-names            | Spaltennamen-Überschrift anzeig. (STD)  |
  | -N | --skip-column-names       | Spaltennamen-Überschrift weglassen      |
  | -m | --column-type-info        | Spaltentyp anzeigen (Metadaten)         |
  +----+---------------------------+-----------------------------------------+
  | -C | --compress                | Datenübertr. zw. Cl. + Server komprim.  |
  +----+---------------------------+-----------------------------------------+
  | -# | --debug=<Opts>            | Debuglogfile gemäß <Opts> schreiben     | !!!
  | -T | --debug-info              | Einige Debuginfo am Programmende ausg.  |
```

	--debug-check	Einige Debuginfo am Programmende ausg.	
	--default-character-set=<N>	Standard-Zeichensatz <Name>	
	--character-sets-dir=<Dir>	Verz. <DirPath> der Zeichensätze	
-L	--line-numbers	Zeilennummer bei Fehler ausgeben (STD)	
	--skip-line-numbers	Zeilennummer bei Fehler NICHT ausgeb.	
	--tee=<OutFile>	Ausgabe auf Datei <OutFile> zusätzlich	
	--no-tee	Ausgabe auf Datei wieder abschalten	
-n	--unbuffered	Ausgabe nach jeder Abfrage leeren	
-w	--wait	Warten+neu versuch. Verb. aufzubauen	
	--reconnect	Bei Verb.verlust diese wieder aufbauen	
	--skip-reconnect	Bei Verb.verlust Client verlassen	
-c	--comments	Kommentare zum Server senden (erhalten)	
	--skip-comments	Kommentare NICHT zum Server send (STD)	
-G	--named-commands	Interne Client-Kmdos überall erlaubt	
-g	--disable-named-commands	Interne Client-Kmdos nur in 1.Zl erl.	-g
	--no-named-commands	Analog (veraltet)	
	--delimiter=<Text>	SQL-Kmdo-Begrenzer def. (STD: ";")	
-i	--ignore-spaces	Leerzeichen nach Fkt.name ignorieren	
	--pager=<CmdPath>	Seitenweises Blätterkmdo. <CmdPath>	
	--disable-pager	Kein seitenweises Blättern	
	--no-pager	Kein seitenweises Blättern (veraltet)	
	--prompt=<Text>	Prompt-Definition (STD: "mysql> ")	
-o	--one-database	Nur -D<Db> zählt, USE <Db> ignoriert	
-q	--quick	Abfrageergebnisse nicht cachen	
	--secure-auth	Altes Protokoll (vor 4.1.1) verhindern	

1n) Client "mysql": Umgebungsvariablen

Folgende Umgebungsvariablen werden vom Client "mysql" ausgewertet:

Umgebungsvariable	Bedeutung
MYSQL_DEBUG	Debug-Trace-Optionen
MYSQL_HISTFILE	MySQL-History-Datei (STD: "$HOME/.mysql_history")
MYSQL_HISTIGNORE	MySQL-History-Datei ignorieren
MYSQL_HOME	Server-spezifische "my.cnf"-Datei
MYSQL_HOST	Hostname
MYSQL_PS1	Client-Prompt-String (STD: "mysql> ")
MYSQL_PWD	Passwort (unsicher!)
MYSQL_TCP_PORT	TCP/IP-Port (STD: 3306)
MYSQL_UNIX_PORT	Socket (bei "localhost" benutzt)
MYSQL_GROUP_SUFFIX	
DATA_VAR_MYSQL	
PAGER	Programm zum seitenweisen Blättern (--pager)
EDITOR	Editor für interaktives Bearbeiten (1. Wahl)
VISUAL	Editor für interaktives Bearbeiten (2. Wahl)
USER	Benutzer
HOME	Heimatverzeichnis des Benutzers
PATH	Shell-Suchpfad für Programme

LD_RUN_PATH	Suchpfad für Bibliothek "libmysqlclient.so"
UMASK	Maske für neu angelegte Dateien
UMASK_DIR	Maske für neu angelegte Verz.
TZ	Lokale Zeitzone

1o) Client "mysql": Interne Befehle (Escape-Sequenzen)

Neben SQL-Anweisungen versteht der "mysql"-Client folgende INTERNEN BEFEHLE bzw. ihre Abkürzung in Form einer zweibuchstabigen Escape-Sequenz (ein ";" als Abschluss ist hier nicht notwendig, schadet aber auch nicht):

Befehl	ESC	Bedeutung
? [<Arg>]	\?	Synonym für "help" (Argument <Arg>)
CHARSET <C>	\C	Zeichensatz <C> einschalten (für binlog-Verarbeitung)
CLEAR	\c	SQL-Anweisung abbrechen
CONNECT [D H]	\r	Serververbindung neu aufbauen (Opt: <Db> und <Host>)
DELIMITER <S>	\d	SQL-Kmdo-Begrenzer <S> def. (Zeilenrest, STD: ";")
EDIT	\e	Letzte SQL-Anweisung per $EDITOR/$VISUAL bearbeiten
EGO	\G	SQL-Anweisung ausführen (vErtikale Anzeige)
EXIT	\q	"mysql"-Client verlassen (auch Strg-C, Strg-D)
GO	\g	SQL-Anweisung ausführen (STD: ASCII-Tabellen Anz.)
HELP	\h	Diese Hilfe anzeigen (auch \?)
NOPAGER	\n	Seitenweise Anzeige ausschalten (stdout)
NOTEE	\t	Nicht mehr in Ausgabedatei schreiben
NOWARNING	\w	Keine Warnungen nach jeder Anweisung anzeigen (STD)
PAGER [<Cmd>]	\P	Seitenweise Anzeige ein (STD: $PAGER, z.B. "less")
PRINT	\p	Aktuelle SQL-Anweisung ausgeben
PROMPT [<Str>]	\R	"mysql"-Client-Prompt <Str> festleg. (STD: "mysql> ")
QUIT	\q	"mysql"-Client verlassen (auch Strg-C, Strg-D)
REHASH	\#	TAB-Completion-Hash aufbauen
RESETCONNECTION	\x	Verb. neu aufbauen (ab MY!5.7.3)
SOURCE <File>	\.	SQL-Datei <File> einlesen und ausführen (Include)
STATUS	\s	Server-Status anzeigen
SYSTEM <Cmd>	\!	System-Kommando <Cmd> ausführen (Shell-Escape)
TEE <File>	\T	Ausgabedatei <File> setzen (alles zusätzlich dorthin)
USE <Db>	\u	Auf Default/Standard-Datenbank <Db> umschalten
WARNING	\W	Warnungen nach jeder Anweisung anzeigen

1p) Client "mysql": Hilfe anzeigen

Im Client "mysql" ist zu den Client-internen Befehlen und den SQL-Anweisungen jederzeit Hilfe anzeigbar (die Hilfetexte stehen in der internen Verwaltungs-Datenbank "mysql" in den Tabellen "mysql.help_...").

Hilfe-Kommando	Bedeutung
?	Liste der Client-internen Befehle
\?	Liste der Client-internen Befehle
\h	Liste der Client-internen Befehle
HELP	Liste der Client-internen Befehle
HELP HELP	Hilfe zur Hilfe
HELP <SqlCmd>	Hilfe zu SQL-Anweisung (z.B. JOIN)
HELP CONTENTS	Hilfethemenliste (Topics)
HELP <Topic>	Hilfe zu Thema aus Themenliste

```
  | ... Account Management     | Hilfethema                          |
  | ... Administration         |                              "      |
  | ... Compound Statements    |                              "      |
  | ... Data Definition        |                              "      |
  | ... Data Manipulation      |                              "      |
  | ... Data Types             |                              "      |
  | ... Functions              |                              "      |
  | ... Functions and Modifiers for Use with GROUP BY       "      |
  | ... Geographic Features    |                              "      |
  | ... Help Metadata          |                              "      |
  | ... Language Structure     |                              "      |
  | ... Plugins                |                              "      |
  | ... Procedures             |                              "      |
  | ... Storage Engines        |                              "      |
  | ... Table Maintenance      |                              "      |
  | ... Transactions           |                              "      |
  | ... Triggers               |                              "      |
  | ... User-Defined Functions |                              "      |
  | ... Utility                |                              "      |
  +----------------------------+-------------------------------------+


1q) Client "mysql": Prompt-Definition
-------------------------------------

Das Aussehen des Prompts kann definiert werden per (STD: "mysql> "):

  PROMPT <Text>   # Escape-Sequenzen im <Text> erlaubt (ohne Quotierung "...")
  PROMPT          # Rückkehr zum STD.-Prompt "mysql> " (Leerz. am Ende!)

Beispiel (mit Escape-Sequenzen \h=host, \u=user, \d=database):

  PROMPT \u@\h:\d>\          # Leerzeichen am Ende!

"Escape-Sequenzen" um spezielle Zeichen oder variable Werte anzuzeigen:

  +---+----------+-------------------------------------------------+
  | Esc| Name     | Beschreibung                                    |
  +---+----------+-------------------------------------------------+
  | \  |          | Leerzeichen (dem Backslash folgt ein Leerzeichen) |
  | \_ |          | Leerzeichen (Unterstrich!)                      |
  | \" |          | Gänsefüßchen                                    |
  | \' |          | Hochkomma                                       |
  | \\ |          | Backslash-Zeichen "\"                           |
  +---+----------+-------------------------------------------------+
  | \S | Semicolon | Semikolon ";"                                  |
  | \t | tabulator | Tabulator-Zeichen                              |
  | \n | newline   | Zeilenvorschub                                 |
  +---+----------+-------------------------------------------------+
  | \c | count     | Anweisungszähler (pro Anweisung um 1 erhöht)   |
  | \l | delimiter | Aktueller Kmdo-Begrenzer (STD: ";", MY!5.0.25) |
  +---+----------+-------------------------------------------------+
  | \C | connection| Aktuelle Verbindungs-ID (ab MY!5.7.6)          |
  | \d | database  | Default-Datenbank                              |
  | \h | host      | MySQL-Server                                   |
  | \p | port      | Aktueller TCP/IP-Port oder Socket-Datei        |
  | \u | user      | Benutzername                                   |
  | \U | User      | Vollständiger Benutzername "user_name@host_name" |
  | \v | version   | MySQL-Server Version                           |
  +---+----------+-------------------------------------------------+
  | \D | date      | Aktuelles Datum "DD.MM.YYYY"                   |
  | \O | mOnth     | Aktueller Monat 3 Zeichen (Jan, Feb, ...)      |
  | \o | month     | Aktueller Monat in numerischer Form (1 ... 12) |
  | \w | weekday   | Aktueller Wochentag 3 Zeichen (Mon, Tue, ...)  |
  | \Y | Year      | Aktuelles Jahr (4 Ziffern)                     |
  | \y | year      | Aktuelles Jahr (2 Ziffern)                     |
```

\m	minutes	Minuten der aktuellen Zeit (00-59)
\P	PM	Vormittag/Nachmittag der aktuellen Zeit (AM/PM)
\R	Realtime	Stunde der aktuellen Zeit (00-23)
\r	realtime	Stunde der aktuellen Zeit (00-12)
\s	seconds	Sekunden der aktuellen Zeit (00-59)
\Z		Zeichen "Z" für jedes hier nicht aufgeführte Z.

TIP: Prompt in benutzerspez. Konfig-Datei definieren (Leerzeichen am Ende!):

```
[mysql]
prompt = "\u@\h:\d>\_"
```

TIP: Prompt auf der Kmdo-Zeile definieren:

```
mysql --prompt="\u@\h:\d>\_" ...
```

1r) Grafische MySQL-Programme (GUI)

Neben den bedienungstechnisch relativ einfachen aber trotzdem leistungsfähigen Kommandozeilenprogrammen sind noch eine Reihe GUI-basierter MySQL-Clients verfügbar, die per Maus bedienbar sind. Diese Programme müssen getrennt installiert werden, sie sind nicht Bestandteil des MySQL-Servers oder -Clients:

GUI-Programm	Beschreibung
MySQL Workbench	GUI-Client (frei, inkl. ER-Designer)
MySQL GUI Tools	Zusammenfassung von:
MySQL Administrator	Server-Konfiguration, -Verwaltung, -Wartung
MySQL Query Browser	Grafischer SQL-Client
MySQL Migration Toolkit	Schemata + Daten aus anderen DB importieren
phpMyAdmin	Webbasiertes DB-Management (Webserver nötig)
mysql-admin	Webbasiertes DB-Management (Webserver nötig)
Adminer	Webbasiertes DB-Management (Webserver nötig)
winMySQLadmin	GUI-Client (Windows, STD., MUSS lokal laufen!)
SQL-Front	GUI-Client (Windows, kommerziell)
SQLyog	GUI-Client (Windows, kommerziell)
MySQLManager	GUI-Client (Windows, kommerziell)
Navicat for MySQL	GUI-Client (Windows, kommerziell)
EMS MySQL Manager	GUI-Client (Windows, kommerziell)
MyAdmin	GUI-Client (Windows, kommerziell)
HeidiSQL (MySQL-Front)	GUI-Client (Windows, Linux/MacOS per Wine)
SQuirrel	GUI-Client (Java: Windows, MacOS, Linux)
TOAD	GUI-Client (Windows, kommerziell, viele DB)
Knoda	GUI-Client analog Access (Linux)
Rekall	GUI-Client (Linux)
Ksql	GUI-Client (Linux)
KMySQLadmin	GUI-Client (Linux)
Emma	GUI-Client (Linux)
MySQL Control Center	Weiterentw. von MySQLGUI (mysqlcc, veraltet)
MySQLGUI	Grafischer SQL-Client (veraltet)
mysql-navigator	Grafischer SQL-Client (veraltet)

HINWEIS: Diese Liste erhebt keinen Anspruch auf Aktualität und Vollständigkeit.

2) Syntax

2a) Leerraum und Formatierung

Leerraum + Zeilenvorschübe + GROSS/kleinschreibung der SQL-Schlüsselworte ist in SQL-Anweisungen (SQL-Statements) irrelevant. Allerdings empfiehlt sich die übliche Konvention:

  SQL-Schlüsselworte GROSS schreiben!

Erst ";" oder "\g" (go) bzw. "\G" (ego = Option -E = --vertical = vErtikale Ausgabe) schließen eine SQL-Anweisung ab. Ausnahme: Interne "mysql"-Client Befehle wie "USE <Db>" (besser nicht daran gewöhnen ;-).

Es ist sinnvoll, SQL-Anweisungen per Einrückung und Zeilenvorschübe übersichtlich zu formatieren, um eine optimale Lesbarkeit zu erreichen. Die Übersetzungs- und Ausführungsgeschwindigkeit der Anweisung verringert sich dadurch nicht. Insbesondere lange SQL-Anweisungen der Übersicht halber sinnvoll auf mehrere Zeilen umbrechen und einrücken, z.B.:

```
  UPDATE TABLE pers SET nr = 7, vorname = "Heinz", name = "Bayer";
```

besser so formatieren:

```
  UPDATE TABLE pers
    SET nr      = 7,
        vorname = "Heinz",
        name    = "Bayer";
```

2b) Zeichenketten (Strings) und Quotierung

Alle konstanten Werte außer Zahlen (d.h. Text-, Blob-, Bit-, Set-, Enum-, Datum- und Zeitwerte) sind in "..." oder '...' einzuschließen (zu QUOTIEREN). Prinzipiell sind aber auch Zahlen so darstellbar, d.h. ALLE Werte dürfen quotiert werden (einheitliches Prinzip). Um in "..." das Zeichen " und in '...' das Zeichen ' einzutragen, dieses einfach verdoppeln oder mit "\" quotieren:

```
  SELECT "Hallo", 'Welt';       # --> Hallo Welt
  SELECT Hallo, Welt;           # --> FEHLER!
  SELECT 1, 23, 456;            # --> 1 23 456
  SELECT '1', "23", '456';      # --> 1 23 456
  SELECT 'Don''t';              # --> Don't
  SELECT 'Don\'t';              # --> Don't
  SELECT "Don't";               # --> Don't
  SELECT "Er sagte: ""..."""";  # --> Er sagte: "..."
  SELECT "Er sagte: \"...\"";   # --> Er sagte: "..."
  SELECT 'Er sagte: "..."';     # --> Er sagte: "..."
```

HINWEIS: Zwischen Zeichenketten in "..." und '...' gibt es KEINEN Unterschied (wie in anderen Programmiersprachen z.B. bei Escape-Sequenzen \C), allerdings ist nur '...' ANSI-SQL-kompatibel ("..." wird dort verwendet, um Bezeichner mit Sonderzeichen zu quotieren, in MySQL erfolgt dies durch Backquotes `...`).

2c) Escape-Sequenzen

Escape-Sequenzen (Maskierungszeichen) \C zur Darstellung von Sonderzeichen in Zeichenketten der Form "..." und '...':

ESC	Bedeutung

\0	ASCII 0-Zeichen (NUL)
\'	Einfaches Hochkomma in '...' (auch '' in '...')
\"	Doppeltes Hochkomma in "..." (auch "" in "...")
\\	Backslash (\)
\a	Alert (Ton)
\b	Backspace
\f	Formfeed (Seitenvorschub beim Drucker)
\n	Newline (Linefeed)
\r	Carriage Return
\t	Tabulator
\v	Vertikaler Tabulator
\Z	ASCII 26 (Control-Z, unter Windows evtl. notwendig)
\%	%-Zeichen (Zeichen "%" selbst in LIKE)
_	_-Zeichen (Zeichen "_" selbst in LIKE)

2d) Zahlen

Wird eine Zeichenkette in einem Zahlenzusammenhang verwendet, dann wird das max. Anfangsstück (PRÄFIX), das noch wie eine Zahl aussieht, AUTOMATISCH in diese Zahl KONVERTIERT (und eine Warnung ausgegeben).

ACHTUNG: Sieht eine Zeichenkette überhaupt nicht wie eine Zahl aus, dann entspricht sie der Zahl 0 (Null).

```
  SELECT 123 + 0, -5 + 0, 1e+5 + 0;                  # --> 123, -5, 100000
  SELECT "123" + 0, "-5" + 0, "1e+5" + 0;            # --> 123, -5, 100000
  SELECT "123def" + 0, "-5ghi" + 0, "1e+5kjl" + 0;   # --> 123, -5, 100000
  SELECT "def" + 0, 100 + "ghi";                     # --> 0, 100
```

Fließkommazahlen sind (unabhängig von Sprach- und Collation-Einstellungen) immer mit DEZIMALPUNKT zu schreiben (Dezimalkomma und Tausendertrennzeichen sind nicht erlaubt). Die Ausgabe von Dezimalkommas ist per FORMAT(<Zahl>, <Nkst>) erreichbar. Rundung auf feste Anzahl Nachkommastellen oder ganze Zahl ist mit ROUND(<Zahl>, <Nkst>) erreichbar.

```
  SELECT 123.456 + 0, 123,456 + 0;                 # --> 123.456, 123, 456
  SELECT FORMAT(123.456, 2), FORMAT(123.456, 0);   # --> 123.46, 123
  SELECT ROUND(123.456, 2), ROUND(123.456, 0);     # --> 123.46, 123
```

Hexadezimalzahlen bzw. -bytes folgendermaßen schreiben:

```
  0xaffe...    # Variante A (0X1010 nicht erlaubt!)
  X'affe...'   # Variante B (x"1010" nicht erlaubt!)
  x'affe...'   # Variante C (X"1010" nicht erlaubt!)
```

Binärzahlen und Bitfelder folgendermaßen schreiben:

```
  0b1010...    # Variante A (0B1010 nicht erlaubt!)
  b'1010...'   # Variante B (b"1010" nicht erlaubt!)
  B'1010...'   # Variante C (B"1010" nicht erlaubt!)
```

Zahlen mit führender 0 sind Dezimalzahlen (keine Oktaldarstellung):

```
  SELECT 0123 + 0, 0815 + 0;   # --> 123, 815
```

2e) Datenbank-Objekte

MySQL kennt folgende "Datenbank-Objekte" (oder kurz "Objekte") und

"Syntax-Elemente". Die Platzhalter für BEZEICHNER dieser Objekte stehen in diesem Skript in SQL-Anweisungen immer in spitzen Klammern <...> und müssen durch einen konkreten Bezeichner ersetzt werden:

Objekt	Platzhalter	Bedeutung
Alias	<Alias>	Weiterer Name für ein Datenbank-Objekt
Column	<Col>	Tabellenspalte
Database	<Db>	Datenbank (Schema)
Event	<Event>	Ereignis (einmalig oder periodisch)
Index	<Idx>	Index einer Tabelle
Log File	<LogFile>	Logdatei
Partition	<Part>	Horizontale Zerlegung einer Tabelle
Prepared Statement	<PrepStmt>	Vorkompilierte Anweisung
Privilege	<Priv>	Benutzerrecht
Statement	<Stmt>	SQL-Anweisung
Stored Function	<Func>	Funktion (Rückgabewert)
Stored Procedure	<Proc>	Prozedur (kein Rückgabewert)
Table	<Tbl>	Tabelle
Tablespace	<TblSpace>	Datei für Tab.daten (analog Partition)
Trigger	<Trig>	Trigger einer Tabelle
User	<User>	Benutzeraccount
View	<View>	Sicht (vordef. gespeicherte Abfrage)
Datatype	<Typ>	Datentyp (z.B. INT, CHAR, VARCHAR, ...)
Character Set	<CharSet>	Zeichensatz (z.B. latin1, utf8, ...)
Collation	<Coll>	Sortierregel (z.B. latin1_german_ic)
Directory Path	<DirPath>	Verz.pfad (absolut)
Transcodierung	<Transcode>	Zeichensatzcodierung

2f) Identifier (Bezeichner)

Bezeichner dürfen standardmäßig aus den Zeichen A-Z, a-z, 0-9, _ und $ bestehen (d.h. insbesondere KEINE Leerzeichen!), führende Ziffern sind nicht erlaubt. Mit der Schreibweise `...` (QUOTIERUNG mit Backquotes, NICHT '...' oder "...") sind auch beliebige andere Zeichen verwendbar (NICHT empfohlen!).

```
SELECT Name                          FROM test;   # OK (STD-Form)
SELECT `Name`                        FROM test;   # OK (STD-Form)
SELECT _Name_                        FROM test;   # OK (STD-Form)
SELECT $Name                         FROM test;   # OK (STD-Form)
SELECT Und_Ein_Name$                 FROM test;   # OK (STD-Form)
SELECT `Name mit Leer zeichen`       FROM test;   # OK (Sonderzeichen)
SELECT `Name mit Sonderzeichen äöüß` FROM test;   # OK (Sonderzeichen)
```

Kollidiert ein BEZEICHNER (Objektname) mit einem SQL-Schlüsselwort, so kann er in `...` (Backquotes!) gesetzt dennoch verwendet werden (QUOTIERUNG). Die normale String-Quotierung mit "..." oder '...' funktioniert hier nicht!

```
CREATE TABLE alter (...);                  # FALSCH (Kollision)!
CREATE TABLE `alter` (...);                # OK (Quotes notwendig)
SELECT user, host FROM mysql.user;         # OK (STD-Form ohne Quotes)
SELECT `user`, `host` FROM `mysql`.`user`; # OK (Quotes überflüssig)
SELECT `user`, `host` FROM `mysql.user`;   # FALSCH (Namen einzeln quot.)!
SELECT "user", "host" FROM "mysql"."user"; # FALSCH (String)!
SELECT 'user', 'host' FROM 'mysql'.'user'; # FALSCH (String)!
```

HINWEIS: In generierten SQL-Anweisungen (z.B. per "mysqldump") werden ALLE BEZEICHNER grundsätzlich prophylaktisch in `...` gesetzt (auch wenn das gar nicht notwendig wäre).

Maximal erlaubte Bezeichner-Länge von Datenbank-Objekten und Namen:

```
+-----+------------+
| Max | Bezeichner |
+-----+------------+
|  64 | Datenbank  |
|  64 | Tabelle    |
|  64 | Spalte     |
|  64 | Routine    |
| 255 | Alias      |
+-----+------------+
|  60 | Host       |
|  16 | Benutzer   |
|  41 | Passwort   |
+-----+------------+
```

Zugriff auf die Objekte Datenbank, Tabelle, Spalte und Stored Procedure erfolgt durch "relative" oder "vollqualifizierte" (full qualified) Bezeichner (im "mysql"-Client per <TAB>-Taste = TAB-Completion automatisch vervollständigbar):

```
+--------------------+-------------------------------------------------------+
| Bezeichner         | Bedeutung                                             |
+--------------------+-------------------------------------------------------+
| <Db>.<Tbl>         | A) Vollständiger Pfad (immer OK!)                     |
| <Db>.<Tbl>.<Col>   |    Analog                                             |
| <Db>.<Tbl>.*       |    Analog (ALLE Spalten der Tabelle)                  |
+--------------------+-------------------------------------------------------+
| <Tbl>              | B) Relativ zu Default/Standard-Datenbank (USE <Db>)   |
| <Tbl>.<Col>        |    Analog                                             |
| <Tbl>.*            |    Analog (ALLE Spalten der Tabelle)                  |
+--------------------+-------------------------------------------------------+
| <Col>              | C) Analog B) in INSERT/SELECT/UPDATE/DELETE           |
|                    |    Relativ zu EINER Tabelle immer OK!                 |
|                    |    Bei MEHREREN verknüpften Tabelle nur bei           |
|                    |    pro Tabelle eindeutigen Spaltennamen OK.           |
|                    |    (mit eindeutigen Spalten-Aliasen auch OK)          |
| *                  |    Analog (ALLE Spalten einer Tabelle)                |
+--------------------+-------------------------------------------------------+
| <Db>.<Proc>        | D) Vollständiger Pfad (immer OK!)                     |
| <Proc>             | E) Relativ zu Default/Standard-Datenbank (USE <Db>)   |
+--------------------+-------------------------------------------------------+
```

```
SELECT test.pers.name FROM test.pers;      # A) OK (Datenbank "test")
USE test;                                  #    Default-DB "test" auswählen
SELECT pers.name FROM pers;                # B) OK (Datenbank "test")
SELECT name FROM pers;                     # C) OK bei EINER Tabelle (DB "test")
SELECT name FROM pers, age;                # C) Problem bei Sp.namen-Kollision
SELECT pers.name, age.name FROM pers, age; # C) OK (auch bei Sp.Kollision)
SELECT p.name, a.name FROM pers p, age a;  # C) OK (Aliase auch bei Sp.Koll.)
```

ACHTUNG: Wird im Fall C) zu einer beteiligten Tabelle nachträglich eine Spalte hinzugefügt, die GLEICHNAMIG zu einer Spalte einer anderen beteiligten Tabelle ist, führt das in bereits vorhandenen SQL-Anweisungen zu Syntaxfehlern, wenn darin auf diese Spalte zugegriffen wird. Daher am besten mit Variante B) arbeiten und zur Abkürzung ALIASE einführen.

HINWEIS: Die Bestandteile eines relativen oder vollqualifizierten Bezeichners müssen GETRENNT mit `...` quotiert werden:

```
SELECT DISTINCT `mysql`.`user`.`host` FROM `mysql`.`user`;   # OK
SELECT DISTINCT `mysql.user.host`     FROM `mysql.user`;     # FALSCH!
SELECT DISTINCT mysql.user.host       FROM `mysql`.`user`;   # OK
```

Analog MÜSSEN die beiden Bestandteile "User" und "Host" eines Benutzernamens "User@Host" GETRENNT quotiert werden (allerdings mit "..." oder '...', da es sich nicht um Bezeichner, sondern um externe Namen handelt):

```
  CREATE USER "hans"@"localhost";   # OK
  CREATE USER 'hans'@'localhost';   # OK
  CREATE USER `hans`@`localhost`;   # FALSCH
  CREATE USER "hans@localhost";     # FALSCH!
  CREATE USER 'hans@localhost';     # FALSCH!
  CREATE USER `hans@localhost`;     # FALSCH!
  CREATE USER hans@localhost;       # FALSCH!
```

2g) GROSS/kleinschreibung

Beim Vergleichen und beim Sortieren von Daten in Tabellen ignoriert MySQL normalerweise die GROSS/kleinschreibung außer in folgenden Fällen:

* BINARY vor einem String in den Vergleichen = != <> <=> < <= > >=, BETWEEN, IN, LIKE, RLIKE/REGEX und in ORDER BY erzwingt die Beachtung der GROSS/kleinschreibung bei Vergleichen / Sortieren.

* Die Datentypen BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB und LONGBLOB erzwingen Beachtung der GROSS/kleinschreibung beim Vergleichen / Sortieren.

Die GROSS/kleinschreibung in SQL-Anweisungen ist meist egal AUSSER bei den Bezeichnern von Datenbanken, Tabellen, Views und Logfilegroups unter einem Betriebssystem mit case-sensitivem Dateisystem (da als Verz./Datei abgelegt). Die Namen von Triggern, Labeln (Marken) und Benutzern sowie Passwörter sind immer "case-sensitive". Es gibt daher gewisse KONVENTIONEN bgzl. GROSS/kleinschreibung, an die man sich halten sollte.

Sprachelement	GROSS/kleinschreibung relevant	Konvention
Datenbank-Bezeichner	JA (Linux), NEIN (Windows)	klein
Tabellen-Bezeichner	JA (Linux), NEIN (Windows)	klein
View-Bezeichner	JA (Linux), NEIN (Windows)	klein
Dateiname/pfad	JA (Linux), NEIN (Windows)	klein
Logfilegroup-Name	JA (Linux), NEIN (Windows)	klein
Tablespace-Name	JA (Linux), NEIN (Windows)	klein
Host-Name	NEIN	klein
Benutzer-Name	JA	klein
Passwort	JA	---
Trigger-Bezeichner	JA	klein
Label (Marke)	JA	GROSS
SQL-Schlüsselwort	NEIN	GROSS
SQL-Funktions-Bezeichner	NEIN	GROSS
Spalten-Bezeichner	NEIN	klein
Index-Bezeichner	NEIN	klein
Variablen-Bezeichner	NEIN (ab MY!5.0), JA (bis MY!4.1)	klein
Prepared-Stmt-Bezeichner	NEIN	klein
Partitions-Bezeichner	NEIN	klein
Prozedur-Bezeichner	NEIN	klein
Funktions-Bezeichner	NEIN	klein
Event-Bezeichner	NEIN	klein
String-Vergleich	JA/NEIN (abhängig von Collation)	---
RegEx-Vergleich	JA/NEIN (abhängig von Collation)	---
Tabellen-Alias	NEIN	klein
Spalten-Alias	NEIN	klein

TIP: SQL-Schlüsselworte aus Gründen der besseren Lesbarkeit und der einfacheren Suchmöglichkeit IMMER GROSS schreiben (Konvention):

```
  select * from pers order by nr asc;     # Schlecht lesbar
  SELECT * FROM pers ORDER BY nr ASC;     # Gut lesbar
```

In diesem Skript werden daher alle SQL-Schlüsselworte GROSS, hingegen alle Datenbank-, Tabellen-, Spalten- und sonstige Bezeichner klein geschrieben.

2h) Kommentare

Es gibt folgende Möglichkeiten, Kommentare in den SQL-Quelltext einzubauen:

Syntax	Typ	Beschreibung
-- ...	ANSI-SQL	Bis Zeilenende (Leerz. nach "--" nötig!, MY!)
#...	Shell	Bis Zeilenende (MY!)
/*...*/	C	Beliebig viele Zeilen
/*! ...*/	C	Inhalt NUR von MySQL ausgeführt (MY!)
/*!NXXYY ...*/	C	Inhalt ab MySQL-Version N.XX.YY ausgeführt (MY!)
--...	ANSI-SQL	NICHT möglich (Leerz. nach "--" nötig!, MY!)
//...	C++	NICHT möglich!

Die ersten 3 Kommentartypen werden vom "mysql"-Client VOR dem Transfer einer SQL-Anweisung zum MySQL-Server ENTFERNT. Sollen sie doch mit übertragen und somit in den Logdateien abgelegt werden, dann den Option "--comments" beim Aufruf des "mysql"-Client angeben (STD: --skip-comments). Kommentare der Form /*!...*/ werden grundsätzlich an den MySQL-Server übertragen und stehen somit immer in den Logdateien.

Beispiele:

```
  SHOW TABLES; # Kommentar bis zum Zeilenende      # OK
  SHOW TABLES; #Kommentar bis zum Zeilenende       # OK
  SHOW TABLES; -- Kommentar bis zum Zeilenende     # OK
  SHOW TABLES; --Kommentar bis zum Zeilenende      # FEHLER (Leerz. fehlt!)
  SHOW TABLES; // Dies ist KEIN Kommentar!         # FEHLER (nicht erlaubt)
  SHOW /* mehrzeiliger                             # OK (mehrzeiliger Kommentar)
          Kommentar                                # OK (mehrzeiliger Kommentar)
       */ TABLES;                                  # OK (mehrzeiliger Kommentar)
  SELECT * FROM user /*! STRAIGHT_JOIN */ host     # Von MySQL ausgeführt,
    ON user.Host = host.Host;                      # von anderen DB nicht
  CREATE /*!32302 TEMPORARY */ TABLE t1 (a INT);   # Ab MY!3.23.02 ausgeführt
  CREATE /*!99999 Kommentar */ TABLE t2 (a INT);   # NIE ausgef. aber erhalten
```

3) Typische MySQL-Datenbankbefehle (Tutorial)

Dieser (lange) Abschnitt bietet eine Einführung in die wichtigsten SQL-Kommandos von MySQL anhand von Beispielen. Verwendet werden darin die beiden Datenbanken "first" und "test" sowie mehrere Tabellen (z.B. "pers" und "age"). Eine ausführliche Beschreibung dieser und anderer SQL-Anweisungen ist in den folgenden Kapiteln und in --> mysql-admin-HOWTO.txt zu finden.

HINWEIS: SQL-Anweisungen dürfen auf mehrere Zeilen umbrochen werden und sind immer mit ";" (oder "\g" bzw. "\G") abzuschließen.

TIP: Zu jeder neuen Datenbank einen EIGENEN MySQL-Benutzer (hier: "tom") anlegen, ihm ein Passwort (hier: "geheim") sowie geeignete Zugriffsrechte (hier: GRANT ALL PRIVILEGES = sämtliche Rechte) geben. Er darf dann nur mit dieser Datenbank arbeiten (sie muss dazu noch gar nicht existieren!):

```
  CREATE USER "tom"@"localhost";   # --> Leeres Passwort, keine Zugriffsrechte
                                   #
  GRANT ALL PRIVILEGES             # Alle Zugriffsrechte...
    ON first.*                     # ...auf alle Objekte der Datenbank "first"
      TO "tom"@"localhost"         # ...für Benutzer "tom@localhost"
        IDENTIFIED BY "geheim"     # ...mit Passwort "geheim"
          WITH GRANT OPTION;       # ...mit Rechteweitergabe (optional)

Passwort nachträglich ändern:

  SET PASSWORD FOR "tom"@"localhost" = PASSWORD("geheim");

HINWEIS: Seit MY!5.6 ist wg. dem Plugin "validate_password" standardmäßig ein
Passwort mit folgenden Eigenschaften nötig:

  * Mindestlänge 8 Zeichen
  * Mindestens ein Grossbuchstabe
  * Mindestens ein Kleinbuchstabe
  * Mindestens eine Ziffer
  * Mindestens ein Sonderzeichen (außer Buchstabe und Ziffer)

TIP: Vorher "CREATE USER" ist nicht unbedingt notwendig, schadet aber auch
nicht, da das GRANT-Statement den Benutzer automatisch anlegt, sofern er noch
nicht existiert (solange SQL-Modus "NO_AUTO_CREATE_USER" nicht gesetzt ist).

TIP: Username == Datenbankname != Tabellenname != Spaltenname wählen, sonst
besteht Verwechslungsgefahr (oder mit Präfix "u_" = User, "d_" = Database, "t_"
= Table, "c_" = Column arbeiten).

HINWEIS: Manche DB-Systeme legen zu jeder Datenbank automatisch einen Benutzer
gleichen Namens an, der alle Rechte bzgl. dieser Datenbank zugewiesen bekommt
(PostgreSQL, Oracle). MySQL kennt dieses Standard-Verhalten nicht.

Datenbank (Schema) "first" anlegen (meist als MySQL-Administrator "root"):

  CREATE DATABASE first;                # Fehler falls schon existent
  CREATE SCHEMA   first;                #   (analog)
  CREATE DATABASE IF NOT EXISTS first;  # Nur falls noch nicht existent (MY!)
  CREATE SCHEMA   IF NOT EXISTS first;  #   (analog MY!)

Datenbank (Schema) umbenennen ("root" oder Besitzer, nur von MY!5.1.7 bis
MY!5.1.23 verfügbar da zu gefährlich!):

  RENAME DATABASE first TO new;   # "first" --> "new" umbenennen
  RENAME SCHEMA   new TO first;   #   (analog umgekehrt)

Benutzer "tom" nimmt per "mysql"-Client Verbindung mit dem MySQL-Server (lokal)
auf und wählt beim Verbindungsaufbau "first" als Default/Standard-Datenbank aus:

  mysql -utom -p                  # Password interaktiv abgefragt (sicher!)
  mysql -utom -pgeheim            # Keine Default-Datenbank --> "USE first"
  mysql -utom -pgeheim -Dfirst    # Default-Datenbank --> "first"
  mysql -utom -pgeheim first      # Default-Datenbank --> "first"
  mysql -utom -pgeheim -hglasgow  # MySQL-Server auf Rechner "glasgow"

Abfrage der Verbindungsdaten (Klammern "()" notwendig, da Funktionsaufrufe):

  SELECT DATABASE();                    # Standard-Datenbank (oder "NONE")
  SELECT SCHEMA();                      #   (analog)
  SELECT USER();                        # Benutzer-Anmeldung <User>@<Host> !=
  SELECT CURRENT_USER();                #   Angemeld. Benutzer (evtl. anonym)
  SELECT SESSION_USER();                #   (analog)
  SELECT SYSTEM_USER();                 #   (analog)
  SELECT CONNECTION_ID();               # Sitzungs-ID (Thread, Prozess)
  SELECT VERSION();                     # MySQL-Server Version
```

Abfrage des Server-Status (Database, User, Server Version, Connection-ID,
Character set, Protocol Version, Connection type, UNIX Socket, Uptime, ...):

```
  STATUS   # Nur im "mysql"-Kommandozeilenclient vorhanden
  \s       # (analog)
```

Vorhandene Datenbanken auflisten:

```
  SHOW DATABASES;              # Alle Datenbanknamen auflisten
  SHOW SCHEMAS;                #   (analog)
  SHOW DATABASES LIKE "my%";   # Nur DB-Namen mit "my" am Anfang
  SHOW SCHEMAS   LIKE "my%";   #   (analog)
```

Als Default/Standard-Datenbank (Schema) "first" auswählen. Da es sich um einen
internen "mysql"-Clientbefehl handelt, darf der abschließende ";" weggelassen
werden (besser gar nicht erst daran gewöhnen!). Alle Bezeichner (Objektnamen)
ohne weitere Qualifizierung beziehen sich dann auf diese Datenbank (dieses
Schema) --> 2e) Datenbank-Objekte

```
  USE first    # Variante A (ohne ";")
  USE first;   # Variante B (mit ";")
```

Tabellen einer Datenbank auflisten:

```
  SHOW TABLES;                          # Alle aus Standard-DB "first"
  SHOW TABLES FROM mysql;               # Alle aus Verwaltungs-DB "mysql"
  SHOW TABLES LIKE "a%";                # Mit Name "a..." aus Standard-DB "first"
  SHOW TABLES FROM mysql LIKE "%a%";    # Mit Name "...a..." aus DB "mysql"
```

Tabelle "pers" (bedingt wenn sie noch nicht existiert) in Default-DB anlegen
(3 Spalten, im 2. Fall mit einer anderen Engine statt Standard-Engine "InnoDB",
Spalten ohne/mit Defaultwert und mit NULL erlaubt/nicht erlaubt):

```
  CREATE TABLE pers (                 # STD-Engine = InnoDB
    nr      INT          NOT NULL,    # Komma       (     NULL verboten)
    vorname VARCHAR(30),              # Komma       (STD: NULL erlaubt)
    name    VARCHAR(30)               # KEIN Komma! (STD: NULL erlaubt)
  ) COMMENT = "Personen";             # Kommentar zur Tabelle

  CREATE TABLE IF NOT EXISTS pers (   # Kein Fehler falls schon existent
    nr      INT         NOT NULL,     # NULL verboten
    vorname VARCHAR(30) DEFAULT "",   # Defaultwert leere Zeichenkette
    name    VARCHAR(30) NULL          # NULL explizit erlaubt (sowieso STD)
  ) ENGINE = "MyISAM";                # Engine MyISAM (alt: TYPE statt ENGINE)
```

HINWEIS: Jedes DB-Objekt hat automatisch einen "Besitzer", nämlich den User,
der es anlegt. Dieser User hat automatisch alle Rechte daran und kann es z.B.
umbenennen und auch wieder löschen.

HINWEIS: Die Spalten-REIHENFOLGE einer Tabelle ist nur relevant bei:
  * Abfragen mit "*"
  * Speichern mit INSERT INTO <Tbl> VALUES (...) ohne Liste von Spaltennamen
    (irrelevant bei INSERT INTO <Tbl> (<Col1>, ...) VALUES (...))
  * Mehr als einer TIMESTAMP-Spalten in einer Tabelle
    (ERSTE TIMESTAMP-Spalte wird bei jeder Datensatz-Änderung aktualisiert).

Temporäre Tabelle anlegen:

```
  CREATE TEMPORARY TABLE pers (      #
    nr      INT          DEFAULT 0   #
    vorname VARCHAR(30),             #
    name    VARCHAR(30)              #
  ) CHARACTER SET latin1;            # Standard-Zeichensatz für Textspalten
```

HINWEIS: Temporäre Tabellen

```
  * Dürfen genauso heißen wie echte Tabellen.
  * ÜBERDECKEN evtl. vorhandene gleichnamige echte Tabelle,
    bis sie wieder gelöscht werden.
  * Sind SITZUNGSBEZOGEN (d.h. pro Sitzung gleicher Name verwendbar,
    ohne dass es zu Kollisionen kommt).
  * Werden am Ende einer Sitzung automatisch entfernt,
    sofern nicht vorher manuell gelöscht (per DROP TEMPORARY <Tab>).
  --> mysql-admin-HOWTO.txt --> 8a) Temporäre Tabellen

Tabellenstruktur anzeigen (Spalten + Datentypen):

  DESCRIBE pers;             # Variante A
  DESCRIBE pers name;        # Variante B (nur Spalte "name")
  DESC pers;                 # Variante C
  DESC pers name;            # Variante D (nur Spalte "name")
  EXPLAIN pers;              # Variante E
  SHOW COLUMNS FROM pers;    # Variante F
  SHOW TABLE pers;           # FEHLER: Gibt es nicht!

Vollständige Tabellen-Definition anzeigen (mit Engine und allen Einstellungen):

  SHOW CREATE TABLE pers;    # Ausgabe horizontal
  SHOW CREATE TABLE pers\g   # Ausgabe horizontal (go)
  SHOW CREATE TABLE pers\G   # Ausgabe vertikal (ego, Option -E)
  SHOW TABLE pers;           # FEHLER: Gibt es nicht!

Alle Datensätze einer Tabelle anzeigen (aktuell noch leer):

  SELECT * FROM pers;                  # Alle Spalten in Def.reihenfolge
  SELECT nr, vorname, name FROM pers;  #   (analog)
  SELECT vorname, nr, name FROM pers;  # Alle Spalten in anderer Reihenfolge
  SELECT vorname, name FROM pers;      # Nur ausgewählte Spalten
  SELECT nr FROM pers;                 # Nur ausgewählte Spalte

Datensätze in Tabelle "pers" einfügen (nicht angegebene Spalten werden mit
ihrem Defaultwert oder NULL gemäß Angabe in Tabellen-Definition belegt):

  INSERT INTO pers (nr, vorname, name)
    VALUES (1, "Thomas", "Birnthaler");
  INSERT INTO pers (nr, vorname, name)
    VALUES (2, "Markus", "Mueller");
  INSERT INTO pers (nr, vorname, name)
    VALUES (8, "Andrea", "Bayer");

Bei vollständigen Datensätzen kann auf die Angabe der Spaltennamen verzichtet
werden. Dann müssen aber REIHENFOLGE und TYP der einzufügenden Daten exakt mit
der Spaltenreihenfolge in der Tabellen-Definition übereinstimmen (Vorsicht!):

  INSERT INTO pers VALUES (7, "Heinz", "Bayer");
  INSERT INTO pers VALUES (NULL, "Hans", "Dampf");
  INSERT INTO pers VALUES (NULL, NULL, "Unbekannt");

Andere Einfüge-Syntax:

  INSERT INTO pers
    SET nr      = 7,
        vorname = "Heinz",
        name    = "Bayer";

Einfügen von mehr als einem Datensatz gleichzeitig (sehr effizient, MY!):
ACHTUNG: Server-Variable "max_allowed_packet" muss gross genug sein, um
Statement vollständig zum Server übertragen zu können):

  INSERT INTO pers (nr, vorname, name)        # Nur 1x Spaltennamen!
    VALUES (1,    "Thomas",  "Birnthaler"),  # Nur 1x VALUES!
           (2,    "Markus",  "Mueller"),     # "," zw. Datensätzen!
```

```
          (8,    "Andrea",  "Bayer"),        #
          (9,    "Richard", "Seiler"),       #
          (7,    "Heinz",   "Bayer"),        #
          (5,    "Hans",    "Dampf"),        #
          (3,    NULL,      "Unbekannt");    # Anweisungsende
```

Laden der Daten aus einer externen CSV-Datei (Comma Separated Values, enthält
pro Datensatz eine Zeile abgeschlossen durch <NL> (Unix), <CR> <NL> (Windows)
oder <CR> (MacOS). Die Datensatz-Elemente müssen in SPALTENREIHENFOLGE
im Datensatz stehen und durch ein Tabulatorzeichen <TAB> getrennt sein.
Bei CSV-Dateien notwendige 1. Zeile mit Spaltennamen kann ignoriert werden.
Die Daten können auf Client ("LOCAL") oder Server (kein "LOCAL") liegen.

```
  LOAD DATA LOCAL INFILE "/path/to/pdata.txt" INTO TABLE pers;    # Linux A
  LOAD DATA LOCAL INFILE "/path/to/pdata.txt" INTO TABLE pers     # Linux B
    LINES TERMINATED BY "\n"                                      #
    IGNORE 1 LINES;                                               # 1.Zl ign.
  LOAD DATA LOCAL INFILE "D:/path/to/pdata.txt" INTO TABLE pers   # Windows
    LINES TERMINATED BY "\r\n"                                    #
    IGNORE 1 LINES;                                               # 1.Zl ign.
  LOAD DATA INFILE "D:\\path\\to\\pdata.txt" INTO TABLE pers      # Windows
    LINES TERMINATED BY "\r\n"                                    #
    IGNORE 1 LINES;                                               # 1.Zl ign.
  LOAD DATA INFILE "/path/to/pdata.txt" INTO TABLE pers           # MacOS
    FIELDS TERMINATED BY "\t"                                     # Tabs
    LINES TERMINATED BY "\r"                                      #
    IGNORE 1 LINES                                                # 1.Zl ign.
    (nr, vorname)        # Spalten in Reihenfolge laden (sonst alle)
    SET name = '';       # Spalten fix füllen
```

ACHTUNG: Voraussetzungen:

* Damit LOAD LOCAL INFILE funktioniert, muss die Server-Variable "local_infile"
  bei Server UND Client auf "ON" gesetzt sein.
* Damit LOAD DATA INFILE ... (sowie LOAD_FILE() und SELECT ... INTO OUTFILE ...)
  funktionieren, muss das Recht "FILE" für den Benutzer gesetzt sein.
* Die Server-Variable "secure_file_priv" legt evtl. Verz. fest, in dem export.
  und importierten Dateien liegen MÜSSEN (z.B. "/var/lib/mysql-files").

Inhalt der Datei "pdata.txt" (NULL-Wert durch "\N" codiert!):

```
  +------------------------------+   #
  |Nr<TAB>Vorname<TAB>Name<NL>   |   # 1. Zeile mit Spaltennamen --> Ignorieren
  +------------------------------+   #
  |1<TAB>Thomas<TAB>Birnthaler<NL>|   # 1. Datensatz
  |2<TAB>Markus<TAB>Mueller<NL>  |   # 2. Datensatz
  |8<TAB>Andrea<TAB>Bayer<NL>    |   # 3. Datensatz
  |9<TAB>Richard<TAB>Seiler<NL>  |   # 4. Datensatz
  |7<TAB>Heinz<TAB>Bayer<NL>     |   # 5. Datensatz
  |\N<TAB>Hans<TAB>Dampf<NL>     |   # 6. Datensatz (\N = NULL-Wert)
  |\N<TAB>\N<TAB>Unbekannt<NL>   |   # 7. Datensatz (\N = NULL-Wert)
  +------------------------------+   #
```

Um eine CSV-Datei zu erzeugen, folgende SELECT-Syntax verwenden
(kein LOCAL, d.h. Dateiablage nur auf Server möglich, NICHT auf Client!):

```
  SELECT ...
    INTO OUTFILE <FilePath> [<SaveOpt>]
      FROM ...;
```

Als <SaveOpt> sind möglich (in dieser Reihenfolge anzugeben!, "FIELDS" und
LINES nur 1x!, alle Elemente können fehlen, ebenso die Spalten)

```
  FIELDS TERMINATED BY <Char>              # STD: "\t" = Tabulator
         [OPTIONALLY] ENCLOSED BY <Char>   # STD: ""   = keines
         ESCAPED BY <Char>                 # STD: "\\" = \
```

```
  LINES STARTING BY <String>              # STD: ""   = leer
        TERMINATED BY <String>            # STD: "\n" = \n
```

Kopieren einer Tabelle: Struktur und/oder Indices und/oder Datensätze.
ACHTUNG: Primary Key und Indices werden NICHT mitkopiert (ausser bei LIKE!):

```
                                                  # Struktur  Indices  Daten
  CREATE TABLE copy    SELECT * FROM pers;        #    OK       Nein    OK
  CREATE TABLE copy AS SELECT * FROM pers;        #    OK       Nein    OK
  CREATE TABLE copy    SELECT * FROM pers WHERE 0; #    OK       Nein   Nein (TRICK!)
  CREATE TABLE copy AS SELECT * FROM pers WHERE 0; #    OK       Nein   Nein
  CREATE TABLE copy    LIKE pers;                 #    OK        OK    Nein (MY!5.0)
  INSERT INTO  copy    SELECT * FROM pers;        #   Nein      Nein    OK
```

Um eine Tabelle VOLLSTÄNDIG zu kopieren (Struktur + Indices + Datensätze),
die beiden folgenden Anweisungen kombinieren:

```
                                                  # Struktur  Indices  Daten
  CREATE TABLE copy LIKE pers;                    #    OK        OK    Nein (MY!5.0)
  INSERT INTO  copy SELECT * FROM pers;           #   Nein      Nein    OK
```

Kopieren bestimmter Spalten der Datensätze aus einer Tabelle in eine andere
Tabelle (Anzahl Spalten von Quelle und Ziel MUSS übereinstimmen!):

```
  INSERT INTO copy (nr, vorname, name)   # "vorname" und "name" vertauschen!
    SELECT nr, name, vorname             #
      FROM pers;                         #
```

Tabelleninhalt abfragen (ALLE Datensätze mit ALLEN Spalten in ihrer
Definitionsreihenfolge bzw. falls Primary Key vorhanden sortiert nach diesem):

```
  SELECT * FROM copy;       #
  SELECT ALL * FROM copy;   # (analog, ALL ist STD)
```

Tabelleninhalt abfragen (ALLE Datensätze mit bestimmten Spalten in der
angegebenen Reihenfolge):

```
  SELECT name, vorname FROM pers;
```

Tabelleninhalt abfragen (IDENTISCHE Datensätze nur 1x anzeigen, analog GROUP BY
über gewählte Spalten, DISTINCT und DISTINCTROW verhalten sich identisch):

```
  SELECT DISTINCT    *             FROM copy;   # Alle Spalten
  SELECT DISTINCTROW *             FROM copy;   #   (analog)
  SELECT DISTINCT    name          FROM copy;   # Eine Spalte
  SELECT DISTINCT    name, vorname FROM copy;   # Zwei Spalten
```

Spezifische Spaltentexte (Aliase) für Ausgabe als Überschrift festlegen,
Standard ist der Spaltenname laut Tabellen-Definition (bei Spalten-Aliasen ist
GROSS/kleinschreibung nicht relevant, "AS" ist auch weglassbar):

```
  SELECT vorname "Vorname", name "Familienname" FROM pers;   # Ohne AS
  SELECT vorname AS "Vorname",                               # Mit AS
         name    AS "Familienname"                           # (besser lesbar)
    FROM pers;
```

Konstanten und Ergebnis einer Rechnung mit spezifischer Überschrift ausgeben
(STD: Konstante, Ergebnis oder Formel auch als Überschrift verwendet):

```
  SELECT 2010          AS "Jahr",
         "Ergebnis = " AS "Text",
         5 * 4 - 3 / 2 AS "Formel";
```

Beispiel:

```
  CREATE TABLE IF NOT EXISTS artikel (
    nr    INT,
```

```
    name  VARCHAR(50),
    preis NUMERIC(10,2),
    anz   INT
  );

  INSERT INTO artikel VALUES (1, "Locher", 4.95, 18),
                             (2, "Hefter", 9.90, 12),
                             (3, "Papier", 5.49, 123);

  SELECT name                   AS `Artikel`,      # Warum OK?
         preis                  AS "Nettopreis",   #
         round(preis * 0.19, 2) AS 'MwSt',         # Rundung auf 2 NkSt
         round(preis * 1.19, 2) AS "Bruttopreis"   # Rundung auf 2 NkSt
    FROM artikel;
```

Tabelleninhalt nach Spalte "name" AUFsteigend sortiert abfragen (ohne GROSS/kleinschreibung zu berücksichtigen; STD: ASC = ascending = aufsteigend, der Deutlichkeit halber TROTZDEM hinschreiben!). Neben SpaltenNAMEN sind auch SpaltenNUMMERN in ORDER BY erlaubt. Diese Nummern beziehen sich auf die Reihenfolge der selektierten Spalten (bei "*" ist das die Reihenfolge der Spalten in der Tabellen-Definition):

```
  SELECT * FROM pers ORDER BY name;              # GROSS/kleinschr. ignorieren
  SELECT * FROM pers ORDER BY name ASC;          #   (analog, besser!)
  SELECT * FROM pers ORDER BY BINARY name ASC;   # GROSS/kleinschr. beachten
  SELECT * FROM pers ORDER BY 3, 2;              # Nach Spalte 3+2 sortieren
  SELECT * FROM pers ORDER BY name, vorname;     #   (analog, Sp.name statt nr)
```

Tabelleninhalt nach Spalte "nr" ABsteigend sortiert abfragen (DESC = descending, BINARY = GROSS/kleinschreibung beachten):

```
  SELECT * FROM pers ORDER BY nr DESC;           # GROSS/kleinschr. ignorieren
  SELECT * FROM pers ORDER BY BINARY nr DESC;    # GROSS/kleinschr. beachten
```

Sortierung des Tabelleninhalts nach Spalte "name" AUFsteigend, bei gleichem Nachnamen nach Spalte "vorname" AUFsteigend und bei gleichem Vor+Nachnamen nach Spalte "nr" ABsteigend (ohne GROSS/kleinschreibung zu berücksichtigen):

```
  SELECT * FROM pers
    ORDER BY name     ASC,    # AUFsteigend (ASC besser immer hinschreiben!)
             vorname  ASC,    # AUFsteigend
             nr       DESC;   # ABsteigend
```

Teil der Datensätze einer Tabelle über eine Bedingung abfragen
* Sogenannte "WHERE-Klausel/clause"
* Textvergleiche ignorieren GROSS/kleinschreibung (außer BINARY vor Operator)
* LIKE   = Unscharfe Suche mit Wildcards % (beliebig viele belieb. Z.)
  _ (1 beliebiges Zeichen)
* REGEX/RLIKE = Unscharfe Suche mit Regulären Ausdrücken (s.u.)
* i/-   = Index nutzbar/NICHT nutzbar (beschleunigt evtl. Suche)

```
  SELECT nr                                          # i = Index nutzbar
    FROM pers                                        # - = Index NICHT nutzbar
      WHERE name > "Andrea" AND name < "Thomas";     # i Ränder weglassen
      WHERE name >= "Andrea" AND name <= "Thomas";   # i Ränder einschließen (A)
  ... WHERE name BETWEEN "Andrea" AND "Thomas";      # i Ränder einschließen (B)
  ... WHERE name NOT BETWEEN "Andrea" AND "Thomas";  # i Ränder weglassen
  ... WHERE name IS NULL;                            # i Spalte LEER
  ... WHERE name IS NOT NULL;                        # i Spalte NICHT LEER
  ... WHERE name = NULL;                             # - FALSCH! immer NULL --> FALSE!
  ... WHERE name <> NULL;                            # - FALSCH! immer NULL --> FALSE!
  ... WHERE name = "Andrea" OR name = "Thomas";      # i Werteliste (A)
  ... WHERE name IN ("Andrea", "Thomas");            # i Werteliste (B)
  ... WHERE name NOT IN ("Andrea", "Thomas");        # i Ausschlussliste
  ... WHERE name LIKE "a%";                          # i "a" am Anfang
  ... WHERE BINARY name LIKE "%a%";                  # - "a" irgendwo
```

```
  ... WHERE name NOT LIKE "%a";                    # - KEIN "a" am Ende
  ... WHERE name NOT LIKE "__a%";                  # - KEIN "a" an 3. Z.pos.
  ... WHERE name REGEXP "abc";                     # - "abc" enthalten
  ... WHERE name RLIKE "abc";                      # - "abc" enthalten
  ... WHERE name NOT REGEXP "abc";                 # - KEIN "abc" enthalten
  ... WHERE name NOT RLIKE "abc";                  # - KEIN "abc" enthalten
```

Weitere Tabelle anlegen und füllen ("age" statt "alter", da "ALTER" eine SQL-Anweisung ist, durch Einschließen in Backquotes `alter` wäre "alter" doch als Tabellenname möglich, "..." oder '...' funktioniert nicht!):

```
  CREATE TABLE age (
    nr            INT NOT NULL,
    geburtsdatum  DATE,
    jahre         INT,
    geschlecht    CHAR(1)
  );                                                # Mehrere Sätze einfügen
  INSERT INTO age VALUES (1, "1971-1-8",  39, "m"),  # Nur möglich, da Werte
                         (2, "2001-5-13",  9, "m"),  # ALLER Spalten in der
                         (8, "1969-12-1", 41, "w"),  # korrekten Reihenfolge
                         (9, "1975-7-3",  35, "m");  # angegeben werden!
  INSERT INTO age                                   # Andere Schreibweise
    SET nr           = 7,
        geburtsdatum = "1966-1-1",
        jahre        = 44,
        geschlecht   = "m";
```

Zwei (oder mehr) Tabellen gemeinsam abfragen und alle Kombinationen aller Datensätze ausgeben (Kreuzprodukt = "CROSS JOIN" von "pers" und "age"). Die Liste der Tabellen ist durch "," getrennt hintereinander anzugeben:

```
  SELECT *                          # CROSS JOIN (alle Kombinationen)
    FROM pers, age;                 #
```

Kreuzprodukt von zwei (oder mehr) Tabellen durchführen und Ergebnismenge über gleiche Werte in der Spalte "nr" einschränken ("INNER JOIN" von "pers" und "age"). Die mehrdeutige Spalte "nr" wird durch vorangestellte Tabellennamen "pers" und "age" qualifiziert (genau spezifiziert):

```
  SELECT *                          # INNER JOIN mit WHERE (implizit)
    FROM pers, age                  #
      WHERE pers.nr = age.nr;       #
```

Alternativ "Aliase" (eindeutige Kurznamen, GROSS/kleinschreibung relevant) "p" und "a" für die Tabellen "pers" und "age" einführen (besser lesbar und kürzer):

```
  SELECT *                          # INNER JOIN mit WHERE (implizit)
    FROM pers AS p, age AS a        # Auch ... FROM pers p, age a ...
      WHERE p.nr = a.nr;            #
```

Gleiche Abfrage in expliziter JOIN-Syntax (Variante B+C ist nur möglich, falls die Verknüpfungsspalte in beiden Tabellen gleich heißt, wie hier "nr"):

```
  SELECT *                          # A) INNER JOIN mit ON
    FROM pers JOIN age              # (Spalte "nr" doppelt)
      ON pers.nr = age.nr;          # Allgemeine Syntax

  SELECT *                          # B) INNER JOIN mit USING
    FROM pers JOIN age              # (Spalte "nr" in beiden Tab. vorhanden)
      USING (nr);

  SELECT *                          # C) NATURAL JOIN (nur möglich, falls
    FROM pers NATURAL JOIN age;     #  "nr" einzige gleichnamige Spalte ist)
```

TIP: Index oder Primary Key auf diversen Spalten anlegen (damit laufen obige SELECT-Abfragen mit Verknüpfung beider Tabellen SEHR VIEL SCHNELLER!).

Nur Teil der Datensätze ab bestimmter Position holen (<Rows> Datensätze ab Datensatz <Offset>, STD: alle ab 0), die Datensätze sind beginnend mit "0" nummeriert --- MY!).

```
SELECT * FROM pers LIMIT 10;          # 10 Sätze ab Datensatz 0
SELECT * FROM pers LIMIT 3, 5;        #  5 Sätze ab Datensatz 3
SELECT * FROM pers LIMIT 5 OFFSET 3;  #  5 Sätze ab Datensatz 3
```

TIP: Abfrage SORTIEREN, sonst ist die Reihenfolge der Datensätze zufällig gemäß ihrer "physikalischen" Engine-Reihenfolge.

```
SELECT * FROM pers ORDER BY name, vorname LIMIT 10;
```

TIP: Immer einen Primärschlüssel anlegen, der jeden Datensatz eindeutig identifiziert. Falls dies aus den Daten heraus nicht möglich ist, dafür eine Spalte mit einem "künstlichen Schlüssel" (eindeutige Nummer) anlegen.

Index gleich beim Erstellen einer Tabelle mit anlegen (Variante A):

```
CREATE TABLE pers (
  nr  INT NOT NULL,                       # NOT NULL bei PRIMARY KEY!
  ...                                     #
  PRIMARY KEY         ON (nr),            # Primärschlüssel
  UNIQUE   INDEX idx2 ON (name),          # Eindeutige Spalte
           INDEX idx3 ON (name, vorname), # 2 Spalten verkettet
  FULLTEXT INDEX idx4 ON (vorname),       # Volltextsuche
  SPATIAL  INDEX idx4 ON (nr)             # Geometriedaten
);
```

Index nach Erstellen einer Tabelle hinzufügen (Variante B, jederzeit möglich):

```
CREATE PRIMARY KEY         ON pers (nr);             # FALSCH! --> ALTER TABLE
CREATE UNIQUE INDEX   idx2 ON age  (nr);             # Eindeutige Spalte(n)
CREATE INDEX          idx3 ON pers (name, vorname);  # 2 Spalten verkettet
CREATE FULLTEXT INDEX idx4 ON age  (name);           # Volltextsuche
CREATE SPATIAL INDEX  idx5 ON age  (nr);             # Geometriedaten
```

Index / Primary Key zu Tabelle hinzufügen (Variante C, jederzeit möglich):

```
ALTER TABLE pers ADD PRIMARY KEY         (nr);     # Name unnötig ("PRIMARY")
ALTER TABLE age  ADD UNIQUE INDEX   idx2 (nr);     # Eindeutige Spalte(n)
ALTER TABLE pers ADD INDEX          idx3 (name, vorname); # 2 Spalten verk.
ALTER TABLE age  ADD FULLTEXT INDEX idx4 (name);   # Volltextsuche
ALTER TABLE age  ADD SPATIAL INDEX  idx5 (nr);     # Geometriedaten
```

Indices zu einer Tabelle anzeigen:

```
SHOW INDEX FROM pers;
SHOW INDEX FROM age;
```

Index oder Primary Key einer Tabelle entfernen (jederzeit möglich):

```
ALTER TABLE pers DROP PRIMARY KEY;   # Kein Name nötig
ALTER TABLE pers DROP INDEX idx2;    # Name "idx2" nötig, UNIQUE weglassen!
DROP INDEX idx3 ON pers;             # Name "idx3" nötig
```

TIP: Index auf Präfix von Spalten bzw. AUF/ABsteigend erzeugen (STD: ASC), für Tuningzwecke zur Platzersparnis interessant (MY!).

```
CREATE INDEX idx8 ON pers (name(5), vorname(5));     # Platz sparen!
CREATE INDEX idx9 ON pers (name ASC, vorname DESC);  # Auf+Absteigend
```

Alle Datensätze ändern (VORSICHT: keine WHERE-Bedingung --> ALLE Datensätze!):

```
UPDATE pers
```

```
    SET nr = nr + 100;
```

Bestimmte Datensätze ändern:

```
  UPDATE pers                       # UPDATE FROM <Tbl> geht in MySQL nicht!
    SET nr = 111,                   #
        name = DEFAULT              # DEFAULT-Wert aus Tab.definition einsetzen
      WHERE vorname = "Thomas";     #
```

Datensätze ersetzen oder einfügen: Analog INSERT falls Datensatz mit diesem Primärschlüssel/UNIQUE INDEX noch nicht vorhanden ist, sonst DELETE des alten + INSERT des neuen Datensatzes durchführen (Primärschlüssel notwendig, DELETE-Recht notwendig, AUTO_INCREMENT-Feld erhöht sich --- MY!):

```
  REPLACE INTO copy (nr, vorname, name)   # (INTO optional)
    VALUES (1, "Thomas", "Birnthaler"),   # Mehrere Datensätze erlaubt
           (2, "Hans",   "Dampf");        #
                                          #
  REPLACE INTO copy (nr, vorname, name)   # (INTO optional)
    SELECT nr, vorname, name              #
      FROM pers,                          #
        WHERE nr < 100;                   #
                                          #
  REPLACE pers                            # (INTO optional)
    SET nr      = 7,                      #
        vorname = "Heinz",                #
        name    = "Beier";                #
```

Datensätze ersetzen oder einfügen: Analog INSERT falls Datensatz mit diesem Primärschlüssel/UNIQUE INDEX noch nicht vorhanden, sonst UPDATE des alten Datensatzes durchführen (bessere Variante, Primärschlüssel notwendig, AUTO_INCREMENT-Feld bleibt gleich --- MY!):

```
  INSERT INTO pers (nr, vorname, name)
    VALUES (1, "Thomas", "Birnthaler")
      ON DUPLICATE KEY UPDATE vorname = "Thomas",
                              name    = "Birnthaler";
```

HINWEIS: REPLACE ändert AUTO_INCREMENT-Wert bereits vorhandener Datensätze, INSERT INTO ON DUPLICATE KEYS UPDATE macht das nicht (schwierigere Syntax)!

Bestimmte Datensätze löschen:

```
  DELETE FROM pers
    WHERE vorname = "Markus" OR nr >= 9;
```

Alle Datensätze löschen, Tabelle belassen (VORSICHT: keine WHERE-Bedingung!):

```
  DELETE FROM pers;      # Langsam (Datensätze einzeln löschen = Transaktion!)
  TRUNCATE TABLE pers;   # Schnell (Tabelle löschen + neu anlegen, nicht in TA!)
  TRUNCATE pers;         #   (analog)
```

HINWEIS: Auch Lösch- und Update-Operation sind per LIMIT auf eine bestimmte Anzahl Datensätze beschränkbar (ob das sinnvoll ist, sei dahingestellt!). Dabei sollten die Datensätze mit ORDER BY sortiert werden, um sie nicht zufällig gemäß ihrer "physikalischen" Reihenfolge zu löschen oder zu ändern:

```
  DELETE FROM pers             # Max. 2 Datensätze löschen
    WHERE vorname = "Markus"   #
      ORDER BY name            # Wichtig!
        LIMIT 2;               #

  UPDATE pers                  # Max. 3 Datensätze ändern
    SET nr = nr + 1000         #
      WHERE nr < 1000          #
        ORDER BY name          # Wichtig!
```

```
          LIMIT 3;            #
```

TIP: Mit der "mysql"-Client-Option --i-am-a-dummy / --safe-updates / -U sind UPDATE- und DELETE-Anweisungen gegen Weglassen einer WHERE- oder LIMIT-Klausel geschützt (d.h. versehentliches Ändern/Löschen ALLER Datensätze --- außer per TRUNCATE --- wird verhindert).

Anzahl Datensätze, Spaltenwerte oder Häufigkeit verschiedener Spaltenwerte in einer Tabelle ermitteln (Gruppieren bzw. Aggregieren):

```
  SELECT COUNT(*) FROM pers;             # Anz. Datensätze in Tab. (inkl. NULL!)
  SELECT COUNT(nr) FROM pers;            # Anz. Werte von Sp. "nr" (ohne NULL!)
  SELECT COUNT(DISTINCT nr) FROM pers;   # Anz. versch. Werte von Sp. "nr" (ohne NULL!)
  SELECT name, COUNT(name) FROM pers     # Häufigkeit der Werte in Sp. "name"
    GROUP BY name;                       #
  SELECT name, COUNT(name) FROM pers     #   (analog, nur Werte mit Häuf. > 3)
    GROUP BY name                        #
      HAVING COUNT(name) > 3;            #
  SELECT name, COUNT(name) FROM pers     #   (analog + Gesamtsumme (NULL als Wert!)
    GROUP BY name                        #
      WITH ROLLUP;                       # (MY!) ACHTUNG: HAVING wirkt darauf!
```

Weitere typische Aggregatfunktionen analog COUNT (arbeiten auf der Menge aller von einer Abfrage gefundenen Datensätze):

```
  SELECT SUM(nr),            # Summe aller Werte von Spalte "nr"
         SUM(DISTINCT nr),   # Summe aller verschiedenen Werte von Spalte "nr"
         AVG(nr),            # Durchschnitt aller Werte von Spalte "nr"
         MIN(nr),            # Minimum aller Werte von Spalte "nr"
         MAX(nr)             # Maximum aller Werte von Spalte "nr"
    FROM pers;
```

Datensatz mit höchster Nummer:

```
  SELECT DISTINCT * FROM pers WHERE nr = MAX(nr);  # FALSCH!
  SELECT DISTINCT * FROM pers                      # A) Subselect für Maximum
    WHERE nr = (SELECT MAX(nr) FROM pers);         #
  SET @max := (SELECT MAX(nr) FROM pers);          # B) Variable @max füllen
  SET @max = (SELECT MAX(nr) FROM pers);           #    Variable @max füllen
  SELECT * FROM pers WHERE nr = @max;              #    (:= / = ist Zuweisung)
  SELECT * FROM pers ORDER BY nr DESC LIMIT 1;     # C) Abbruch per LIMIT
  SELECT p1.* FROM pers p1 LEFT JOIN pers p2       # D) FALSCH!
    ON p1.nr <= p2.nr WHERE p2.nr IS NULL;         #
```

Datensatz mit niedrigster oder höchster Nummer:

```
  SELECT * FROM pers WHERE nr = MIN(nr) OR nr = MAX(nr);   # A) FALSCH!
  SELECT * FROM pers WHERE nr IN (MIN(nr), MAX(nr));       # B) FALSCH!
  SELECT * FROM pers WHERE nr IN (                         # C) Subselect
    (SELECT MIN(nr) FROM pers),                            #
    (SELECT MAX(nr) FROM pers)                             #
  );                                                       #
  SELECT @min := MIN(nr), @max := MAX(nr) FROM pers;       # D) Variablen + ...
  SELECT MIN(nr), MAX(nr) INTO @min, @max;                 #      (analog MY!)
  SELECT * FROM pers WHERE nr = @min OR nr = @max;         # D1) ... log. Bed
  SELECT * FROM pers WHERE nr IN (@min, @max);             # D2) ... Menge
  SELECT * FROM pers WHERE nr = @min                       # D3) ... Union
    UNION ALL
  SELECT * FROM pers WHERE nr = @max;
```

Allgemeines SELECT-Statement (alle Möglichkeiten):

```
  SELECT <Col>, ...              # Spaltenauswahl
    FROM <Tbl>, ...              # Tabellenauswahl
      [WHERE <Cond>]             # Bedingung auf Spalten (Condition)
        [GROUP BY <Col>, ...     # Aggregation auf Spalten (Datensätze zusammenfassen)
```

```
          [HAVING <Cond>]]      # Bedingung auf Aggregation (nur bei GROUP BY!)
            [ORDER BY ...]      # Sortierung nach Spalten (Name oder Nummer!)
              [LIMIT ...]       # Anz. Datensätze + Startpos. begrenzen (MY!)
```

Beispiel:

```
  SELECT nr,
         COUNT(name) AS "Anz",
         name AS "Name"
    FROM pers
      WHERE nr > 2
        GROUP BY name
          ORDER BY nr DESC
            LIMIT 2;
```

HINWEIS: MySQL erlaubt SELECT-Anweisungen ohne Tabellen-Bezug, um z.B. das Ergebnis von Ausdrücken zu berechnen (in Oracle ist dazu das Anhängen von "FROM dual" notwendig!). Die bei MySQL (und Oracle) grundsätzlich vorhandene "Dummy"-Tabelle "dual" (enthält EINEN Datensatz ohne Spalten) gibt es daher nur aus Kompatibilitätsgründen zu Oracle:

```
  SELECT SQRT(2);             # In MySQL erlaubt (in Oracle nicht!)
  SELECT SQRT(2) FROM dual;   # In MySQL und Oracle erlaubt (Ergebnis gleich)
```

Tabellenstruktur und -name ändern:
  * Mehrere Änderungen gleichzeitig durch Komma getrennt angebbar
  * Dabei wird KOPIE mit Änderungen angelegt (dauert bei vielen Daten lange!) und anschließend das ORIGINAL durch die KOPIE ersetzt
  * Tabelle wird GESPERRT, d.h. weitere Anfragen an sie müssen solange warten
  * Reines Umbenennen einer Tabelle wird ohne Kopie erledigt
  * Verschieben einer Tabelle in andere DB nur im gleichen Dateisystem möglich
  * Beim Spaltentyp ändern werden Spaltendaten so weit möglich erhalten (evtl. abgeschnitten oder mit Leerzeichen aufgefüllt)
  * Tabelle/Engine ändern NICHT auf Systemtabellen in DB "mysql" anwenden!

Änderungs-Operationen auf Tabellen (mehrere gleichzeitig erlaubt, COLUMN, TO und CONSTRAINT dürfen weggelassen werden):

Operation	Bedeutung
ADD [COLUMN]	Spalte einfügen (FIRST, AFTER <Col>, STD: hinten)
DROP [COLUMN]	Spalte entfernen (mit Daten + Indices darauf)
MODIFY [COLUMN]	Sp.typ ändern (Sp.name bleibt) oder Sp. verschieben
CHANGE [COLUMN]	Spalte neu definieren (Name + Typ, d.h. auch umben.)
ALTER [COLUMN]...	Defaultwert...
...SET DEFAULT	...ändern
...DROP DEFAULT	...entfernen
RENAME [TO]	Tabellenname ändern, Tabelle in andere DB verschieben
ORDER BY	Datensätze für schnelle Abfrage sortieren (MY!)
ENGINE	Andere Datenbank-Engine verwenden (MY!)
TYPE	Andere Datenbank-Engine verwenden (veraltet, MY!)
IMPORT TABLESPACE	(MY!)
DISCARD TABLESPACE	(MY!)
ADD PRIMARY KEY	Primärschlüssel hinzu (muss NOT NULL UNIQUE sein)
ADD UNIQUE INDEX	Sekundärschlüssel hinzu (muss NOT NULL UNIQUE sein)
ADD {INDEX | KEY}	Index hinzu (muss nicht NOT NULL oder UNIQUE sein)
DROP PRIMARY KEY	Primärschlüssel entfernen
DROP {INDEX | KEY}	Index entfernen
DISABLE KEYS	Alle Indices der Tabelle abschalten (nur Non-UNIQUE!)
ENABLE KEYS	Alle Indices der Tabelle aktivieren (nur Non-UNIQUE!)
ADD FOREIGN KEY	Fremdschlüsselbezug hinzufügen
DROP FOREIGN KEY	Fremdschlüsselbezug entfernen

```
  | ADD CONSTRAINT     | Spalten-Beschränkung hinzufügen (ignoriert MY!)       |
  | DROP CONSTRAINT    | Spalten-Beschränkung entfernen (nicht verfügbar MY!)  |
  +--------------------+-------+-----------------------------------------------+
  | CONVERT TO CHARACTER SET ... | Zeichensatz konvertieren                    |
  |            COLLATE ...       |                                             |
  | [DEFAULT] CHARACTER SET  ... | Standard-Zeichensatz festlegen              |
  |            COLLATE ...       |                                             |
  +------------------------------+---------------------------------------------+
```

Beispiele:

```
  ALTER TABLE pers ...
  ... ADD COLUMN preis DECIMAL(10,2);       # STD: Spalte hinten anfügen
  ... ADD COLUMN (strasse CHAR(30), plz LONG, ort CHAR(30));
  ... ... FIRST;                            # Als 1. Spalte einfügen
  ... ... AFTER nr;                         # Nach Spalte "nr" einfügen
  ... DROP   COLUMN name;                   # Spalte entfernen (inkl. Daten)
  ... MODIFY COLUMN vorname VARCHAR(50);    # Typ ändern (Daten mögl. erhalten)
  ... CHANGE COLUMN name name VARCHAR(20);  # (analog)
  ... ... FIRST;                            # Als 1. Spalte verschieben
  ... ... AFTER nr;                         # Nach Spalte "nr" verschieben
  ... CHANGE COLUMN vorname name CHAR(20);  # Name+Typ änd. (Daten mögl. erh.)
  ... CHANGE COLUMN name vorname CHAR(20);  # Name änd. (Typ identisch wählen!)
  ... ALTER  COLUMN vorname SET DEFAULT ""; # Default änd.: Leere Zeichenkette
  ... ALTER  COLUMN vorname DROP DEFAULT;   # Default änd.: NULL
                                            #
  ... RENAME TO pers_old;                   # Tabelle umbenennen
  ... RENAME TO test2.pers;                 # Tabelle in andere DB verschieben
  ... ORDER BY vorname DESC, name ASC;      # Datensätze sortieren
  ... ENGINE = MyISAM;                      # Engine ändern (neu!)
  ... TYPE = MyISAM;                        # Engine ändern (veraltet!)
                                            #
  ... ADD PRIMARY KEY (nr);                 # Primärschlüssel hinzufügen
  ... DROP PRIMARY KEY;                     # Primärschlüssel entfernen
  ... INDEX idx1 (name, vorname);           # Index hinzufügen
  ... DROP INDEX idx1;                      # Index entfernen
  ... UNIQUE INDEX idx2 (name, vorname);    # Sekundärschlüssel hinzufügen
  ... DROP UNIQUE INDEX idx2;               # Sekundärschlüssel entfernen
  ... DISABLE KEYS;                         # Alle Indices abschalten (nur Non-UNIQUE)
  ... ENABLE KEYS;                          # Alle Indices aktivieren (nur Non-UNIQUE)
                                            #
  ... ADD FOREIGN KEY age(nr)               # Fremdschlüssel hinzufügen
                  REFERENCES pers(nr);      #
  ... DROP FOREIGN KEY age;                 # Fremdschlüssel entfernen
  ... ADD CONSTRAINT pruef CHECK (nr >= 1); # Spalten-Beschränkung hinzufügen
  ... DROP CONSTRAINT pruef;                # Nicht verfügbar!
                                            #
  ... CONVERT TO CHARACTER SET "latin1"     # Zeichensatz aller Sp. konvertieren
              COLLATE "latin1_german_ic";   #
  ... [DEFAULT] CARACTER SET = "latin1"     # Zeichensatz aller Sp. konvertieren
              COLLATE = "latin1_german_ic"; #
```

Tabelle(n) umbenennen (auch in andere Datenbank verschieben, wenn die beiden Datenbanken auf dem gleichen Dateisystem liegen). Stellt eine "atomare" Operation dar, auch bei gleichzeitiger Umbenennung mehrerer Tabellen. Läuft sehr schnell ab, da keine Kopie erstellt wird:

```
  RENAME TABLE pers TO new [, ...];   # Nicht für temp. Tabelle
  ALTER TABLE new RENAME TO pers;     # Auch für temp. Tabelle
  RENAME TABLE pers TO copy,          # Zwei Tabellenamen vertauschen (FALSCH!)
               copy TO pers;          # (-> ERROR: Table 'copy' already exists)
  RENAME TABLE pers TO tmp,           # Zwei Tabellenamen vertauschen (OK!)
               copy TO pers,          #
               tmp  TO copy;          #
```

Tabelle (bedingt) löschen (inkl. aller Daten und Indices!):

```
  DROP TABLE pers;              # Fehler falls nicht existent
  DROP TABLE IF EXISTS pers;    # Kein Fehler falls nicht existent
  DROP TEMPORARY TABLE pers;    # Temporäre Tabelle löschen (nicht echte!)
```

Datenbank (bedingt) löschen (alle Tabellen mit allen Daten!):

```
  DROP DATABASE first;             # Fehler falls nicht existent
  DROP SCHEMA   first;             #   (analog)
  DROP DATABASE IF EXISTS first;   # Kein Fehler falls nicht existent
  DROP SCHEMA   IF EXISTS first;   #   (analog)
```

Abfrage von Informationen zur vorhergehenden SQL-Anweisung (pro Sitzung):

```
  SELECT ROW_COUNT();              # Anz. verarb. DS (INSERT, UPDATE, DELETE, REPLACE)
  SELECT SQL_CALC_FOUND_ROWS ...;  # Vor Einsatz von FOUND_ROWS() notwendig!
  SELECT FOUND_ROWS();             # Gesamtanz. Datensätze bei SELECT mit LIMIT
  SELECT LAST_INSERT_ID();         # Letzter AUTO_INCREMENT-Wert bei INSERT
```

Fehlermeldungen, Warnungen und Bemerkungen (Notes) zur vorhergehenden SQL-Anweisung ausgeben (pro Sitzung):

```
  SHOW ERRORS;               # Nur Fehler ausgeben
  SHOW WARNINGS;             # Fehler, Warnungen, Bemerkungen (Notes) ausgeben
  SET max_error_count = 0;   # Aufzeichnung der Warnungen/Fehler ausschalten
  SET max_error_count = 64;  # Max. Anz. aufgezeichneter Warnungen/Fehler (STD)
  SET sql_notes = 1;         # Bemerkungen (Notes) 1=aufzeichnen/0=unterdrücken
  SET sql_warnings = 1;      # INSERT-Warnung 1=aufzeichnen/0=unterdrücken
  SELECT @@error_count;      # Anzahl gemerkter Fehler
  SELECT @@warning_count;    # Anzahl gemerkter Warnungen
```

4) MySQL-Datentypen

Im Prinzip könnten man die Spaltenwerte einer Tabelle immer in Form von Zeichenketten abspeichern. MySQL würde diese automatisch beim Zugriff in andere Datentypen wie Ganz/Fließkommazahlen, Datums/Zeitwerte sowie Boolesche Werte umwandeln. Der Konvertierungsaufwand wäre aber für jeden Datensatz bei jedem Zugriff durchzuführen und daher zu aufwendig.

Daher sollte abgestimmt auf die zu speichernden Werte für jede Tabellenspalte einer der folgenden MySQL-Datentypen ausgewählt werden. Diese Auswahl ist nicht immer eindeutig (z.B. könnte ein Jahr als INT, FLOAT, DECIMAL oder YEAR gespeichert werden). Bei sinnvoller Auswahl des Datentyps gilt:

* Erlaubter Wertebereich sinnvoll eingeschränkt (ungültige Daten abgelehnt)
  (neben dem Spaltennamen ein wichtiger Teil der Dokumentation)
* Platzbedarf minimiert
* Zugriffsgeschwindigkeit maximiert
* Indices effizient nutzbar für Zugriff und Sortierung
* Typspezifische MySQL-Funktionen und -Operatoren einsetzbar

TIP: Auch an evtl. zukünftige Erweiterungen denken und z.B. nicht für das Speichern einer Jahreszahl nur 2 Stellen ("Y2K-Problem") oder einer IP-Adresse oder Netzwerk-Maske nur 4 Byte (IPv4) vorsehen (IPv6 benötigt 16 Byte).

Datentyp	Typname	Byte	Wertebereich
Ganzzahl (mit Vorzeichen)	TINYINT	1	-127..128
	SMALLINT	2	-32768..32767
	MEDIUMINT	3	-8388608..8388607
	INT	4	-2147483648..2147483647
	BIGINT	8	-9223372036854775808..
			..9223372036854775807

Ganzzahl (ohne Vz.)	TINYINT UNSIGNED	1	0..255	
	SMALLINT UNSIGNED	2	0..65535	
	MEDIUMINT UNSIGNED	3	0..16777215 (16 Mio)	
	INT UNSIGNED	4	0..4294967295 (4 Mrd)	
	BIGINT UNSIGNED	8	0..18446744073709551615	
Fließkommazahl (technisch, nicht kaufm.)	FLOAT	4	10^38 (7 Dez.stellen)	
	DOUBLE	8	10^308 (15 Dez.stellen)	
Fließkommazahl (ohne Vorz.)	FLOAT UNSIGNED	4	10^38 (7 Dez.stellen)	
	DOUBLE UNSIGNED	8	10^308 (15 Dez.stellen)	
Festkommazahl (L max. 65) (vor MY!5.0 als String)	DECIMAL(L,N)	L+2 L/9*4	Länge L mit N Nkst. (ab MY!5.0!)	
Festkommazahl (ohne Vorz.)	DECIMAL(L,N) UNSIGNED	L+2 L/9*4	Länge L mit N Nkst. (ab MY!5.0!)	
Boolean	BOOL	1	0/1 FALSE=0/TRUE=1	
Bitfeld	BIT(N)	(N+7)/8	N Bits (1..64)	
Zeichenkette (Länge fest)	CHAR(L)	L	L=0..255	
(Länge fest)	BINARY(L)	L	L=0..255	
(Länge variabel)	VARCHAR(L)	L+1/2	L=0..255/65535 (MY!5.1)	
(Länge variabel)	VARBINARY(L)	L+1/2	L=0..255/65535 (MY!5.1)	
Datum + Zeit	YEAR	1	JJJJ/JJ (1900-2155)	
	DATE	3	JJJJ-MM-TT (1000-9999)	
	TIME	3	hh:mm:ss	
	DATETIME	8	JJJJ-MM-TT hh:mm:ss	
(Sek. seit 1.1.1970 00:01)	TIMESTAMP	4	JJJJMMTThhmmss	
Mikrosekunden <Fsp> 0..6 (Fractional seconds part) seit MY!5.6.4 (0..3 Byte)	TIME(Fsp)	3+0-3	hh:mm:ss.ffffff fff	
	DATETIME(Fsp)	5+0-3	JJJJ-MM-TT hh:mm:ss.fff	
	TIMESTAMP(Fsp)	4+0-3	JJJJMMTThhmmss.ffffff	
Text (Sonderzeichen interpretiert)	TINYTEXT	L+1	0-255	
	TEXT	L+2	0-65535	
	MEDIUMTEXT	L+3	0-16777216 (16 Mio)	
	LONGTEXT	L+4	0-4294967296 (4 Mrd)	
Binary large object BLOB (binäre Daten, keine Interpretation von Sonderzeichen)	TINYBLOB	L+1	0-255	
	BLOB	L+2	0-65535	
	MEDIUMBLOB	L+3	0-16777216 (16 Mio)	
	LONGBLOB	L+4	0-4294967296 (4 Mrd)	
Aufzählung (Single-Value)	ENUM(V1, ...)	1/2	1..255, 2=256-65535	
Menge (Multi-Value)	SET(V1, ...)	1-4/8	1..8/16/24/32/64 Elem.	
Unicodestring	NCHAR(L)	L	Synonym für CHAR	
	NATIONAL VARCHAR(L)	L+1	Synonym für VARCHAR	

Mögliche Attribute zu allen Datentypen (danach anzugeben):

Attribut	Bedeutung
DEFAULT <Val>	Defaultwert <Val> setzen, falls KEIN Wert angegeben
NOT NULL	Wert MUSS angegeben sein (evtl. per Defaultwert)
NULL	Wert darf weggelassen werden ("undefiniert", STD)

HINWEIS: Der Defaultwert <Val> muss konstant sein, Funktionen sind unzulässig.

Mögliche Attribute zu numerischen Datentypen (Ganzzahl, Fließkomma, Festkomma):

Attribut	Bedeutung	
(<N>)	Ausgabebreite (INT)	
(<N>, <M>)	Ausgabebreite N und Nkst. M (FLOAT, DOUBLE, DECIMAL)	
UNSIGNED	Wert immer positiv	
ZEROFILL	Ausgabe mit führenden Nullen gemäß Breite (nicht DEC!)	
AUTO_INCREMENT	Pro NEU eingefügtem Datensatz automatisch hochzählen (künstlicher Key, nur 1x pro Tabelle, NICHT DECIMAL!)	!!!
PRIMARY KEY	Spalte ist Primärschlüssel (--> NOT NULL)	
CHARACTER SET <CharSet> COLLATE <Sort>	Zeichensatz und Sortierordnung	

Mögliche Attribute zu TIMESTAMP:

Attribut	Bedeutung
DEFAULT CURRENT_TIMESTAMP	Neuer Datensatz --> aktuelle Zeit speichern
ON UPDATE CURRENT_TIMESTAMP	Datensatz geändert --> akt. Zeit speichern

Beispiel:

```
DROP TABLE IF EXISTS data;
CREATE TABLE data (
  nr        INT NOT NULL AUTO_INCREMENT,      #0  Automatisch füllen
  anz       TINYINT(3) UNSIGNED,              #1  Ausgabebreite 3
  grad      FLOAT(7,1),                       #2  Ausgabeformat 12345.6
  betrag    DECIMAL(9,2),                     #3  Ausgabeformat 123456.78
  vorname   CHAR(30),                         #4  Zeichen (mit Leerz. aufgef.)
  name      VARCHAR(30) NOT NULL,             #5  Zeichen
  binaer1   BINARY(10),                       #6  Byte (mit 0-Byte aufgefüllt)
  binaer2   VARBINARY(8),                     #7  Byte
  ts1       TIMESTAMP,                        #8  Automatisch gefüllt (1.Sp.)
  ts2       TIMESTAMP(3),                     #9  NICHT " (2.Sp., Milli-Sek.)
  ts3       TIMESTAMP(6),                     #10 NICHT " (3.Sp., Mikro-Sek.)
  datum     DATE DEFAULT '2000-01-01',        #11
  uhrzeit   TIME DEFAULT '12:00:00',          #12
  datumzeit DATETIME,                         #13 JJJJ-MM-TT hh:mm:ss
  jahr      YEAR(4),                          #14
  wahr      BOOL,                             #15 TRUE/FALSE
  flags     BIT(8),                           #16 Bitmuster 8 Bit = 1 Byte
  dokument  TEXT,                             #17
  antwort   ENUM('Ja', 'Nein', 'Vielleicht'), #18 Wert aus Liste
  menge     SET('gross','reich','maechtig'),  #19 Menge aus Listenwerten
  datei     BLOB,                             #20
  PRIMARY KEY (nr)   # Kein Komma, notwendig wg. AUTO_INCREMENT von Spalte nr
);

INSERT INTO data VALUES (
  NULL,                    #0  INT:          Nächste freie Nr. wg. AUTO_INCREMENT
  123,                     #1  TINYINT(3):   füllen
  12345.6,                 #2  FLOAT(7,1):   füllen
  876543.21,               #3  DECIMAL(9,2): füllen
  "Thomas",                #4  CHAR(30):     füllen (mit Leerz. hinten auffüllen)
  "Birnthaler",            #5  VARCHAR(30):  füllen
  0x4142434445464748494A,  #6  BINARY(10):   füllen (mit 0-Byte hinten auffüllen)
  x'6162636465666768',     #7  VARBINARY(8): füllen
  NULL,                    #8  TIMESTAMP:    Aktuellen Zeitp. autom. eintragen
```

```
    "20120907112233.987654",  #9  TIMESTAMP:    füllen
    "2012-09-07 11:22:33.123456", #10 TIMESTAMP:füllen ("7.9.2012 11:22:33" NICHT OK!)
    "2012-08-03",             #11 DATE:         füllen
    "23:59:59",               #12 TIME:         füllen
    "2012-08-03 23:59:59",    #13 DATETIME:     füllen ("3.8.2012 23:59:59" NICHT OK!)
    2012,                     #14 YEAR(4):      füllen
    TRUE,                     #15 BOOL:         füllen
    b'01000001',              #16 BIT(8):       füllen ASCII-Code 65 = Zeichen 'A'
  # "Dokument(text)",         #17 TEXT:         füllen (direkt) oder
  # LOAD_FILE("/etc/my.cnf"), #17 TEXT:         laden aus externer Datei
    LOAD_FILE("C:/ProgramData/MySQL/MySQL Server 5.6/my.ini"), #17 Windows
    "Vielleicht",             #18 ENUM:         Aufz.wert als String (EINER!)
    "gross,reich,maechtig",   #19 SET:          Elem. per "," getr. als String
  # "Blobdaten(binaer)"       #20 BLOB:         füllen (direkt) oder
  # LOAD_FILE("/etc/my.cnf")  #20 BLOB:         laden aus externer Datei
    LOAD_FILE("C:/ProgramData/MySQL/MySQL Server 5.6/my.ini") #20 Windows
  # 0x41424344454647484 94A,  #20 BLOB:         füllen (mit 0-Byte hinten)
  # x'6162636465666768',      #20 BLOB:         füllen
  );

  SELECT * FROM data\G        # Ergibt (Spalten vertikal anzeigen)

    *************************** 1. row ***************************
           nr: 1                      #0  INT
         anz: 123                     #1  TINYINT(3)
        grad: 12345.6                 #2  FLOAT(7,1)
      betrag: 876543.21               #3  DECIMAL(9,2)
    vorname: Thomas                   #4  CHAR(30)
       name: Birnthaler               #5  VARCHAR(30)
    binaer1: ABCDEFGHIJ               #6  BINARY(10):   Zeichen zu Code 41 ... 4A
    binaer2: abcdefgh                 #7  VARBINARY(8): Zeichen zu Code 61 ... 68
        ts1: 2012-09-11 18:07:08      #8  TIMESTAMP
        ts2: 2012-09-07 11:22:33      #9  TIMESTAMP
        ts3: 2012-09-07 11:22:33      #10 TIMESTAMP
      datum: 2012-08-03               #11 DATE
    uhrzeit: 23:59:59                 #12 TIME
  datumzeit: 2012-08-03 23:59:59      #13 DATETIME
       jahr: 2012                     #14 YEAR(4)
       wahr: 1                        #15 BOOL
      flags: A                        #16 BIT(8)
   dokument: Dokument(text)           #17 TEXT
    antwort: Vielleicht               #18 ENUM
      menge: gross,reich,maechtig     #19 SET
      datei: Blobdaten(binaer)        #20 BLOB
  1 row in set (0.00 sec)
```

Alternative Namen für einige Datentypen:

Name	Alternativer Name
BOOL	BOOLEAN, TINYINT(1)
CHAR	CHARACTER
DECIMAL	DEC, NUMERIC
DOUBLE	REAL, DOUBLE PRECISION
INT	INTEGER
NCHAR	NATIONAL CHAR, NATIONAL CHARACTER
VARCHAR	CHARACTER VARYING

Beispiele für die Angabe von Konstanten (Literalen) für die Datentypen
(Q = Quotieren des Wertes mit "..." oder '...' notwendig):

Datentyp	Q	Beispiele für Konstanten/Literale

```
  | INT       | - | 0  123  -456  +7899123   0x10EF  x'10EF'  b'11010'  |
  | FLOAT     | - | 0.0  1.23  -0.456  1.23e14  -12E-34                |
  | DECIMAL   | - | 0.0  -123.000  +456789.00                          |
  +-----------+---+----------------------------------------------------+
  | BOOL      | - | TRUE  FALSE  0  1  "aaa"  ""                       |
  | BIT       | x | b'10101100'  0x10EF  x'10EF'                       |
  +-----------+---+----------------------------------------------------+
  | CHAR      | x | "Hallo"  'Hallo'  ""  ''                           |
  | VARCHAR   | x | "Hallo"  'Hallo'  ""  ''                           |
  | BINARY    | x | 0x10EF  x'10EF'  _utf8'uc'  ""  ''                 |
  | VARBINARY | x | 0x10EF  x'10EF'  _utf8'uc'  ""  ''                 |
  +-----------+---+----------------------------------------------------+
  | YEAR      | - | 69  99  13  1972  2010  2100                       |
  | DATE      | x | "2009-10-18"  "0000-00-00"  '2009-10-00'           |
  | TIME      | x | "23:59:59"  '09:45'  "00:00:00"                    |
  | DATETIME  | x | "2009-10-18 22:30:00"  YYYYMMDDhhmmss              |
  |           | x | "2009-10-18 22:30:00.123456"  YYYYMMDDhhmmss.nnnnnn |
  | TIMESTAMP |x/-| "2009-10-18 22:30:00"  20091018223000 (Zahl, sic!)  |
  |           |   | "2009-10-18 22:30:00.123456"                       |
  |           |   | 20091018223000.12345 (Zahl, sic!)                  |
  +-----------+---+----------------------------------------------------+
  | TEXT      | x | "Hallo"  'Hallo'  ""  ''  _utf8'uc'                |
  | BLOB      | x | 0x10EF  x'10EF'  b'11010'  "Daten..."  'Daten...'  |
  +-----------+---+----------------------------------------------------+
  | ENUM      | x | "1.Wert"  '5.Wert'                                 |
  | SET       | x | "1.Wert,5.Wert,8.Wert"  "5.Wert"  "" (leer)        |
  +-----------+---+----------------------------------------------------+
```

Defaultwert und Defaultattribut NULL/NOT NULL für Datentypen:

```
  +-----------+-----------------------------------------------+-------------+
  | Datentyp  | Defaultwert                                   | Defaultfall |
  +-----------+-----------------------------------------------+-------------+
  | INT       | 0                                             | NULL        |
  | FLOAT     | 0.0                                           | NULL        |
  | DECIMAL   | 0.0                                           | NULL        |
  +-----------+-----------------------------------------------+-------------+
  | BIT       | b'' (kein Bit gesetzt)                        | NULL        |
  | BOOL      | 0 (FALSE)                                     | NULL        |
  +-----------+-----------------------------------------------+-------------+
  | CHAR      | "" (leerer String)                            | NULL        |
  | VARCHAR   | "" (leerer String)                            | NULL        |
  | BINARY    | "" (leere Bytefolge)                          | NULL        |
  | VARBINARY | "" (leere Bytefolge)                          | NULL        |
  +-----------+-----------------------------------------------+-------------+
  | YEAR      | 00/0000                                       | NULL        |
  | DATE      | "0000-00-00"                                  | NULL        |
  | TIME      | "00:00:00"                                    | NULL        |
  | DATETIME  | "0000-00-00 00:00:00"                         | NULL        |
  | TIMESTAMP | CURRENT_TIMESTAMP()   (1.Spalte, akt. Datum+Zeit)| NOT NULL(!) |
  |           | "0000-00-00 00:00:00" (restliche Spalten)     | NULL        |
  +-----------+-----------------------------------------------+-------------+
  | TEXT      | "" (leerer String)                            | NULL        |
  | BLOB      | "" (leerer String)                            | NULL        |
  +-----------+-----------------------------------------------+-------------+
  | ENUM      | 0 (falls NULL), "1. Wert" (falls NOT NULL)    | NULL        |
  | SET       | "" (leere Menge)                              | NULL        |
  +-----------+-----------------------------------------------+-------------+
```

HINWEISE:

* Datentyp-Konvertierung erfolgt entweder automatisch oder explizit per
  (BINARY entspricht CAST(<String> AS BINARY)):

```
    BINARY <String>                  # GROSS/kleinschreibung berücksichtigen
    CAST(<Val> AS <Typ>)             # Datentyp von <Val> in <Typ> umwandeln
```

```
    CONVERT(<Val> AS <Typ>)            # Datentyp von <Val> in <Typ> umwandeln
    CONVERT(<Val> USING <Transcode>)   # Zeichensatz von <Val> konvertieren
```

* Zieldatentyp <Typ> einer Datentyp-Konvertierung kann sein:

BINARY[(n)]	Max. n Byte nutzen (mit 0x00 auffüllen)
CHAR[(n)]	Max. n Byte nutzen (mit Space auffüllen)
DATE	
DATETIME	
DECIMAL[(m[,d])]	
SIGNED [INTEGER]	
TIME	
UNSIGNED [INTEGER]	

* Folgende Datentyp-Konvertierungen sind möglich:

Von	Nach
DECIMAL	CHAR VARCHAR DATE TIME DATETIME
CHAR VARCHAR	DECIMAL DATE TIME DATETIME
DATE TIME DATETIME	CHAR VARCHAR
BLOB TEXT	(keine!)

* INT/FLOAT/DOUBLE: Bei Ganzzahlen ist in Klammern eine "Darstellungsgröße" (1-255) angebbar, bei Fließkommazahlen die "Darstellungsgröße" und die "Anzahl der Nachkommastellen" (DSG >= NKST + 2). Sie bestimmen nicht die Länge oder Genauigkeit der gespeicherten Zahl, sondern legen fest, mit welcher Breite und welcher Nachkommstellenanzahl die AUSGABE erfolgen soll.

* FLOAT/DOUBLE: Rundungsfehler beim Rechnen möglich, besser DECIMAL nehmen, wenn es um finanzmathematische Rechnungen geht (z.B. Buchhaltung).

* DECIMAL: Zahlen dieses Typs sind vor MY!5.0 in Wirklichkeit Strings mit je einem Zeichen pro Ziffer, für das Komma und für das Vorzeichen (STD: 10.0) und werden bei Rechnungen nach DOUBLE konvertiert (max. 15 Stellen genau).

  Ab MY!5.0 wird dafür "Precision Math" verwendet, d.h. eine kompaktere (9 Ziffern pro 4 Byte + Restziffern) und genauere Darstellung (max. 64 St.) benutzt und bei Rechnungen nicht mehr nach DOUBLE umgewandelt. Hat ähnliches Verhalten wie BCD-Format (Binary Coded Digits) für kaufmännisches Rechnen.

  Kaufmännische oder finanzmathematische Daten wie Preise, Umsatz, ... mit DECIMAL darstellen (vermeidet Rundungsfehler von FLOAT/DOUBLE) oder statt in Euro mit 2 Nkst besser in Cent ohne Nkst in INT speichern.

  ACHTUNG: Dezimalkomma in Zahl (z.B. 1,50) ist (meist) ein Syntaxfehler, Dezimalkomma in String (z.B. "1,50") schneidet Rest nach Komma ab, falls als Zahl interpretiert (ignoriert).

* Zahlen können keine führenden 0-en speichern
  + Falls das notwendig ist, CHAR/VARCHAR verwenden
    - Damit kann man trotzdem rechnen
    - Preis: Konvertierung String --> Zahl zur Ausführungszeit (langsam)!
  + Alternativ ZEROFILL (feste Gesamtlänge abhängig von Darstellungsbreite)

* BOOL: Kennt die Konstanten TRUE (Wahr, Wert 1) und FALSE (Falsch, Wert 0). Jeder Wert außer 0, FALSE (und NULL) ist TRUE (Wahr)!

* YEAR: Deckt folgende Bereiche ab (d.h. nur 255 Jahre darstellbar)
    YEAR(2): 1970-2069 (00-69 --> 20XX, 70- 99 --> 19XX)
    YEAR(4): 1900-2155 (00-99 --> 19XX, 100-255 --> 2XXX)
  ACHTUNG: Die Abbildungsregeln YY --> YYYY unterscheiden sich!

YEAR(2) wird seit MY!5.X.X automatisch auf YEAR(4) abgebildet (deprecated)

* DATE: Werte decken die Jahre 01.01.1000-31.12.9999 ab. Jeder Teil (Tag, Monat, Jahr) ist auf "00" setzbar, um auszudrücken, dass ein Datumteil unbekannt ist. Derartige Teile werden bei Sortierung als 1. Wert einsortiert. Ein insgesamt ungültiger Datumswert wird als "0000-00-00" gespeichert.

* Per SQL-Modus ist steuerbar, welche ungültigen Datumswerte akzeptiert werden:

sql_mode = "..."	Bedeutung
ALLOW_INVALID_DATES	Beliebige Komb. von Mon=1..12 + Tag=1..31 erlaubt
NO_ZERO_DATE	0000-00-00 verboten (aber YYYY-01-00 / YYYY-00-01)
NO_ZERO_IN_DATE	Jahr=00 oder Monat=00 oder Tag=00 nicht erlaubt

Beispiel ("_" und "-" in Option und Konfig-Datei möglich):

```
--sql_mode="ALLOW_INVALID_DATES,NO_ZERO_DATE"           # "mysqld"-Start
sql_mode = ""                                           # In "my.cnf"
SET GLOBAL sql_mode  = "ALLOW_INVALID_DATES";           # Server-Global
SET SESSION sql_mode = "NO_ZERO_DATE,NO_ZERO_IN_DATE";  # Sitzungsbezogen
SELECT @@GLOBAL.sql_mode;                               # Globaler Wert
SELECT @@SESSION.sql_mode;                              # Sitzungsbez. Wert
```

* TIME: Werte decken Zeitbereich -838:59:59.000000 bis 838:59:59.999999 ab
  + Ab MY!5.6.4 auch auf Mikrosekunden .nnnnnn genau: TIME(<Fsp>)
    (Fsp = Fractional seconds part = 0-6 Stellen als zusätzliche Angabe)

* DATETIME: Zusammenfassung von DATE und TIME. Abhängig vom SQL-Modus auch ungültige Datumswerte (z.B. "00.00.0000" oder Jahr/Tag/Monat="0") speicherbar.
  + Ab MY!5.6.4 auch auf Mikrosekunden .nnnnnn genau: DATETIME(<Fsp>)
    (Fsp = Fractional seconds part = 0-6 Stellen als zusätzliche Angabe)

* TIMESTAMP: Wert als Anzahl Sekunden seit 1.1.1970 00:00:00 gespeichert,
  + Zeitraum: 01.01.1970 00:00:01 ... 19.01.2038 03:14:07 (UTC)
  + UNIX-Zeitrechnung mit 32-Bit Zahl mit Vorzeichen --> Y2K-Problem!
  + Unabhängig vom SQL-Modus nur gültige Datumswerte möglich
    (kein "00.00.0000" oder Tag/Monat="0").
  + Ab MY!5.6.4 auch auf Mikrosekunden .nnnnnn genau: TIMESTAMP(<Fsp>)
    (Fsp = Fractional seconds part = 0-6 Stellen als zusätzliche Angabe)

  Eine TIMESTAMP-Angabe hat die Form YYYYMMDDhhmmss.nnnnnn (Zahl oder "String") oder "YYYY-MM-DD hh:mm:ss.nnnnnn" ("String"), eine TIMESTAMP-Ausgabe hat je nach MySQL-Version unterschiedliche Form:

```
YYYYMMDDhhmmss.nnnnnn           # Alt (vor MY!5.4)
"YYYY-MM-DD hh:mm:ss.nnnnnn"    # Neu (seit MY!5.4)
```

  ACHTUNG: Der Wert der 1. TIMESTAMP-Spalte in einer Tabelle wird bei JEDER Änderung eines Datensatzes auf aktuelle(s) Datum + Uhrzeit CURRENT_TIMESTAMP() gesetzt (weitere TIMESTAMP-Spalten bleiben unverändert). Soll diese Spalte bei Änderungen des Datensatzes unverändert bleiben, ist ihr explizit der aktuelle Wert zuzuweisen:

```
UPDATE ... SET ..., ts = ts;
```

  ACHTUNG: Ohne Angabe des Attributs NULL/NOT NULL erlauben alle Datentypen den Wert NULL außer dem Datentyp TIMESTAMP. Hier ist explizit das Attribut NULL anzugeben, wenn der NULL-Wert erlaubt sein soll.

  Lässt man Teile am Ende der Zeitangabe YYYYMMDDhhmmss / "YYYY-MM-DD hh:mm:ss" weg, dann wird der KLEINSTE mögliche Wert dafür eingesetzt. Die Bedingungen (a) bis (f) lassen also am rechten Rand des Zeitraums einiges weg, was man zunächst so nicht erwartet. Erst Bedingung (7) umfasst den beabsichtigten

```
  Zeitraum vollständig:
                                                  # 1.1.12 00:00:00 ..
    ts BETWEEN 2012           AND 2013            # ..  1.1.13 00:00:00 (a)
    ts BETWEEN 201201         AND 201301          # ..  1.1.13 00:00:00 (b)
    ts BETWEEN 20120101       AND 20130131        # .. 31.1.13 00:00:00 (c)
    ts BETWEEN 20120101000000 AND 20130131000000  # .. 31.1.13 00:00:00 (d)
    ts BETWEEN 2012010100     AND 2013013123      # .. 31.1.13 23:00:00 (e)
    ts BETWEEN 201201010000   AND 201301312359    # .. 31.1.13 23:59:00 (f)
    ts BETWEEN 20120101000000 AND 20130131235959  # .. 31.1.13 23:59:59 (g)

  HINWEIS: Alle Elemente in Zeitangaben der Form YYYYMMDDhhmmss / "YYYY-MM-DD
  hh:mm:ss" sind mit 4 (Jahr) bzw. 2 Ziffern (sonst) anzugeben. Es ist
  unzulässig, die führende Null bei einstelligem Tag, Monat, Stunde, Minute
  und Sekunde wegzulassen.

* CHAR(n) speichert n Zeichen mit Leerzeichen aufgefüllt (SPACE-padded),
          beim Lesen werden die Leerzeichen ENTFERNT (stripped).
  BINARY(n) speichert n Byte mit NUL-Bytes aufgefüllt (NUL-padded),
          beim Lesen werden die NUL-Bytes NICHT ENTFERNT (not stripped).

* VARCHAR(n) speichert 0-n Zeichen PLUS eine Länge (1/2 Byte).
  VARBINARY(n) speichert 0-n Byte PLUS eine Länge (1/2 Byte).

* ACHTUNG: CHAR, VARCHAR und TEXT ignorieren GROSS/kleinschreibung beim
  Vergleichen und Sortieren (case-INsensitive).
  BINARY, VARBINARY und BLOB berücksichtigen GROSS/kleinschreibung beim
  Vergleichen und Sortieren (case-sensitive).

* TEXT/BLOB: Ein BLOB ist ein "Binary Large OBject" der Länge 0-4 Mrd Byte.
  Nicht im Datensatz gespeichert, dort steht nur "Zeiger" der Größe L Byte.
  Die Daten selber stehen pro Objekt an einer anderen Stelle.
  + TEXT: Textdaten, Vergleich und Sortierung erfolgt anhand Zeichensatz und
          Collation (ohne Berücksichtigung der GROSS/kleinschreibung).
  + BLOB: Binärdaten, Vergleich und Sortierung erfolgt anhand der Byte-Codes.

  Entsprechen VARCHAR/VARBINARY mit folgenden Unterschieden:
  + TEXT/BLOB-Spalten in Indices MÜSSEN eine Präfix-Längenangabe haben.
  + TEXT/BLOB-Spalten können KEINE Defaultwerte haben.
  + Speicherung erfolgt nicht im Datensatz sondern außerhalb,
    im Datensatz steht nur ein Zeiger.

* ENUM: Wie Datentyp (VAR)CHAR verwendbar, der String wird aber als ZAHL
  gespeichert. Als Werte sind nur Elemente einer vordefinierten Werteliste
  erlaubt, anstelle des Elements wird platzsparend seine Nummer gespeichert.
  Ein String, der nicht in der Werteliste enthalten ist, entspricht NULL.
  Auch Index 1..N gemäß Werteliste speicherbar (1="Elem1", 2="Elem2", ...).

* SET: Wie Datentyp (VAR)CHAR verwendbar, eine Kombination von Strings wird
  als BITMUSTER gespeichert. Als Wert ist eine beliebige Kombination ("Menge")
  der Werte aus einer vordefinierten Werteliste (max. 64 Werte) erlaubt (auch
  die leere Menge ""). Anstelle der Element-Kombination wird platzsparend ein
  Bitmuster gespeichert, jede Bitposition entspricht einem Element der Menge.
  + Bit 1 an Bitposition N heißt, das N-te Elem. befindet sich in der Menge,
  + Bit 0 an Bitposition N heißt, das N-te Elem. befindet sich NICHT in der M.

* BIT: Kann eine Menge von 1..64 Bit (1-8 Byte) aufnehmen. Jedes Bit ist
  entweder gesetzt (Wert 1) oder nicht gesetzt (Wert 0).


4a) Datentyp-Optimierung (Performance/Speicherplatz)
----------------------------------------------------

* Kleinstmöglichen Datentyp nutzen (z.B. MEDIUMINT statt INT)
  --> Weniger Festplattenplatz belegt --> Weniger Platten-I/O --> Schneller!
  ABER: Zukunftssicher auslegen (Y2K-Problem, IPv4 --> IPv6,
        UNIX-Timestamp endet am 19.01.2038)!
```

* Spalten möglichst "NOT NULL" deklarieren!
  --> Weniger Platzverbrauch (NULL benötigt zusätzl. Byte pro Spalte)
  --> Index schneller!

* Mind. eine VARCHAR/VARBINARY/TEXT/BLOB-Spalte --> VARIABLE Record-Länge
  Nur CHAR/BINARY-Spalten   --> FIXE   Record-Länge
  --> 4b) Fixe/Variable Record-Länge (Row-Format)

* In SELECT keine Spalten-Konvertierung in der WHERE-Bedingung durchführen
  --> Index sonst nicht nutzbar
  --> Für jeden Satz notwendig --> langsam!
  ACHTUNG: Findet implizit statt, falls mit Textspalte GERECHNET wird!

* In JOIN verwendete Spalten sollten exakt gleichen Datentyp + Länge haben,
  --> Index nutzbar
  --> Bei (VAR)CHAR/(VAR)BINARY/TEXT/BLOB unterschiedliche Länge toleriert

* MySQL-Optimierer macht selbständig folgende Änderungen bei CREATE/ALTER TABLE
  (Speicherplatz minimal, Tabellenstruktur optimiert, autom. Konvertierung)

Datentyp	Automatische Änderung
VARCHAR(N<=4)	CHAR(N)
VARBINARY(N<=4)	BINARY(N)
CHAR(N>4)	VARCHAR(N) (falls mind. eine VARCHAR-Spalte vorhanden)
TIMESTAMP(N)	TIMESTAMP (2 <= N geradzahlig <= 14)
PRIMARY KEY	Spalten erhalten Attribut NOT NULL
YEAR(2)	YEAR(4)

4b) Fixe/Variable Record-Länge (Row-Format)
----

* Datensätze (Rows) können max. 65535 Byte lang sein
  (abhängig von Engine auch weniger) --> 26) Begrenzungen von MySQL (Limits)

* FIXES Rowformat (feste Länge): ALLE Spalten NICHT von folgenden 4 Datentypen.
  VARIABLES Rowformat (unterschiedl. Länge): Mind. EINE Spalte dieses Datentyps:

    VARCHAR   > 4 Zeichen
    VARBINARY > 4 Zeichen
    ...BLOB
    ...TEXT

* FIXES Rowformat hat Vorteile bei der Adressierung der Datensätze,
  nutzt aber evtl. Plattenplatz schlechter (wg. Füllbyte)

* VARIABLES Rowformat nutzt Plattenplatz besser aus,
  benötigt aber weitere Bytes (<=255 --> 1, >255 --> 2) zur Längeninformation

* Bei vielen Lösch/Einfüge-Operationen in Tabelle führt variables
  Rowformat leichter zur "Fragmentierung" (Zersplitterung) der Tabellendaten
  --> OPTIMIZE TABLE durchführen

* Für Table Scan und Zugriff per Index ist fixes/variables Rowformat egal!

* Besondere Einstellungen bei Tabellen-Definition
  + MyISAM: ROW_FORMAT=FIXED/DYNAMIC (myisampack + myisamchk -rq --> COMPRESSED)
  + InnoDB: ROW_FORMAT=REDUNDANT/COMPACT (20% weniger Platz, CPU erhöht)

* TIP: Breite Tabelle in mehrere Teile aufsplitten (fixer + variabler Anteil)

4c) AUTO_INCREMENT

* Jedem Datensatz in Tabelle automatisch eindeutigen Key zuordnen
  + "Künstlicher" Schlüssel (von Datenbank generiert)
  + Eindeutige positive Zahl (negativ --> Große positive Zahl)
    - Startwert beim Anlegen der Tabelle wählbar (STD: 1):
      CREATE TABLE ... (...) ... AUTO_INCREMENT = <N> ...
      ALTER TABLE <Tbl> ... AUTO_INCREMENT = <N>;
    - Alternativ Zeile mit Wert N-1 einfügen und wieder löschen

* Nur in EINER Spalte pro Tabelle erlaubt
  + Spalten-Datentyp muss INT, FLOAT oder DOUBLE sein
    (DECIMAL nicht möglich, UNSIGNED schon)
  + Index MUSS darauf liegen (nicht unbedingt UNIQUE, aber sinnvoll;
                              PRIMARY KEY am sinnvollsten)
  + Keine DEFAULT-Angabe möglich

* Einfügen von NULL/0 in diese Spalte setzt nächsten noch nicht benutzten Wert
  + SQL-Modus "NO_AUTO_VALUE_ON_ZERO": 0 als Wert möglich (nicht empfohlen!)
  + Spaltenwert angegeben
    - Schon vorhanden   --> Einfügen wird verweigert
    - Noch nicht vorhanden --> Eingefügt + nächster automatisch ermittelter
                               um 1 höher (Lücken entstehen!)

* Letzten verwendeten AUTO_INCREMENT-Wert direkt nach INSERT abfragen
  (nur in derselben Sitzung/Transaktion):

```
SELECT LAST_INSERT_ID();
```

* HINWEIS: Damit Master-Master-Replikation möglich ist (2 gegenseitige Master
  oder Ring aus mehreren Mastern), müssen AUTO_INCREMENT-Spalten auf allen
  Mastern garantiert unterschiedliche Werte erzeugen. Dazu gibt es im
  MySQL-Server folgende Optionen:

```
auto_increment_offset    = <N>   # Startwert von AUTO_INCREMENT-Spalten
auto_increment_increment = <N>   # Schrittweite von AUTO_INCREMENT-Spalten
```

  Ist die Anzahl der sich gegenseitig replizierenden Master z.B. 5, dann
  sollten die einzelnen Master verschiedene Offsets 1, 2, 3, 4, 5 erhalten
  und bei allen Mastern die Schrittweite auf 5 gesetzt werden.

* Pro Tabelle ist ein eigener Startwert für AUTO_INCREMENT möglich,
  aber keine Schrittweite (diese ist nur pro Server einstellbar).

* HINWEISE:
  + In PostgreSQL dafür Datentyp "SERIAL" verwenden
  + In Oracle durch "Sequence"-Objekt nachbilden (deutlich komplexer)

Beispiel:

```
CREATE TABLE pers (
  nr      INT NOT NULL AUTO_INCREMENT,      # Spalte "nr" automatisch füllen
  vorname VARCHAR(30),                      #
  name    VARCHAR(30),                      #
  PRIMARY KEY (nr)                          # Wg. AUTO_INCREMENT nötig
) AUTO_INCREMENT = 10;                      # Werte 1..9 überspringen
                                            #
INSERT INTO pers (nr, vorname, name)        #
  VALUES (NULL, "Thomas",  "Birnthaler"),   # --> nr=10 (Lücke von 1..9)
         (20,   "Markus",  "Mueller");      # --> nr=20 (Lücke von 11..19)
                                            #
INSERT INTO pers (vorname, name)            #
  VALUES ("Andrea",  "Bayer"),              # --> nr=21
         ("Richard", "Seiler"),             # --> nr=22
         ("Heinz",   "Bayer");              # --> nr=23
```

```
                                          #
  INSERT INTO pers (nr, vorname, name)    #
    VALUES (20, "Hans",  "Birnthaler");   # --> verweigert
```

5) MySQL-Operatoren

Folgende Operatoren sind als Verknüpfung in Ausdrücken (Expressions) verwendbar:

Typ	Beispiele	Bemerkung
Arithmetik	+ - * / DIV % MOD	MOD/% = Modulo (Div.rest)
Vergleich	< <= > >=	Nicht => und <=!
	= != <> <=>	<=> ist NULL-sicheres =
	<Val> BETWEEN <Left> AND <Right>	Ränder einschließen
	<Val> NOT BETWEEN <Left> AND <R.>	Ränder nicht einschl.
	<Val> IN (<Val1>, <Val2>, ...)	In Liste enthalten
	<Val> NOT IN (<Val1>, <Val2>, ...)	Nicht in Liste enthalten
	<Val> LIKE <Muster> [ESCAPE <Char>]	Wildcards "_" "%" (s.u.)
	<Val> NOT LIKE <Muster>	Wildcards "_" "%" (s.u.)
	<Val> REGEXP/RLIKE <RegEx>	Regulärer Ausdruck (s.u.)
	<Val> NOT REGEXP/RLIKE <RegEx>	Regulärer Ausdruck (s.u.)
	<Val1> SOUNDS LIKE <Val2>	SOUNDEX(v1) = SOUNDEX(v2)
	<Val> IS NULL/UNKNOWN	Gleich NULL/Unbekannt
	<Val> IS NOT NULL/UNKNOWN	Ungleich NULL/Unbekannt
	<Val> IS TRUE/FALSE	Gleich TRUE/FALSE
	<Val> IS NOT TRUE/FALSE	Ungleich TRUE/FALSE
Logisch	NOT !	Erg: Nur TRUE/FALSE
	OR ||	Erg: Nur TRUE/FALSE
	AND &&	Erg: Nur TRUE/FALSE
	XOR	Erg: Nur TRUE/FALSE
Bit	| & ^ ~ << >>	OR AND XOR INV L/R-SHIFT
Zeichen	BINARY ...	String binär interpret.
	_CHARSET	String-Zeichensatz
	COLLATE	String-Sortierordnung

Beispiele (Rechnen):

```
  SELECT 1 + 2 * 3 / 4 - 5;                                  # --> -2.5
  SELECT 17 / 3, 17 DIV 3, 17 % 3, 17 MOD 3;                 # --> 5.6667 5 2 2
```

Beispiele (Vergleich):

```
  SELECT 10 < 4, 2 <= 5, 1 = 3, 1 <> 3;                      # --> 0 1 0 1
  SELECT 10 > 4, 2 >= 5, 1 <=> 3, 1 != 3;                    # --> 1 0 0 1
  SELECT 0 <=> 0, 0 <=> NULL, NULL <=> 0, NULL <=> NULL;     # --> 1 0 0 1
  SELECT 10 BETWEEN 0 AND 12, 10 NOT BETWEEN 0 AND 12;       # --> 1 0
  SELECT 10 IN (1, 2, 10), 10 NOT IN (1, 2, 10);             # --> 1 0
  SELECT "abc" LIKE "%b%", "abc" NOT LIKE "%b%";             # --> 1 0
  SELECT "abc" RLIKE "b", "abc" NOT RLIKE "b";               # --> 1 0
  SELECT "abc" SOUNDS LIKE "ebz";                            # --> 0
  SELECT NULL IS NULL, NULL IS NOT NULL;                     # --> 1 0
  SELECT TRUE IS TRUE, TRUE IS FALSE;                        # --> 1 0
  SELECT TRUE IS NOT TRUE, TRUE IS NOT FALSE;                # --> 0 1
```

Beispiele (Logische Verknüpfung):

```
  SELECT TRUE AND FALSE OR FALSE AND NOT TRUE;               # --> 0
  SELECT TRUE && FALSE  || FALSE &&   !  TRUE;               # --> 0
```

```
  SELECT TRUE XOR FALSE, TRUE XOR TRUE, FALSE XOR TRUE;     # --> 1 0 1

Beispiele (Bitweise Verknüpfung):

  SELECT BIN(b'11110001' | b'00111100');                    # --> 11111101 (253)
  SELECT BIN(b'11110001' & b'00111100');                    # --> 00110000  (48)
  SELECT BIN(b'11110001' ^ b'00111100');                    # --> 11001101 (205)
  SELECT BIN(~ b'00110101');                                # --> 11..1001010
  SELECT BIN(~ b'00110101' & b'11111111');                  # --> 11001010 (202)
  SELECT BIN(b'00110101' >> 3);                             # --> 00000110   (6)
  SELECT BIN(b'00110101' << 2);                             # --> 11010100 (212)
  SELECT BIN(~(b'00110101' & b'00001111'));                 # --> 11..111010
  SELECT BIN(~(b'00110101' | b'00001111') ^ b'00110011');   # --> 11..110011

Hinweise:
  * Division "/" ist immer Fließkommadivision,
    Division "DIV" schneidet den Divisionsrest ab.
  * Standardmäßig gibt es keinen Stringverkettungs-Operator (z.B. &  +  .  ||)
    --> CONCAT verwenden, damit sind beliebig viele Strings verkettbar
        (SET GLOBAL sql_mode = 'PIPES_AS_CONCAT' oder
         SET GLOBAL sql_mode = 'ANSI' --> Operator "||" verkettet Strings)
      CONCAT("abc", <Col>, "xyz")   # Immer möglich
      "abc" || <Col> || "xyz"       # Falls sql_mode='PIPES_AS_CONCAT/ANSI'
  * Falls in Stringverkettung NULL vorkommen kann --> Auf "" abbilden:
      CONCAT("abc", IF(ISNULL(<Col>), "", <Col>), "xyz")
      CONCAT("abc", IFNULL(<Col>, ""), "xyz")
  * BINARY vor String in Vergleichen = != <> <=> < <= > >= BETWEEN...AND IN LIKE
    REGEX und in ORDER BY vor Spalte erzwingt Berück. der GROSS/kleinschreibung:
      SELECT * FROM pers WHERE BINARY name = "abc";
      SELECT * FROM pers WHERE BINARY name != "abc";
      SELECT * FROM pers WHERE BINARY name > "abc";
      SELECT * FROM pers WHERE BINARY name <= "abc";
      SELECT * FROM pers WHERE BINARY name BETWEEN "abc" AND "def";
      SELECT * FROM pers WHERE BINARY name LIKE "abc%";
      SELECT * FROM pers WHERE BINARY name REGEX "^abc$";
      SELECT * FROM pers ORDER BY BINARY name ASC;
  * LIKE-Wildcards für beliebige/beliebig viele Zeichen (<Muster>):
      "_" = GENAU EIN beliebiges Zeichen
      "%" = Beliebig viele beliebige Zeichen (auch 0!)
  * ESCAPE <Char> am Ende von LIKE setzt Entwertungs-Zeichen von "%" und "_"
    auf das Zeichen <Char> (STD: "\"). D.h. um mit LIKE nach den Zeichen "%"
    und "_" selbst zu suchen, muss man dieses Escape-Zeichen davor setzen:
      SELECT * FROM test WHERE name LIKE "\_\%";              # Standard "\"
      SELECT * FROM test WHERE name LIKE "@_@%" ESCAPE "@";   # Escape = "@"

Hinweise zu NULL:
  * NULL kommt in Ausdruck vor  --> Ergebnis NULL ("schwarzes Loch")
  * NULL kommt in Vergleich vor --> Ergebnis NULL (außer bei <=>)
  * Logische Ausdrücke verkürzt ausgewertet (Short-Cut/Circuit Evaluation):
    + FALSE AND ... --> sofort FALSE (ohne Auswertung von ...)
    + TRUE OR ...   --> sofort TRUE  (ohne Auswertung von ...)
    D.h. hat ... Wert NULL, ist das Gesamtergebnis trotzdem FALSE bzw. TRUE!
  * Ist NULL das Ergebnis einer WHERE-Bedingung, entspricht dies dem Wert FALSE.
  * IS UNKNOWN analog IS NULL (andere Schreibweise, bei Bool. Werten üblich):
      SELECT 0 IS UNKNOWN, 1 IS UNKNOWN, NULL IS UNKNOWN;   # --> 0 0 1
      SELECT 0 IS NULL,    1 IS NULL,    NULL IS NULL;      # --> 0 0 1
  * "<=>" ist NULL-sicherer Vergleich auf gleichen Wert (Ersatz für "="):
      SELECT NULL <=> NULL;         # --> 1  (bei "=" --> NULL)
      SELECT NULL <=> TRUE;         # --> 0  (bei "=" --> NULL)
      SELECT NULL <=> FALSE;        # --> 0  (bei "=" --> NULL)
      SELECT NULL <=> "abc";        # --> 0  (bei "=" --> NULL)
      SELECT NULL <=> "";           # --> 0  (bei "=" --> NULL)
      SELECT NULL <=> "NULL";       # --> 0  (bei "=" --> NULL)
    Verhalten identisch zu "=" falls NULL nicht als Verknüpfungswert vorkommt.

Rangfolge/Priorität der Operatoren (durch Klammerung "(...)" ändern):
```

Nr	Typ	Bemerkung
1	BINARY	GROSS/kleinschreibung beachten
	_CHARSET COLLATE	Zeichensatz + Sortierung anpassen
2	! (NOT)	Negation (falls HIGH_NOT_PRECEDENCE)
3	- ~	Unäres Minus, Bitweise Invertierung
4	||	Stringverkettung (falls sql_mode='PIPES_AS_CONCAT/ANSI')
5	^	Bitweise XOR
6	* / DIV % MOD	MOD/% = Modulo, DIV = ganzz. Div.
7	+ -	Addition/Subtraktion
9	<< >>	Bitweise Shift links/rechts
9	&	Bitweise AND
10	|	Bitweise OR
11	= != <> <=> < <= > >=	Vergleich
	IN IS LIKE REGEXP	Vergleich
12	BETWEEN ... AND ...	Bereich
	CASE ... WHEN ... THEN ... ELSE	Fallunterscheidung
13	NOT	Logische Negation
		(ab MY!5.0.2 vom Vorrang her hier!)
14	AND &&	Logisch AND
15	XOR	Logisch XOR
16	OR ||	Logisch OR (falls sql_mode="...")
17	:=	Zuweisung

HINWEIS: Der Operator NOT hatte vor MY!5.0.2 die gleiche (hohe) Priorität wie der Operator !, seither hat er eine Priorität zwischen BETWEEN und AND. Die alte (hohe) Priorität erhält man mit folgendem SQL-Modus:

```
SET GLOBAL sql_mode = 'HIGH_NOT_PRECENDENCE';
```

HINWEIS: Falls der OR-Operator "||" Strings verketten soll (ANSI-SQL Standard):

```
SET GLOBAL sql_mode = 'PIPES_AS_CONCAT';   # Alternative 1
SET GLOBAL sql_mode = 'ANSI';              # Alternative 2
```

6) Boolesche Logik

In Boolescher Logik gibt es nur die beiden Konstanten TRUE und FALSE mit dem numerischen Wert 1 und 0. Umgekehrt werden alle numerischen Werte ungleich 0 auf TRUE abgebildet und der numerische Wert 0 auf FALSE. Strings im Booleschen Kontext werden in Zahlen umgewandelt, d.h. alle Strings mit Ergebnis "0" sind FALSE, die anderen sind TRUE:

Logische Ausdrücke ergeben 1 für TRUE und 0 für FALSE

```
SELECT TRUE, FALSE, NOT TRUE, NOT FALSE;              # --> 1 0 0 1
SELECT NOT 0, NOT 1, NOT 123, NOT -456, NOT 7.89;     # --> 1 0 0 0 0
SELECT NOT "", NOT "abc", NOT "123def", NOT "0def";   # --> 1 1 0 1
```

Rechnen mit TRUE/FALSE entspricht Rechnen mit 0/1:

```
SELECT TRUE / FALSE;   # --> NULL (Division durch 0!)
SELECT FALSE / TRUE;   # --> 0.0000
SELECT TRUE * FALSE;   # --> 0
SELECT TRUE + FALSE;   # --> 1
SELECT TRUE - FALSE;   # --> 1
SELECT FALSE - TRUE;   # --> -1
```

Das Ergebnis eines Booleschen Ausdrucks kann bereits bei Betrachtung eines TEILS der damit verknüpften Ausdrücke feststehen, egal welchen Wert die

restlichen verknüpften Ausdrücke haben (Short-Cut/Short-Circuit Evaluation = Verkürzte Auswertung):

```
  <Expr1> AND <Expr2>   # <Expr2> nur ausgewertet, falls <Expr1> TRUE ergibt
  <Expr1> OR  <Expr2>   # <Expr2> nur ausgewertet, falls <Expr1> FALSE ergibt
```

Beispiel (SQRT(-1) ist nicht definiert):

```
  SELECT 0 AND SQRT(-1);      # --> 0    (obwohl SQRT(-1) undefiniert)
  SELECT 1 AND SQRT(-1);      # --> NULL (da     SQRT(-1) undefiniert)
  SELECT 1 OR SQRT(-1);       # --> 1    (obwohl SQRT(-1) undefiniert)
  SELECT 0 OR SQRT(-1);       # --> NULL (da     SQRT(-1) undefiniert)
```

7) Der Wert NULL

7a) Eigenschaften von NULL

* NULL steht für Wert "unbekannt" oder "undefiniert"
* NULL ist weder die Zahl 0 noch der leere String "",
  noch TRUE oder FALSE, sondern etwas völlig anderes.
* Fast alle Operationen mit NULL ergeben NULL ("Schwarzes Loch")
  + Ausnahmen: <=>, AND, OR, IS NULL, IS NOT NULL,
               ISNULL(), IFNULL(), NULLIF(), COALESCE()
               SUM(), AVG(), COUNT(), ...
* Benötigt extra Speicherplatz in Tabelle (pro NULL-Spalte 1 Byte oder 1 Bit)
  --> Spalten möglichst mit NOT NULL DEFAULT <Wert> kennzeichnen (STD: NULL)
* NULL-sicheren Vergleich mit <=> statt = durchfuehren (MY!)

```
  SELECT 1 <=> NULL, NULL <=> NULL, 0 <=> NULL, "" <=> NULL;   # --> 0 1 0 0
  SELECT 1 = NULL, NULL = NULL, 0 = NULL, "" = NULL;           # --> 4x NULL
  SELECT NULL IS NULL, 0 IS NULL, 1 IS NULL, "" IS NULL;       # --> 1 0 0 0
  SELECT NULL IS NOT NULL, 0 IS NOT NULL, 1 IS NOT NULL,       #
         "" IS NOT NULL;                                       # --> 0 1 1 1
  SELECT ISNULL(NULL), ISNULL(0), ISNULL(1), ISNULL("");       # --> 1 0 0 0
```

Kommt Wert NULL in einem Ausdruck vor, so hat in der Regel der GESAMTE Ausdruck als Ergebnis den Wert NULL:

```
  SELECT 3 * 4 - 1 + NULL AS "Wert";                           # --> NULL
  SELECT 1 = NULL, 1 != NULL, 1 < NULL, 1 > NULL;              # --> 4x NULL
  SELECT NULL = 1, NULL = NULL, 0 = NULL;                      # --> 3x NULL
  SELECT NULL != 1, NULL != NULL, 0 != NULL;                   # --> 3x NULL
  SELECT CONCAT(NULL, "abc", "def", NULL, "123");              # --> NULL
  SELECT CONCAT("abc", "def", NULL, "123");                    # --> NULL
```

7b) Prüfung auf Wert NULL

Zur expliziten Prüfung auf den Wert NULL gibt es folgende Möglichkeiten:

Ausdruck	Bedeutung
<Col> IS NULL	In WHERE-Bedingung
<Col> IS NOT NULL	In WHERE-Bedingung
<Expr> IS NULL	In Ausdruck
<Expr> IS NOT NULL	In Ausdruck
ISNULL(<Expr>)	1 wenn <Expr> = NULL, sonst 0
IFNULL(<Expr1>, <Expr2>)	<Expr1> wenn nicht NULL, sonst <Expr2>
IF(ISNULL(<E1>), <E2>, <E1>))	(analog)
NULLIF(<Expr1>, <Expr2>)	NULL wenn <Expr1>=<Expr2>, sonst <Expr1>
COALESCE(<Expr1>, <Expr2>, ...)	Erster Ausdruck <Expr_i> != NULL

```
  +-------------------------------+----------------------------------------+
```

Beispiel:

```
  SELECT NULL IS NULL, NULL IS NOT NULL;                  # --> 1 0
  SELECT 0 IS NULL, 0 IS NOT NULL;                        # --> 0 1
  SELECT 1 IS NULL, 1 IS NOT NULL;                        # --> 0 1
  SELECT "" IS NULL, "" IS NOT NULL;                      # --> 0 1
  SELECT ISNULL(NULL), ISNULL(0), ISNULL(1), ISNULL("");  # --> 1 0 0 0
  SELECT IFNULL("eins", NULL), IFNULL(NULL, "zwei");      # --> eins zwei
  SELECT NULLIF("eins", "eins"), NULLIF("eins", "zwei");  # --> NULL eins
  SELECT COALESCE(NULL, NULL, "abc", NULL, 123);          # --> abc
```

7c) Vergleiche mit NULL

Vergleiche ergeben einen Booleschen Wert außer mit NULL wird verglichen,
dann resultiert der Wert NULL als Ergebnis (außer bei "<=>").

Der normale Vergleich "=" liefert daher:

```
  +-------+-------+-------+-------+
  |   =   | TRUE  | FALSE | NULL  |   # Ein NULL bei "=" ergibt insgesamt NULL
  +-------+-------+-------+-------+
  | TRUE  | TRUE  | FALSE | NULL  |
  | FALSE | FALSE | TRUE  | NULL  |
  | NULL  | NULL  | NULL  | NULL  |
  +-------+-------+-------+-------+
```

Ebenso gilt für den Vergleich "!=" (oder "<>"):

```
  +-------+-------+-------+-------+
  |  !=   | TRUE  | FALSE | NULL  |   # Ein NULL bei "!=" ergibt insgesamt NULL
  +-------+-------+-------+-------+
  | TRUE  | FALSE | TRUE  | NULL  |
  | FALSE | TRUE  | FALSE | NULL  |
  | NULL  | NULL  | NULL  | NULL  |
  +-------+-------+-------+-------+
```

Die anderen Vergleiche <Cmp> ("<", "<=", ">" und ">=") ergeben:

```
  +-------+------------+-------+
  | <Cmp> | !NULL      | NULL  |   # Ein NULL bei <Cmp> ergibt insgesamt NULL
  +-------+------------+-------+
  | !NULL | TRUE/FALSE | NULL  |
  | NULL  | NULL       | NULL  |
  +-------+------------+-------+
```

Der NULL-sichere Vergleich "<=>" liefert hingegen (MY!):

```
  +-------+-------+-------+-------+
  |  <=>  | TRUE  | FALSE | NULL  |
  +-------+-------+-------+-------+
  | TRUE  | TRUE  | FALSE | FALSE |   # NULL <=> TRUE  ergibt FALSE
  | FALSE | FALSE | TRUE  | FALSE |   # NULL <=> FALSE ergibt FALSE
  | NULL  | FALSE | FALSE | TRUE  |   # NULL <=> NULL  ergibt TRUE
  +-------+-------+-------+-------+
```

7d) Boolesche Logik mit NULL (ternär/dreiwertig)

Die Booleschen Operatoren AND, &&, OR, ||, XOR, NOT und ! werden auf die
Verknüpfung mit NULL-Werten erweitert (ternäre oder dreiwertige Logik).
Aufgrund der "verkürzten Auswertung" von AND und OR gilt:

  * FALSE links von AND ergibt sofort FALSE (egal was rechts davon steht)
  * TRUE links von OR ergibt sofort TRUE (egal was rechts davon steht)
  * FALSE in AND-Verknüpfung ergibt FALSE   (egal ob NULL-Werte verknüpft)
  * TRUE in OR-Verknüpfung ergibt TRUE   (egal ob NULL-Werte verknüpft)
  * NULL in XOR-Verknüpfung ergibt NULL   (egal was sonst verknüpft)

Hier die dreiwertigen Auswertungstabellen der Booleschen Operatoren:

```
  +-------+-------+-------+-------+
  | AND &&| TRUE  | FALSE | NULL  |  # Ein FALSE bei AND ergibt insgesamt FALSE!
  +-------+-------+-------+-------+
  | TRUE  | TRUE  | FALSE | NULL  |
  | FALSE | FALSE | FALSE | FALSE |
  | NULL  | NULL  | FALSE | NULL  |
  +-------+-------+-------+-------+

  +-------+-------+-------+-------+
  | OR || | TRUE  | FALSE | NULL  |  # Ein TRUE bei OR ergibt insgesamt TRUE!
  +-------+-------+-------+-------+
  | TRUE  | TRUE  | TRUE  | TRUE  |
  | FALSE | TRUE  | FALSE | NULL  |
  | NULL  | TRUE  | NULL  | NULL  |
  +-------+-------+-------+-------+

  +-------+-------+-------+-------+
  |  XOR  | TRUE  | FALSE | NULL  |  # Ein NULL bei XOR ergibt insgesamt NULL!
  +-------+-------+-------+-------+
  | TRUE  | FALSE | TRUE  | NULL  |
  | FALSE | TRUE  | FALSE | NULL  |
  | NULL  | NULL  | NULL  | NULL  |
  +-------+-------+-------+-------+

  +-------+-------+
  |       | NOT ! |  # NULL negiert ergibt NULL!
  +-------+-------+
  | TRUE  | FALSE |
  | FALSE | TRUE  |
  | NULL  | NULL  |
  +-------+-------+
```

Beispiele:

```
  SELECT TRUE  AND NULL, NULL AND TRUE;    # --> NULL NULL
  SELECT FALSE AND NULL, NULL AND FALSE;   # --> 0 0
  SELECT TRUE  &&  NULL, NULL &&  TRUE;    # --> NULL NULL
  SELECT FALSE &&  NULL, NULL &&  FALSE;   # --> 0 0
  SELECT TRUE  OR  NULL, NULL OR  TRUE;    # --> 1 1
  SELECT FALSE OR  NULL, NULL OR  FALSE;   # --> NULL NULL
  SELECT TRUE  ||  NULL, NULL ||  TRUE;    # --> 1 1
  SELECT FALSE ||  NULL, NULL ||  FALSE;   # --> NULL NULL
  SELECT TRUE  XOR NULL, NULL XOR TRUE;    # --> NULL NULL
  SELECT FALSE XOR NULL, NULL XOR FALSE;   # --> NULL NULL
  SELECT NOT NULL, ! NULL;                 # --> NULL NULL
```

8) Reguläre Ausdrücke in MySQL (RegEx)

Eigenschaften:
  * Reguläre Ausdrücke nennt man auch "RegEx" oder "Pattern" (Muster)
    (von englisch "Regular Expressions")
  * Vergleichen wie LIKE/NOT LIKE Texte mit einem Muster --> TRUE/FALSE
    (stark erweiterte Fähigkeiten gegenüber LIKE/NOT LIKE)
  * Kennen wesentlich mehr "Metazeichen" als LIKE (kennt nur "%" und "_"),
    um ein Muster auszudrücken, das mit dem Text verglichen wird

```
    + Wildcard "_" <=> RegEx "."
    + Wildcard "%" <=> RegEx ".*"
  * Operatoren: <Expr> REGEX <RegEx>
                <Expr> RLIKE <RegEx>
                <Expr> NOT REGEX <RegEx>
                <Expr> NOT RLIKE <RegEx>
  * Vergleiche mit Regulären Ausdrücken können KEINEN Index nutzen!
  * RegEx Muss IRGENDWO in Zeichenkette passen, damit TRUE
    (außer bei Verwendung von ^ bzw. $, dann am Anfang bzw. Ende)
```

Metazeichen	Bedeutung
X	Zeichen X steht für sich selbst (sofern kein Metazeichen)
.	1 beliebiges Zeichen
[...]	1 Zeichen aus Zeichenmenge ... (z.B. [A-Z])
[^...]	1 Zeichen NICHT aus Zeichenmenge ... (z.B. [^0-9])
[[:CLASS:]]	Zeichenklasse (siehe unten, [[und]] sind doppelt!)
...*	0-N mal Zeichen/geklammerter Ausdruck ... davor (= {0,})
...+	1-N mal Zeichen/geklammerter Ausdruck ... davor (= {1,})
...?	0,1 mal Zeichen/geklammerter Ausdruck ... davor (= {0,1})
...{N}	N mal Zeichen/geklammerter Ausdruck ... davor
...{N,M}	N-M mal Zeichen/geklammerter Ausdruck ... davor
^	Stringanfang
$	Stringende
[[:<:]]	Wortanfang (Wort = Folge von alnum + "_")
[[:>:]]	Wortende (Wort = Folge von alnum + "_")
(...)	Ausdruck ... gruppieren (für | * + ? {...})
...|...	Alternation/Alternative (ODER)
\X	Metazeichen X quotieren für ^ $. [ ] | * + ? { } ( ) \
\n \r \t ...	Escape-Sequenz (Steuerzeichen)

Zeichenklassen fassen Zeichen mit einer bestimmten Eigenschaft zusammen
(genauer Inhalt hängt von der locale-Einstellung per CHARACTER SET ab):

Klasse	Bedeutung
alnum	Buchstaben und Ziffern (alpha + digit)
alpha	Buchstaben (upper + lower)
cntrl	Steuerzeichen (Code 0-31)
blank	Leerzeichen (Space + Tabulator)
digit	Ziffern (0-9)
empty	Leerzeichen (Space + Tabulator)
graph	Druckbare Zeichen (print ohne Leerzeichen)
lower	Kleine Buchstaben (a-z)
print	Druckbare Zeichen (mit Leerzeichen)
punct	Interpunktionszeichen (print -- space -- alnum)
space	Leerraum (Space, FormFeed, Newline, Carriage Return, Tabulator)
upper	Große Buchstaben (A-Z)
xdigit	Hexadezimalziffern (0-9 + a-f/A-F)

Beispiele (keine Indices verwendbar!):

```
  SELECT nr, name
    FROM pers
      WHERE name REGEXP "^abc$";                  # Genau "abc"
  ... WHERE name = "abc";                         #   (analog)
  ... WHERE name REGEXP "abc$";                   # "abc" am Ende
  ... WHERE name LIKE "%abc";                     #   (analog)
  ... WHERE name REGEXP "abc";                    # "abc" enthalten
```

```
  ... WHERE name LIKE "%abc%";                      #   (analog)
  ... WHERE name NOT REGEXP "abc";                  # KEIN "abc" enthalten
  ... WHERE name NOT LIKE "%abc%";                  #   (analog)
  ... WHERE name RLIKE "^a*$";                      # "", "a", "aa", "aaa", ...
  ... WHERE name = "" OR name = "a" OR ...          #   (analog)
  ... WHERE name RLIKE "^a+$";                      # "a", "aa", "aaa", ...
  ... WHERE name = "a" OR name = "aa" OR ...        #   (analog)
  ... WHERE name RLIKE "thomas|hans";               # Enth. "thomas" oder "hans"
  ... WHERE name = "thomas" OR name = "hans";       #   (analog)
  ... WHERE name RLIKE "ab{1,8}a";                  # "aba","abba", ...max. 8 "b"
  ... WHERE name = "aba" OR name = "abba" OR ...;   #   (analog)
```

Mit LIKE, AND, OR nicht mehr formulierbar:

```
  ... WHERE name RLIKE "^[0-9]+$";         # Nur Zahl akzept. (mind. 1 Ziffer)
  ... WHERE name RLIKE "^[[:digit:]]+$";   # Nur Zahl akzept. (mind. 1 Ziffer)
```

9) MySQL-Funktionen

MySQL kennt eine große Menge eingebauter Funktionen, die beliebig in SELECT-Anweisungen zum Konvertieren/Verknüpfen von Spalten sowie in WHERE-Bedingungen einsetzbar sind (auf diesen Spalten liegende Indices sind in so einem Fall nicht nutzbar).

Beispiele für den Aufruf von in MySQL eingebauten Funktionen:

```
  SELECT CONCAT(vorname, " ", name) AS "Name" FROM pers;
  SELECT (YEAR(CURDATE()) - YEAR(geburtsdatum)) AS "Alter" FROM age;           ###
  SELECT NOW(), ADDDATE(NOW(), INTERVAL 14 DAY);
```

Folgende Funktionen sind in MySQL vordefiniert. Diese Liste ist erweiterbar durch Laden von externen User Defined Functions (UDF) und durch in SQL programmierte benutzerdefinierte Funktionen (--> 19) Stored Routines):

```
  +--------------+-------------------------------------------------------------+
  | Bereich      | Eingebaute Funktionen + ihre Parameter                      |
  +--------------+-------------------------------------------------------------+
  | Datenbank    | USER()   SESSION_USER()  SYSTEM_USER()   # Anmeldungsuser   |
  |              | CURRENT_USER()                           # Interner User    |
  |              | DATABASE()   SCHEMA()                                       |
  |              | CONNECTION_ID()                          # Sitzungs-ID      |
  |              | VERSION()                                # MySQL-Server     |
  +--------------+-------------------------------------------------------------+
  | Queryergebnis | FOUND_ROWS()        # Bei LIMIT mit SQL_CALC_FOUND_ROWS   |
  |              | LAST_INSERT_ID()    # Bei AUTO_INCREMENT                    |
  |              | ROW_COUNT()         # Bei INSERT, UPDATE, DELETE, REPLACE   |
  +--------------+-------------------------------------------------------------+
  | Arithmetik   | ABS(zahl)                     # Betrag                      |
  |              | GREATEST(zahl1, zahl2, ...)   # Größte Zahl                |
  |              | LEAST(zahl1, zahl2, ...)      # Kleinste Zahl               |
  |              | MOD(x, y)                     # Divisionsrest x / y         |
  |              | RAND([seed])                  # Zufallszahl 0 <= z < 1      |
  |              | SIGN(zahl)                    # Vorzeichen                  |
  +--------------+-------------------------------------------------------------+
  | Potenzierung | EXP(zahl)                                                   |
  |              | POW(num, exp)   POWER(num, exp)                             |
  |              | SQRT(zahl)                                                  |
  +--------------+-------------------------------------------------------------+
  | Logarithmus  | LN(zahl)                                                    |
  |              | LOG(zahl [, basis])                                         |
  |              | LOG2(zahl)                                                  |
  |              | LOG10(zahl)                                                 |
  +--------------+-------------------------------------------------------------+
  | Trigonometrie | ACOS(zahl)                                                 |
```

	ASIN(zahl)	
	ATAN(zahl)	
	ATAN2(x, y)	
	COS(zahl)	
	COT(zahl)	
	DEGREES(rad)	
	PI()	
	RADIANS(dec)	
	SIN(zahl)	
	TAN(zahl)	
Rundung	CEILING(zahl) CEIL(zahl)	
	FLOOR(zahl)	
	FORMAT(zahl, dezstellen [, locale])	
	ROUND(zahl [, dezstellen])	
	TRUNCATE(zahl, dezstellen)	
Bit	BIT_COUNT(zahl)	
	BIT_LENGTH(str)	
	EXPORT_SET(zahl, ein, aus [,trennzeichen, [numbits]])	
	MAKE_SET(bits, str1, str2, ...)	
String	COERCIBILITY(str)	
	COLLATION(str)	
	INSERT(str, pos, len, neu)	
	WEIGHT_STRING(str [AS {CHAR|BINARY} (n)] [LEVEL n] [,opt])	
Stringlänge	CHAR_LENGTH(str) CHARACTER_LENGTH(str)	
	LENGTH(str) CHAR/CHARACTER/OCTET_LENGTH(str)	
String-verkettung	CONCAT(str1, str2, ...)	
	CONCAT_WS(trennzeichen, str1, str2, ...)	
String-wiederholung	REPEAT(str, anz)	
	SPACE(anz)	
Stringteile extrahieren	LEFT(str, len)	
	MID(str, pos, len) --> SUBSTR(...)	
	RIGHT(str, len)	
	SUBSTR(str, pos) SUBSTRING(...)	
	SUBSTR(str, pos, len) SUBSTRING(...)	
	SUBSTR(str FROM pos FOR len) SUBSTRING(...)	
	SUBSTR(str FROM len) SUBSTRING(...)	
	SUBSTR_INDEX(str, zeichen, anz) SUBSTRING_INDEX)(...)	
String verkürzen + auffüllen	LPAD(str, len, fuellstr)	
	LTRIM(str)	
	RPAD(str, len, fuellstr)	
	RTRIM(str)	
	TRIM([BOTH | LEADING | TRAILING] [zeichen] [FROM] str)	
String-umwandlung	LCASE(str) LOWER(str)	
	UCASE(str) UPPER(str)	
	REVERSE(str)	
Stringsuche + ersatz	INSTR(str, teil)	
	REPLACE(str, alt, neu)	
	SOUNDEX(str)	
Stringvergl.	FIELD(str, str1, str2, ...) # --> ELT(...)	
	FIND_IN_SET(str, menge)	
	LOCATE(teil, str [, zahl])	
	MATCH (spalte1, ...) AGAINST (str [modifier])	
	POSITION(teil, str) --> LOCATE	
	STRCMP(str1, str2)	

Datum	CURDATE() CURRENT_DATE() # --> Datum YYYY-MM-DD
	DATE(datum/zeit)
	DATE_FORMAT(datum, format)
	EXTRACT(zeitabschnitt FROM datetime)
	FROM_DAYS(tage)
	LAST_DAY(datum)
	MAKEDATE(jahr, tag)
	STR_TO_DATE(str, formatmuster)
	UTC_DATE()
Datumteile	DAYNAME(datum)
	DAYOFMONTH(datum) DAY(datum)
	DAYOFWEEK(datum)
	DAYOFYEAR(datum)
	MONTH(datum)
	MONTHNAME(datum)
	QUARTER(datum)
	WEEK(datum)
	WEEKDAY(datum) --> Mo=0, Di=1, ..., Sa=5, So=6
	WEEKOFYEAR(datum)
	YEAR(datum)
	YEARWEEK(datum [, modus])
Datumrechnung	ADDDATE(datum, INTERVAL anz typ) --> DATE_ADD(...)
	DATE_ADD(datum, INTERVAL zeitspanne typ)
	DATE_SUB(datum, INVERVAL zeitspanne typ)
	DATEDIFF(neu_datum, alt_datum) # --> TAGE
	PERIOD_ADD(datum, monate)
	PERIOD_DIFF(datum1, datum2)
	SUBDATE(datum, INVERVAL anz typ) --> DATE_SUB(...)
	TO_DAYS(datum) --> Anz. Tage seit 01.01.0000
	TO_SECONDS(datum) --> Anz. Sekunden seit 01.01.0000
Zeit	CONVERT_TZ(datum/zeit, zeitzone, zeitzone)
	CURTIME() CURRENT_TIME()
	LOCALTIME()
	MAKETIME(stunde, minute, sekunde)
	SEC_TO_TIME(sek)
	TIME_FORMAT(zeit, format)
	TIME_TO_SEC(zeit)
	UTC_TIME()
Zeitteile	HOUR(zeit)
	MINUTE(zeit)
	MICROSECOND(zeit)
	SECOND(zeit)
Zeitrechnung	ADDTIME(zeit, zeit)
	SUBTIME(datum/zeit, datum/zeit)
	TIMEDIFF(zeit, zeit)
Timestamp	FROM_UNIXTIME(sek [,format]) # --> Datum
	LOCALTIMESTAMP()
	NOW([stellen]) CURRENT_TIMESTAMP([stellen])
	TIMESTAMP(datum, zeit)
	TIMESTAMPADD(intervall, wert, datum/zeit)
	TIMESTAMPDIFF(intervall, wert, datum/zeit)
	UNIX_TIMESTAMP([datum]) # --> Sekunden seit 1.1.1970
	UTC_TIMESTAMP()
Bedingung	CASE wert WHEN auswahl THEN wert ... ELSE wert END
	ELT(zahl, str1, str2, ...) # --> FIELD(...)
	IF(test, wert1, wert2)
	INTERVAL(x, grenze1, grenze2, ...)
NULL-Vergleich	COALESCE(wert1, wert2, ...)

	IFNULL(wert1, wert2)	
	ISNULL(ausdruck)	
	NULLIF(wert1, wert2)	
Konvertierung	ASCII(zeichen)	
	BIN(dezimalzahl)	
	CAST(ausdruck AS typ) CONVERT(ausdruck, typ)	
	CHAR(num1 [,num2, ...]) [USING zeichensatz]	
	CONV(zahl, basis1, basis2)	
	CONVERT(ausdruck USING zeichensatz)	
	GET_FORMAT(datentyp, formattyp)	
	HEX(dezimalzahl)	
	OCT(dezimalzahl)	
	ORD(str)	
	QUOTE(str)	
	UNHEX(str)	
	FROM_BASE64(str)	
	TO_BASE64(str)	
Zeichen-codierung + sortierung	CHARSET()	
	COERCIBILITY()	
	COLLATION()	
Advisory Locking (User Level)	GET_LOCK(name, timeout)	
	IS_FREE_LOCK(name)	
	IS_USED_LOCK(name)	
	RELEASE_LOCK(name)	
Komprimierung	COMPRESS(str)	
	UNCOMPRESS(str)	
	UNCOMPRESSED_LENGTH(str)	
Verschlüsseln Entschlüsseln	AES_DECRYPT(str, password)	
	AES_ENCRYPT(str, password)	
	DES_DECRYPT(str [, password])	
	DES_ENCRYPT(str [, password])	
	DECODE(blob, passwort)	
	ENCODE(blob, passwort)	
	ENCRYPT(str, salt)	
	OLD_PASSWORD(str)	
	PASSWORD(str)	
Hashing	CRC32(str)	
	MD5(str)	
	SHA(str)	
	SHA1(str)	
Eindeutige ID	UUID() # --> 128-Bit Zahl	
	UUID_SHORT() # --> 64-Bit Zahl	
Netzwerk	INET_ATON(adresse)	
	INET_NTOA(zahl)	
Sonstige	BENCHMARK(anz, funktion)	
	LOAD_FILE(dateiname) # Datei als String einlesen	
	MASTER_POS_WAIT(dateiname, pos [, timeout])	
	SLEEP(sek) # Optional .microsec	

9a) Aggregatfunktionen und Gruppierung (Aggregation)

Aggregatfunktionen fassen in einer SELECT-Anweisung die Werte einer Spalte <Col> über ALLE Datensätze zusammen. Kommt eine GROUP-BY-Klausel bezüglich einer (anderen) Spalte <Col2> vor, erfolgt die Zusammenfassung PRO

unterschiedlichem Wert dieser Spalte <Col2> (mehrere GROUP-BY-Spalten sind möglich). MySQL kennt folgende Aggregatfunktionen:

Funktion	Beschreibung
COUNT(*)	Anzahl ALLER Datensätze (auch NULL!)
COUNT(<Col>)	Anzahl aller Werte ungleich NULL
COUNT(DISTINCT <Col>)	Anzahl aller verschiedenen Werte ungleich NULL
SUM(<Col>)	Summe aller Werte
SUM(DISTINCT <Col>)	Summe aller verschiedenen Werte (MY!5.1)
AVG(<Col>)	Durchschnitt aller Werte
AVG(DISTINCT <Col>)	Durchschnitt aller verschiedenen Werte (MY!5.1)
MIN(<Col>)	Minimum aller Werte
MAX(<Col>)	Maximum aller Werte
STD(<Col>)	Standardabweichung Werte ungleich NULL (MY!)
STDDEV(<Col>)	Standardabweichung Werte ungleich NULL (MY!)
STDDEV_POP(<Col>)	Standardabweichung Werte ungleich NULL
STDDEV_SAMP(<Col>)	Standardabweichung aller Werte (auch NULL!)
VARIANCE(<Col>)	Varianz aller Werte ungleich NULL (MY!)
VAR_POP(<Col>)	Varianz aller Werte ungleich NULL
VAR_SAMP(<Col>)	Varianz aller Werte (auch NULL!)
BIT_AND(<Col>)	Bitweise UND-Verknüpfung aller Werte (MY!)
BIT_OR(<Col>)	Bitweise ODER-Verknüpfung aller Werte (MY!)
BIT_XOR(<Col>)	Bitweise XOR-Verknüpfung aller Werte (MY!)
GROUP_CONCAT()	Verkettung aller ermittelten Gruppenwerte (MY!)

HINWEIS: Je nachdem ob eine Spalte <Col> NULL-Werte enthält oder nicht gilt:

```
COUNT(*) = COUNT(<Col>)    # Kein NULL-Wert in <Col>
COUNT(*) > COUNT(<Col>)    # Mind. ein NULL-Wert in <Col>
```

Beispiele:

```
CREATE TABLE bestellung (
  nr     INT,
  anz    INT,
  preis  FLOAT,
  summe  FLOAT,
  rabatt FLOAT
);

INSERT INTO bestellung VALUES
  (1, 2, 123.45, 2 * 123.45, 10),
  (2, 5,  10.45, 5 *  10.45,  0),
  (3, 1,  67.00, 1 *  67.00,  5);

SELECT COUNT(name)      AS "Artikelanzahl"      FROM bestellung;
SELECT AVG(preis)       AS "Durchschnittspreis" FROM bestellung;
SELECT SUM(preis * anz) AS "Gesamtpreis"        FROM bestellung;
SELECT MIN(preis)       AS "Billig",
       MAX(preis)       AS "Teuer"              FROM bestellung;
```

Mittels der Klausel GROUP BY lassen sich Datensätze basierend auf GLEICHEN Werten einer oder mehrerer Spalten GRUPPIEREN (zusammenfassen). Über die obigen Aggregatfunktionen entstehen dabei aus den Einzelwerten der RESTLICHEN Spalten der jeweils zusammengefassten Datensätze Gesamtwerte.

Eigenschaften:
* Pro VERSCHIEDENEM Wert einer GROUP BY Spalte entsteht EINE Ergebniszeile

```
  * Datensätze mit NULL-Wert in gruppierter Spalte werden (meist) ignoriert
  * Die gruppierten Spaltenwerte sind sortierbar:
    + aufsteigend (ASC, STD)
    + absteigend (DESC)
  * Gesamtzeile bzgl. gruppierter Spalte mit Zusatz "WITH ROLLUP" möglich
    + Spaltenwert dieses Datensatzes ist "NULL"
    + Werttext für Gesamtzeile per IFNULL festlegen
        IFNULL(<GroupCol>, "<NullText>") AS <ColName>
    + Nicht zusammen mit Sortierung verwendbar
    + NULL nicht in HAVING als Bedingung verwendbar

Syntax:

  SELECT ...
    FROM ...
      GROUP BY <Col> [ASC|DESC] [WITH ROLLUP] {, ...}
        [FOR UPDATE |
         LOCK IN SHARE MODE |
         PROCEDURE <ProcExternal>(...) |
         PROCEDURE ANALYSE([<MaxElem>[, <MaxMem>]]) ]

Beispiele:

  SELECT SUM(summe) AS "Gesamt"
    FROM bestellung
      GROUP BY nr                                    # STD: ASC (aufsteigend)
        HAVING Gesamt > 100;

  SELECT nr, AVG(preis) AS "Durchschnittspreis"
    FROM artikel
      GROUP BY nr DESC                               # STD: ASC (aufsteigend)
        HAVING COUNT(nr) > 1;

  CREATE TABLE verkaeufe (
    jahr    INT NOT NULL,
    land    VARCHAR(20) NOT NULL,
    produkt VARCHAR(32) NOT NULL,
    gewinn  INT
  );

  INSERT INTO verkaeufe (jahr, land, produkt, gewinn) VALUES
    (2009, "Deutschland", "Mixer",   11.3),
    (2010, "Deutschland", "Mixer",   10.7),
    (2009, "England",     "Mixer",   11.0),
    (2010, "England",     "Mixer",    9.5),
    (2009, "Frankreich",  "Mixer",   12.1),
    (2010, "Frankreich",  "Mixer",   10.2),
    (2009, "Deutschland", "Toaster", 21.3),
    (2010, "Deutschland", "Toaster", 20.7),
    (2009, "England",     "Toaster", 21.0),
    (2010, "England",     "Toaster", 19.5),
    (2009, "Frankreich",  "Toaster", 22.1),
    (2010, "Frankreich",  "Toaster", 20.2),
    (2009, "Deutschland", "Kocher",   1.3),
    (2010, "Deutschland", "Kocher",   0.7),
    (2009, "England",     "Kocher",   1.0),
    (2010, "England",     "Kocher",   0.5),
    (2009, "Frankreich",  "Kocher",   2.1),
    (2010, "Frankreich",  "Kocher",   0.2);

  SELECT jahr, SUM(gewinn)                  # --> jahr SUM(gewinn)
    FROM verkaeufe                          #     2009         102
      GROUP BY jahr;                        #     2010          94

  SELECT jahr, SUM(gewinn)                  # --> jahr SUM(gewinn)
    FROM verkaeufe                          #     2009         102
```

```
    GROUP BY jahr                          #     2010          94
      WITH ROLLUP;                         #     NULL         196

SELECT IFNULL(jahr, "GESAMT"), SUM(gewinn) # --> jahr SUM(gewinn)
  FROM verkaeufe                           #     2009         102
    GROUP BY jahr                          #     2010          94
      WITH ROLLUP;                         #     GESAMT       196

SELECT jahr, land, SUM(gewinn)             # Mehrere Gruppen
  FROM verkaeufe                           #
    GROUP BY jahr ASC, land DESC;          #

SELECT jahr, land, SUM(gewinn)             #
  FROM verkaeufe                           #
    GROUP BY jahr, land                    # Mehrere Gruppen
      WITH ROLLUP;                         # + Gesamt

SELECT IFNULL(jahr, "JAHRE")    AS jahr,   # Gesamtwert umbenennen
       IFNULL(land, "LAENDER")  AS land,   # Gesamtwert umbenennen
       SUM(gewinn)                         #
  FROM verkaeufe                           #
    GROUP BY jahr, land                    # Mehrere Gruppen
      WITH ROLLUP;                         # + Gesamt
```

10) Schlüssel (Key) und Index

10a) Eigenschaften

"Schlüssel" (Key) und "Index" haben zunächst nichts miteinander zu tun:
- Ein "Schlüssel" adressiert jeden Datensatz einer Datenbanktabelle EINDEUTIG
  - Aus einer Spalte oder Kombination mehrerer Spalten gebildet
  - Nutzen: Beziehung zwischen Tabellen herstellen
  - NUR EIN (primärer) Schlüssel pro Tabelle möglich (PRIMAY KEY)
- Ein "Index" wird für eine Spalte oder Kombination von Spalten angelegt, nach denen häufig gesucht oder sortiert wird
  - Ähnlich einem Inhaltsverz. zu einem Buch:
    - Suche nach Datensätzen beschleunigen (WHERE)
    - Sortierung von Datensätzen beschleunigen (ORDER BY)
    - Eindeutigkeit prüfen (UNIQUE KEY)
    - Beziehung herstellen (FOREIGN KEY)
  - MEHRERE Indices pro Tabelle möglich
  - Muss nicht unbedingt eindeutig (UNIQUE) sein
- Zusammenhang:
  - Zu jeden Schlüssel einer Datenbank meist ein Index erstellt (kein MUSS)

Unterscheidung:
- Primärschlüssel (PRIMARY KEY)
  - Kennzeichnet JEDEN Datensatz EINDEUTIG (z.B. Abteilungsnummer)
  - Meist (abstrakte) fortlaufende Nummer
  - Häufig automatisch beim Anlegen eines Datensatzes erzeugt
  - NUR EIN Primärschlüssel pro Tabelle möglich und notwendig
  - Eine Primärschlüsselspalte darf NIEMALS LEER sein (NOT NULL)
  - Komplexe/lange Primärschlüssel verlangsamen Datenbank-Operationen (am besten nur Spalte vom Typ "kleine ganze Zahl" verwenden)
- Sekundärschlüssel
  - Kennzeichnet ebenfalls Datensatz eindeutig (z.B. Abteilungsname)
  - MEHRERE Sekundärschlüssel pro Tabelle möglich
  - Sekundärschlüssel darf LEER sein (NULL)
- Fremdschlüssel (FOREIGN KEY)
  - Bezeichnet die Übereinstimmung einer/mehrerer Spalten in einer Tabelle mit der/den Primärschlüsselspalte(n) einer anderen Tabelle ("Verweise")
  - MEHRERE Fremdschlüssel pro Tabelle möglich
  - KEINE ZIRKULÄREN Referenzen auf Fremdschlüssel erstellen!

```
    + Häufig in "Lookup-Tabelle" verwendet, die 2 oder mehr Tabellen verknüpft
    + 1:1-Beziehung, 1:N-Beziehung, N:M-Beziehung
  * Index
    + Erleichtern Suche nach Datensätzen vor allem in großen Tabellen
    + Einfügen, Ändern, Löschen von Datensätzen erfordert Index-Aktualisierung
      --> Kann viel Zeit kosten
    + TIP: Indices großer Tabellen erst NACH dem Füllen anlegen

Hinweise:
  * Nur 1 Primärschlüssel pro Tabelle erlaubt
    + Automatisch UNIQUE
    + Automatisch NOT NULL
    + Automatisch auch ein Index
    + Darf mehr als 1 Spalte enthalten!
  * Mehrere UNIQUE Indices anlegbar
    + Automatisch Sekundärschlüssel
    + NULL-Werte darin erlaubt
  * Index auf Spaltenpräfix beschränkbar (z.B. name(5), vorname(5))
    (nicht auf Suffix --> TRICK: Daten per REVERSE() umgedreht speichern)
    - Sollte selektiv genug sein
    - Spart Platz
    - Bei Spaltentyp TEXT/BLOB notwendig
  * Tabelleninhalt wird durch Index nicht verändert
  * Index = "Zeiger" auf Datensätze in sortierter Reihenfolge
  * Index belegt zusätzlichen Plattenplatz
  * Index enthält KEINE zusätzlichen Daten,
    sondern erlauben nur schnelleren Lese-Zugriff
    + Index daher jederzeit löschbar
    + Index daher jederzeit aus Tabellendaten wieder herstellbar
      (in Backup nicht aufnehmen)
    + Beschleunigt nur Zugriff über indizierte Spalten
  * Beim Erzeugen + Ändern von Datensätzen kosten Indices zusätzlichen Aufwand,
    da jeder Schreibvorgang auf Datensätzen die Indices mitpflegen muss
    - Nachträgliche Index-Erzeugung --> Tabelle für Schreiben gesperrt!

HINWEIS: Manche DB-Systeme fordern zu jeder Tabelle einen Primärschlüssel,
MySQL kann darauf auch verzichten (nur in Ausnahmefällen wirklich sinnvoll).

10b) Erstellen/entfernen
------------------------

Primärschlüssel erstellen/entfernen (automatisch/muss NOT NULL + UNIQUE sein):
  * Beim Erzeugen einer Tabelle (1.Form mehrspaltig, 1+2.Form = einspaltig)
      CREATE TABLE <Tbl> (                  CREATE TABLE <Tbl> (
        nr INT NOT NULL AUTO_INCREMENT,       nr INT AUTO_INCREMENT PRIMARY KEY,
        ...,                                  ...,
        PRIMARY KEY (nr)                      ...
      );                                    );
  * Nachträglich erzeugen
    - Nur möglich, wenn Spalte "nr" NOT NULL + UNIQUE (KEINE doppelten Werte)!
        ALTER TABLE <Tbl> ADD PRIMARY KEY (nr);   # OK
        CREATE PRIMARY KEY ON <Tbl> (nr);         # FALSCH! --> ALTER TABLE!
  * Entfernen
      ALTER TABLE <Tbl> DROP PRIMARY KEY;

Sekundärschlüssel erstellen/entfernen:
  * Beim Erzeugen einer Tabelle
      CREATE TABLE pers (
        nr      INT      NOT NULL AUTO_INCREMENT,
        name    CHAR(30) NOT NULL,
        vorname CHAR(30) NOT NULL,
        PRIMARY KEY (nr),
        UNIQUE INDEX idx1 (name, vorname)   # Oder nur UNIQUE ohne INDEX
      );
  * Nachträglich erzeugen
    + Nur möglich, wenn Spalte "nr" UNIQUE (KEINE doppelten Werte enthält!),
```

```
    + NULL-Werte sind nicht erlaubt
        ALTER TABLE pers ADD UNIQUE INDEX idx1 (name, vorname);   # INDEX weglassbar
        CREATE UNIQUE INDEX idx1 ON pers (name, vorname);
  * Entfernen
      ALTER TABLE pers DROP INDEX idx1;   # Nur INDEX (ohne UNIQUE, ...)

Index erstellen, entfernen, (de)aktivieren und anzeigen
  * Allgemeine Syntax:
      CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX <Idx>
        [USING {BTREE | HASH | RTREE}]
          ON <Tbl> (<Col> [(<Len>)] [ASC | DESC], ...);
    + Indexname <Idx> nötig, um Index gezielt entfernen zu können
      --> MUSS pro Tabelle eindeutig sein
    + Index-Art wählbar (UNIQUE, FULLTEXT, SPATIAL)
        CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX <Idx> ...;
    + Indexsortierung wählbar (in MySQL derzeit ignoriert, STD: ASC)
        ... ON <Tbl> (<Col1> ASC, <Col2> DESC, ...);
    + Für Index benutzte Spaltenpräfix-Länge wählbar
        ... ON <Tbl> (<Col1>(<Len1>), <Col2>(<Len2>), ...);
    + Index-Struktur wählbar (BTREE, HASH, RTREE)
      - MyISAM: BTREE, RTREE
      - InnoDB: BTREE
      - MEMORY: BTREE, HASH
      - NDB:    BTREE, HASH
        CREATE INDEX <Idx> [USING {BTREE | HASH | RTREE}] ...;
  * Beim Erzeugen einer Tabelle
      CREATE TABLE <Tbl> (
        nr       INT      NOT NULL,
        vorname  CHAR(30) NOT NULL,
        name     CHAR(30) NOT NULL,
        ...,
        INDEX idx1 (nr),
        INDEX idx2 (name, vorname)
      );
  * Nachträglich erzeugen
      CREATE INDEX <Idx> ON <Tbl> (<Col>, ...);
      ALTER TABLE <Tbl> CREATE INDEX <Idx> (<Col>, ...);
  * Entfernen
      DROP INDEX <Idx> ON <Tbl>;
      ALTER TABLE <Tbl> DROP INDEX <Idx>;
  * Aktivieren/deaktivieren
      ALTER TABLE <Tbl> DISABLE KEYS;   # Alle Non-UNIQUE Indices abschalten
      ALTER TABLE <Tbl> ENABLE KEYS;    # Alle Non-UNIQUE Indices wieder aufbauen
      ALTER INDEX <Idx> ACTIVE;         # In MySQL NICHT möglich (MY!)
      ALTER INDEX <Idx> INACTIVE;       # In MySQL NICHT möglich (MY!)
  * Indices einer Tabelle anzeigen lassen (mit Kardinalitätswert N/S)
      SHOW INDEX FROM <Tbl>;
  * Indices einer Datenbank anzeigen lassen (NICHT unter MySQL, MY!)
      SHOW INDICES;

10c) FULLTEXT-Index
-------------------

Eigenschaften:
  * Nur bei Engine "MyISAM" möglich!
  * Nur auf Spaltentypen CHAR, VARCHAR, TEXT möglich
  * Suche nach einzelnen Worten und beliebig langen Texten möglich
  * Nach einigen englischen "Stopwords" wir NICHT gesucht (z.B. "and", "or")
  * Suche NICHT mit LIKE/REGEX möglich
  * Spezielle Suchsyntax:
      MATCH (<Col1>, ...) AGAINST (<String> [<Modifier>])
    mit <Modifier>
      IN BOOLEAN MODE                                 # Syntax +xxx -yyy ...
      IN NATURAL LANGUAGE MODE                        # Ab MY!5.1.7
      IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION   # Ab MY!5.1.7
      WITH QUERY EXPANSION                            #
```

10d) SPATIAL-Index

Eigenschaften:
  * Nur bei Engine "MyISAM" möglich!
  * Nur auf Geometrie-Spaltentypen möglich
  * Basiert auf dem OpenGIS Geometrie-Modell

10e) Index-Nutzung

  * Indices verwendet bei SELECT (UPDATE, DELETE, REPLACE), ORDER BY und JOIN.
  * Datenbank sucht zu jeder Abfrage SELBSTÄNDIG sinnvoll einzusetzende Indices
    (mit Hilfe des "Query Optimizer")
  * Tabellen/Indices enthalten Informationen über Selektivität und statistische
    Verteilung der indizierten Daten (Histogramm)
    + Bsp: Index auf Spalte "Geschlecht" ist wenig nützlich,
           da immer etwa auf die Hälfte der Daten zugegriffen wird
    + Bsp: Index auf Schlüsselspalte "nr" ist sehr nützlich (da eindeutig)
  * Index kann eine/mehrere Spalten einer Tabelle umfassen ("Composite Index")
    + Umfasst er mehrere Spalten, dann sind alle Spalten-Kombinationen
      von links nach rechts ebenfalls enthalten (Präfixe)!
    + Ein mehrspaltiger Index ist immer dann nutzbar,
      wenn ALLE Spalten von links nach rechts in WHERE-Klausel vorkommen
    + Beispiel:
        CREATE INDEX multi ON adresse (name, vorname, strasse);
      Folgende Abfragen nutzen diesen Index:
        SELECT * FROM adresse WHERE name LIKE "A%";
        SELECT * FROM adresse WHERE name LIKE "A%" AND
                                    vorname LIKE "B%"
        SELECT * FROM adresse WHERE name LIKE "A%" AND
                                    vorname LIKE "B%" AND
                                    strasse LIKE "C%";
      Folgende Abfragen können diesen Index NICHT nutzen:
        SELECT * FROM adresse WHERE vorname LIKE "B%";
        SELECT * FROM adresse WHERE strasse LIKE "B%";
        SELECT * FROM adresse WHERE vorname LIKE "B%" AND
                                    strasse LIKE "C%";
  * Werden nur Datenspalten gelesen, die in EINEM Index enthalten sind:
    + Index enthält bereits alle benötigten Daten
    + Zugriff auf Datentabelle nicht nötig
    + "Covering Index"
    + Beschleunigung
    + Beispiel:
        SELECT name FROM adresse WHERE name LIKE "A%" AND
                                       vorname LIKE "B%"
  * Index bei folgenden Vergleichen verwendbar:
      <=>   # NULL-sicheres "="
      =   # Gleich
      <   # Kleiner
      <=   # Kleiner oder gleich
      >   # Größer
      >=   # Größer oder gleich
      LIKE "abc%"   # Platzhalter "%" hinten
      LIKE "abc__"   # Platzhalter "_" hinten
      BETWEEN <Min> AND <Max>   # Bereich einschließlich Grenzen
  * KEIN Index bei folgenden Vergleichen verwendbar:
      !=   # Ungleich
      <>   # Ungleich
      LIKE "%abc"   # Platzhalter "%" vorne
      LIKE "__abc"   # Platzhalter "_" vorne
      RLIKE "abc"   # Regulärer Ausdruck
      REGEX "abc"   # Regulärer Ausdruck
      NOT RLIKE "abc"   # Regulärer Ausdruck
      NOT REGEX "abc"   # Regulärer Ausdruck

```
      NOT BETWEEN <Min> AND <Max>  # Nicht in Bereich einschließlich Grenzen
  * Werden mehrere Bedingungen per AND/OR verknüpft, dann wird meist der Index
    zu den Spalten einer Bedingung verwendet, der am SELEKTIVSTEN ist, d.h.
    die wenigsten Datensätze selektiert. Diese Datensätze werden alle gelesen
    und die restlichen Bedingungen dagegen geprüft
  * Index auch bei Sortierung ORDER BY und Gruppierung GROUP BY verwendbar
  * Schlüsselworte "KEY" und "INDEX" sind synonym

Indices werden verwendet bei:
  * Auswahl Datensätze gemäß WHERE-Klausel
  * Datensätze weglassen
  * Doppelte Einträge rauswerfen (DISTINCT)
  * JOIN: Typ und Länge müssen gleich sein (Konvertierung verhindert Index!)
  * MIN/MAX für bestimmte Spalte
  * ORDER BY/GROUP BY falls ausgeführt auf Präfix eines Index
  * Daten + Bedingung durch Index befriedigt und numerisch
  * = > >= < <= BETWEEN LIKE (mit fixem Präfix)
  * <Col> IS NULL wenn Index auf <Col>
  * Index muss in jeder AND-Gruppe der WHERE-Klausel verwendbar sein
    (kann auch 1-elementig ohne AND sein!)
  * LIMIT

Indices werden NICHT verwendet bei:
  * Lese-Operationen, die großen Prozentsatz einer Tabelle lesen
  * OR auf Spalten in verschiedenen Indices --> IN, UNION [ALL]!
  * Regulären Ausdrücken


10f) Index-Optimierung
----------------------

Richtlinien für das Erstellen von Indices:
  * Primär-, Sekundär- und Fremdschlüssel erhalten automatisch einen Index
    + PRIMARY KEY so kurz wie möglich und eindeutig
    + Nur im absoluten Ausnahmefall keinen Primärschlüssel definieren
      (Datensätze dann nicht eindeutig identifizierbar)
  * Spalten mit wenigen unterschiedlichen Werten (z.B. Anrede, Geschlecht)
    bringen mit Index KEINEN Geschwindigkeitsgewinn (Kardinalität zu gering)
  * Indices auf Tabellen einsetzen, die oft gefiltert/gruppiert/sortiert werden
    + Indices auf Tab. mit wenig Datensätzen (<=12) bringt KEINEN Geschw.gewinn
    + Mehr als 10% Datensätze einer Tab. ständig gelesen --> Index nicht sinnvoll
  * Zwei Indices notwendig für auf- + absteigendes Sortieren (ASC, DESC)
  * Sortierkriterium an letzter Stelle im Index --> (teuren) Filesort sparen
  * Aggregatfunktionen nutzen keinen Index
    + COUNT, SUM, AVG, STDDEV, ...
    + Da alle Datensätze durchlaufen werden müssen
    + MIN/MAX für bestimmte Spalte schon
  * Ohne Indices wäre immer ein "Full Table Scan" notwendig
    + Beim (fast) vollständigen Durchlesen einer Tabelle KEINEN Index benutzen!
      --> Evtl. Cursor, Handler besser
  * MySQL: Datensätze und Indices IMMER in getrennten Dateien (InnoDB?)
    + Kostet zusätzlichen READ bei Zugriff Index --> Daten
    + Evtl. "Full Table Scan" schneller --> Index lesen unnötig
    + Evtl. "Covering Index" ausreichend --> Daten lesen unnötig
    + Nur 1x Speicherplatz für Indexwert bei Datensätzen mit gleichem Indexwert
    + Delete-Operationen "degenerieren" Tabelle nicht
    + Caching getrennt möglich
  * Nur unbedingt notwendige Indices nutzen (Schreibperformance sonst schlecht)
  * Am häufigsten benutzte Spalten(kombination) indizieren
    + Mehrere Spalten nach abnehmender Häufigkeit sortieren
    + Zusammengesetzter Index nutzbar in mehreren Fälle: 1.Sp, 1+2.Sp, ...
    + Spalte mit mehr doppelten Werten vorne (Kardinalität)
  * Nur ausreichend lange Präfixe indizieren (Platz, Hits)
    + Für BLOB/TEXT ein MUSS!
    + Kardinalität reduziert
    + TRICK: REVERSE oder SUBSTR verwenden, falls Anfang statisch
```

Eigenschaften von Indices:
  * Automatisch Präfix- und Längen-komprimiert (MY!)
  * MySQL verwendet pro Tabelle in einer Query maximal EINEN Index
    + Normalerweise Index mit kleinster Treffermenge (Anz. passender Datensätze)
    + Evtl. mehrspaltiger Index sinnvoll!

Verschiedene Indexverfahren möglich (MY!):
  * BTREE: Fast immer möglich
  * RTREE: Für geometrische Daten
  * HASH: Bei MEMORY-Tabellen
    + Einschränkung des HASH-Index
      - NUR für = und <=>
      - Nicht für != <> < <= > >= verwendbar
      - Nicht für ORDER BY und GROUP BY verwendbar
      - Nur vollständiger Index nutzbar, keine Präfixe

Index-Hint: Manuell Hinweis zur Indexnutzung in SELECT pro Tab. angebbar (MY!):
  * Nach FROM <Tbl>:
      USE   INDEX = NUR diese Indices benutzen ("Full Table Scan" möglich)
      FORCE INDEX = Analog USE INDEX ("Full Table Scan" NUR wenn unmöglich)
      IGNORE INDEX = Diese Indices NICHT benutzen
  * Name des Primären Index ist "PRIMARY"
  * USE INDEX Liste darf auch leer sein --> Keine Indices benutzen!
  * Auswahl für welche Operation (fehlt --> Für alle 3 Fälle gültig):
      FOR JOIN   = Für Tabellen-Join
      FOR ORDER BY = Für Sortierung
      FOR GROUP BY = Für Gruppierung

  +---+
FROM <Tbl> USE INDEX [FOR {JOIN	ORDER BY	GROUP BY}] (<Idx1>, ...)
FORCE INDEX [FOR {JOIN	ORDER BY	GROUP BY}] (<Idx1>, ...)
IGNORE INDEX [FOR {JOIN	ORDER BY	GROUP BY}] (<Idx1>, ...)
  +---+

Beispiel:

```
  SELECT * FROM table1 USE INDEX (idx1, idx2)
    WHERE col1 = 1 AND col2 = 2 AND col3 = 3;

  SELECT * FROM table1 IGNORE INDEX (idx3)
    WHERE col1 = 1 AND col2 = 2 AND col3 = 3;

  SELECT * FROM t1 USE INDEX (i1)
                   IGNORE INDEX FOR ORDER BY (i2),
                 t2 FORCE INDEX (i3),
    WHERE t1.id = t2.id
      ORDER BY a;
```

Indices reorganisieren:
  * Sinnvoll nach dem Laden von Daten
  * Cardinality = Durchschnittliche Datensatzanz. mit gleichem Wert berechnen
      ANALYZE TABLE <Tbl>;
      OPTIMIZE TABLE <Tbl>;
      myisamchk --analyze / -a <Tbl>
      SHOW INDEX FROM <Tbl>;
      myisamchk --description --verbose <Tbl>
  * Evtl. Index UND Daten gemäß einem Index sortieren
    (für sortierten Zugriff auf alle Daten gemäß UNIQUE Index)
    (kann beim 1. Mal sehr lange dauern)
      myisamchk --sort-index / -S <Tbl>   # Indices sortieren
      myisamchk --sort-records=N / -R N <Tbl> # Gemäß Index N sortieren

10g) Fremdschlüssel (Foreign Keys) und Referenzielle Integrität

Grund für Fremdschlüssel:
  * Dokumentieren Beziehungen zwischen Tabellen (Master/Detail, Vater/Kind)
  * Verhindern Einfügen von Inkonsistenzen in DB durch Programmierer
    + "Referenzielle Integrität" (Referential Integrity) erhalten
    + Reihenfolge von Einfüge/Änderungs/Lösch-Operationen nicht zu beachten
      (verhindert "verwaiste Kinder" = orphaned childs)
    + Fehlerbehandlung bei Unterbrechung
  * Zentrale Constraint-Prüfung (in Anwendung verzichtbar + einheitlich)
  * Kaskadierende UPDATEs/DELETEs vereinfachen Anwendungscode

Zusammenhang zwischen Tabellen = Beziehung zw. Primär- und Fremdschlüsseln:
  * Beim UPDATE/DELETE eines Datensatzes aus einer (Eltern)Tabelle werden alle
    damit über den gleichen Fremdschlüssel verknüpften Datensätze in zugehörigen
    (Kind)Tabellen ebenfalls AUTOMATISCH von der Datenbank gelöscht/geändert
  * Beim INSERT eines Datensatzes in eine (Kind)Tabelle mit Fremdschlüssel
    wird geprüft, ob eine Entsprechung in referenzierter (Eltern)Tabelle
    vorhanden ist. Wenn nein: Fehlermeldung + Datensatz nicht eingefügt

Eigenschaften:
  * Grundsätzlich bei Tabellen-Definition angebbar
    + Bei allen Engines außer "InnoDB" syntaktischer Zucker (reine Doku)
    + Nur bei "InnoDB" gespeichert (und "PBXT")
    + Speicherung und Implementierung für "MyISAM" geplant
  * Workaround falls nur ON DELETE nötig:
    + Multi-Table DELETE
  * Erzeugen zusätzlichen Overhead
    + Besser selber machen oder an DB delegieren?
      Hängt von Anwendung ab (1x implementieren statt mehrmals)
  * Restore individueller Tabellen von Backup erschwert
    --> Foreign Keys + Constraints + Trigger temporär abschalten!
  * Sinnvoll für Cascading Updates/Deletes
  * Nicht sinnvoll für Constraints --> Trigger besser --> Noch besser ENUM-Typ

Bedingungen:
  * Bezugstabelle muss vorhanden sein
  * Datentyp und Datengröße korrespondierender Spalten MUSS identisch sein!
    (bei CHAR/VARCHAR darf Länge verschieden sein)
  * Auf korrespondierenden Spalten muss ein Index liegen
  * Verknüpfte Spalten müssen NOT NULL sein und UNIQUE Index haben
    (wird aber nicht erzwungen)
  * Bei temporären Tabellen NICHT erlaubt

Vorteile/Nachteile:
  * Nicht umsonst (zusätzlicher Lookup bei jeder Änderung)
  * Arbeitet "Zeile für Zeile" (auch bei multiplem INSERT/UPDATE/DELETE)
  * Index notwendig (evtl. groß und zu sonst nichts nutze)

Erzeugen einer (Kind)Tabelle mit Fremdschlüsseln einer/mehrere (Eltern)Tabellen
("CONSTRAINT <Name>" optional):

```
  CREATE TABLE <Tbl> (    # Kindtabelle
    ma_id INT NOT NULL,   # Fremdschlüssel aus Elterntabelle
    pr_id INT NOT NULL,   # Fremdschlüssel aus Elterntabelle
    ...,
    FOREIGN KEY (ma_id) REFERENCES mitarbeiter (id)   # Bezug zur Elterntabelle
      ON UPDATE CASCADE                               # E-Aktion --> K-Aktion
      ON DELETE CASCADE,                              # E-Aktion --> K-Aktion
    CONSTRAINT cpj FOREIGN KEY (pr_id) REFERENCES projekt (id)  # Bezug z.E.
      ON UPDATE CASCADE                               # E-Aktion --> K-Aktion
      ON DELETE CASCADE                               # E-Aktion --> K-Aktion
  ) ENGINE = "InnoDB";
```

Fremdschlüssel nachträglich erzeugen (nur möglich, wenn verknüpfte Tabellen ex.
sowie verknüpfte Spalten NOT NULL sind und KEINE doppelten Werte enthalten):

```
  ALTER TABLE <Tbl> ADD [CONSTRAINT <Name>]     # Kindtabelle
    FOREIGN KEY (<Col>, ...)                    # Spalte in Kindtabelle
      REFERENCES <Tbl> (<Col>, ...)             # Spalten in Elterntabelle
        [MATCH <MatchOpt>]                      # Match-Klausel: Weglassen!
          [ON UPDATE <RefOpt>]                  # STD: NO ACTION
          [ON DELETE <RefOpt>]                  # STD: NO ACTION
    ...;

Verhalten eines Fremdschlüssels steuern:
  * MATCH-Klausel <MatchOpt> definiert Behandlung von NULL-Werten in
    mehrspaltigen Fremdschlüsseln beim Vergleich mit dem Primärschlüssel
    (derzeit ignoriert, d.h. immer SIMPLE)
      MATCH SIMPLE         # Mehrsp. Fremdschl. darf teilw./ganz NULL sein (STD)
      MATCH PARTIAL        # Mehrsp. Fremdschl. darf nicht vollständig NULL sein
      MATCH FULL           # Mehrsp. Fremdschl. muss vollständig ungl. NULL sein
  * Verhalten beim Einfügen/Löschen steuern:
      ON DELETE <RefOpt>   # Beim Löschen von Zeilen in Elterntabelle
      ON UPDATE <RefOpt>   # Beim Ändern von Zeilen in Elterntabelle
  * Referenz-Optionen <RefOpt> (STD: Anweisung wird abgebrochen)
      NO ACTION            # Abbruch der Änderung in Elterntabelle (STD)
      RESTRICT             #   (analog NO ACTION)
      CASCADE              # Löschen/Ändern transitiv auf Kindtabelle ausdehnen
      SET NULL             # Ref. Sp. in Kindtab. auf NULL setzen
      SET DEFAULT          # Ref. Sp. in Kindtab. auf Default setzen (nicht MY!)

Fremdschlüssel aus Kindtabelle entfernen:

  ALTER TABLE <Tbl> DROP FOREIGN KEY <Idx>;

Fremdschlüssel in Kindtabelle anzeigen:

  SHOW CREATE TABLE <Tbl>;
  SHOW TABLE STATUS FROM <Db> LIKE <Tbl>;

Fremdschlüssel überprüfen vor Daten laden aus- und danach wieder einschalten:

  SET foreign_key_checks = 0;    # Prüfung ausschalten
  SOURCE <SqlFile>;              # Kein "..." um <SqlFile>!
  SET foreign_key_checks = 1;    # Prüfung einschalten


11) Joins
---------

Ein Join ist eine Verknüpfung von zwei Tabellen zu einer Gesamttabelle über
eine Abhängigkeit zwischen den Tabellen (Primärschlüssel + Fremdschlüssel).
Das Ergebnis ist wieder eine (temporäre) Tabelle (und wieder ein JOIN mit einer
anderen Tabelle durchgeführt werden). MySQL unterstützt folgende JOIN-Arten:

  +---------------+-------------------------------------------------------+
  | Typ           | Bedeutung                                             |
  +---------------+-------------------------------------------------------+
  | [CROSS] JOIN  | Kombination ALLER Datensätze der 1. Tabelle mit ALLEN |
  |               | der 2. Tabelle ("Kreuzprodukt", "Karthesisches Produkt")
  |               | (KEINE Beziehung, andere Schreibweise: <Tbl1>, <Tbl2>).
  +---------------+-------------------------------------------------------+
  | [INNER] JOIN  | Kombin. Datens. aus 1. Tab. mit PASSENDEN aus 2. Tab. bzgl.|
  |               | einer/mehrerer Spalten. Datensätze beider Tab. weglassen,  |
  |               | die in Kombinationsspalte(n) keinen passend. Wert enthalten|
  +---------------+-------------------------------------------------------+
  | {LEFT | RIGHT}| Analog [INNER] JOIN, aber JEDER Datensatz aus LEFT=linker   |
  |   [OUTER] JOIN| bzw. RIGHT=rechter Tabelle ist im Ergebnis vorhanden       |
  |               | (Spaltenwerte der anderen Tabelle evtl. alle NULL)         |
  +---------------+---------+---------------------------------------------+
  | FULL [OUTER]  | Kombination aus LEFT + RIGHT JOIN, d.h. JEDER Datensatz der|
  |   JOIN        | linken + rechten Tabelle ist mind. 1x im Ergebnis vorhanden|
```

```
  |               | (erst ab MY!5.1 verfügbar!)                                  |
  +---------------+--------------------------------------------------------------+
  | STRAIGHT_JOIN | MySQL-Optimierung ignorieren und Daten benutzergesteuert in|
  |               | Reihenfolge der JOINs in FROM-Clause zusammenfügen (MY!)   |
  |               | (linke Tabelle immer zuerst VOR rechter Tab. gelesen),     |
  |               | entspricht JOIN (d.h "Kreuzprodukt").                        |
  +---------------+---------+----------------------------------------------------+
  | NATURAL [{LEFT | RIGHT} | Automatisch verknüpfen über ALLE gemeinsamen     |
  |   [OUTER]] JOIN         | Spaltennamen (USING ... unnötig)                 |
  +-------------------------+--------------------------------------------------+

Hinweise:
  * MySQL-Optimizer macht immer LEFT JOIN von links nach rechts
    (d.h. sortiert Tabellen in Zugriffsreihenfolge von links nach rechts)
  * Schlüsselworte CROSS, INNER und OUTER dürfen weggelassen werden
    (aus Gründen der klareren Ausdrucks besser verwenden!)
  * RIGHT-Join wird in LEFT-Join umgewandelt durch Vertauschen der Seiten (MY!)
  * RIGHT-Join aus Portabilitätsgründen besser als LEFT-Join formulieren (MY!)

Weitere spezielle Arten von Joins:

  +------------+-------------------------------------------------------------+
  | Begriff    | Bedeutung                                                   |
  +------------+-------------------------------------------------------------+
  | Self-Join  | Verknüpfung einer Tabelle mit sich selbst (rekursiv)        |
  | Equi-Join  | Verknüpfung über "="-Relation                               |
  | Theta-Join | Verknüpfung über andere Rel. als "=" (!=  <>  <  <=  >  >=) |
  | Semi-Join  | Analog Natural Join, dann Reduktion auf Spalten der 1. Tab. |
  +------------+-------------------------------------------------------------+

Syntax:
  <TblReferences> =
      <TblRef> [, <TblRef>] ...

  <TblRef> = <TblFactor> | <JoinTable>

  <TblFactor> =
      <Tbl> [[AS] <Alias>] [<IdxHint>]]
    | <TblSubquery> [AS] <Alias>
    | (<TblRef>)
    | {OJ <TblRef> LEFT OUTER JOIN <TblRef>    # ODBC-Syntax, {...} hinschreiben
         ON <Cond>}

  <JoinTable> = <TblRef> [INNER | CROSS] JOIN                  <TblFactor> [<JoinCond>]
              | <TblRef> STRAIGHT_JOIN                         <TblFactor> [ON <Cond>]
              | <TblRef> {LEFT | RIGHT} [OUTER] JOIN           <TblRef>    <JoinCond>
              | <TblRef> NATURAL [{LEFT | RIGHT} [OUTER]] JOIN <TblFactor>

  <JoinCond> = ON <Cond>
             | USING (<Col>, ...)

  <IdxHint> = USE    {INDEX | KEY} [FOR JOIN] (<Idx1>, ...)
            | FORCE  {INDEX | KEY} [FOR JOIN] (<Idx1>, ...)
            | IGNORE {INDEX | KEY} [FOR JOIN] (<Idx1>, ...)

Beispiele:

  SELECT a.name, a.preis, h.name     # INNER JOIN
    FROM artikel a INNER JOIN hersteller h
      ON a.name = h.name             # Möglich: Spalten verschiedennamig
        WHERE a.preis > 200;

  SELECT a.name, h.name              # LEFT OUTER JOIN
    FROM artikel a LEFT OUTER JOIN hersteller h
      USING (name);                  # Notwendig: Spalten gleichnamig
```

```
  SELECT *                          # NATURAL JOIN
    FROM pers NATURAL JOIN age;     # Notwendig: Spalten gleichnamig
```

12) Mengen-Operationen

Mengen-Operationen erlauben die Verknüpfung von zwei oder mehr Selektionen (Tabellen) zu einer Gesamtselektion:

Mengenoperation	Bedeutung
UNION [DISTINCT]	Vereinigung (STD: doppelte Datensätze weglassen)
UNION ALL	Vereinigung (inkl. doppelte Datensätze)
INTERSECT	Durchschnitt = gemeinsame Datensätze (NICHT in MY!)
EXCEPT/MINUS	Differenz = 1. minus 2. Selektion (NICHT in MY!)

Eigenschaften:
- Spaltenanzahl der verknüpften Selektionen muss gleich sein
- Datentypen der Spalten müssen positionsweise kompatibel sein
  - Zeichenkette
  - Zahl
  - Datum
  - Zeit
  - ...
- Spaltennamen des Ergebnisses = Spaltennamen der ersten Selektion
- Doppelte Datensätze werden standardmäßig weggelassen (Menge!), außer UNION ALL wird verwendet (STD: DISTINCT)
- Sortierung der Daten wird zerstört (möglich: Gesamtergebnis sortieren)

Beispiel:

```
  SELECT name, vorname FROM pers        # Selektion 1
    UNION                               # STD: DISTINCT = doppelte weglassen
  SELECT name, vorname FROM copy;       # Selektion 2
                                        #
  SELECT name, vorname FROM pers        # Selektion 1
    UNION ALL                           # Doppelte beibehalten
  SELECT name, vorname FROM copy;       # Selektion 2
```

13) Unterabfragen (Subqueries/Subselect)

Eigenschaften:
- SELECT-Statement eingebettet ("nested") in anderem Statement
  - "Outer query", "Outer Statement"
  - "Inner query", "Subquery"
- Seit MY!4.01 alles gemäß SQL-Standard erlaubt + einige Erweiterungen
- IMMER in Klammern (...) zu setzen!
- Verschachtelungstiefe beliebig
- Ganz außen muss SELECT, INSERT, UPDATE, DELETE stehen
- Tabelle kann nicht gleichzeitig modifiziert und gelesen werden
- Als Ersatz für JOIN und UNION einsetzbar
- Vorsicht bei NULL-Werten und leeren Tabellen!

Syntax:
```
  #  ---- Outer Query ----            --- Inner Query ---
  SELECT * FROM t1   WHERE col1    = (SELECT col1    FROM t2);
  SELECT * FROM t1 a WHERE a.col1  = (SELECT b.col1  FROM t2 b);
  SELECT * FROM t1   WHERE t1.col1 = (SELECT t2.col1 FROM t2);
  #  -- Outer Statement --            ----- Subquery ----
```

Ergebnis einer Subquery kann sein:

```
  * EIN Skalar (einzelner Wert):
      =  (<SubQuery>)
      <> (<SubQuery>)
      >  (<SubQuery>)
      <  (<SubQuery>)
      >= (<SubQuery>)
      <= (<SubQuery>)
  * EIN Datensatz (ROW-Subquery mit "Row Constructor"):
      ("wert1", "wert2", ...)    = (<SubQuery>)
      ROW("wert1", "wert2", ...) = (<SubQuery>)
      EXISTS (<SubQuery>)              # TRUE, wenn mind. 1 Datensatz zurück
      NOT EXISTS (<SubQuery>)          # TRUE, wenn kein Datensatz zurück
  * EINE Datenspalte (Synonym für ALL ist SOME):
      ... =  ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> gleich
      ... <> ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> ungleich
      ... >  ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> kleiner
      ... <  ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> größer
      ... <= ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> kleiner gleich
      ... >= ANY (<SubQuery>)          # Mind. EIN Wert aus <SubQuery> größer gleich
      ... IN     (<SubQuery>)          # Entspricht "= ANY"
      ... =  ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> gleich
      ... <> ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> ungleich
      ... >  ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> größer
      ... <  ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> kleiner
      ... <= ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> kleiner gleich
      ... >= ALL (<SubQuery>)          # ALLE Werte aus <SubQuery> größer gleich
      ... NOT IN (<SubQuery>)          # Entspricht "<> ALL"
  * EINE Tabelle (mehrere Datensätze mit mehreren Datenspalten)
      ... JOIN... (<SubQuery>) AS ...  # Verbundanweisung (beliebigen Typs)

Row Constructor:

  ("tom", 2)    = (SELECT sp1, sp2 FROM t3 WHERE ...)    # Syntax 1
  ROW("tom", 2) = (SELECT sp1, sp2 FROM t3 WHERE ...)    # Syntax 2

Fast identisch (bzw. ganz identisch bei UNIQUE Index Spalte in WHERE ...) mit:

    "tom" = (SELECT sp1 FROM t3 WHERE ...) AND
        2 = (SELECT sp2 FROM t3 WHERE ...)

Beispiel:

  SELECT name, geburtsdatum
    FROM pers, age
      WHERE geburtsdatum = (SELECT MAX(geburtsdatum) FROM age)
        AND pers.nr = age.nr;

  SELECT name, geburtsdatum
    FROM pers, age
      WHERE geburtsdatum >= ALL (SELECT geburtsdatum FROM age)
        AND pers.nr = age.nr;

  SELECT name, preis
    FROM artikel
      WHERE preis = (SELECT MIN(preis) FROM artikel)
         OR preis = (SELECT MAX(preis) FROM artikel);

  SELECT name, preis
    FROM artikel
      WHERE preis <= ANY (SELECT preis FROM artikel);

  DELETE FROM t1
  WHERE s11 > ANY
    (SELECT COUNT(*) FROM t2                  # Subquery 1
     WHERE NOT EXISTS
       (SELECT * FROM t3                      # Subquery 2
```

```
        WHERE ROW(5 * t2.s1, 77) =
          (SELECT 50, 11 * s1 FROM t4         # Subquery 3
           UNION
           SELECT 50, 77 FROM                 # Subquery 4
             (SELECT * FROM t5) AS t5)));     # Subquery 5
```

14) Transaktionen

Ein DBMS soll die Datenkonsistenz sichern bei:
- Gleichzeitigem (Schreib)Zugriff mehrerer Benutzer ("Concurrency")
- Server-Absturz (Stromausfall)
- Hardwaredefekten
- Programmierfehlern
- Netzwerkfehlern (Verbindungsabbruch)
- Zugriffsrechte-Problemen

Transaktion (TA) = Gruppe von zusammengehörenden SQL-Anweisungen
- A) Entweder gemeinsam VOLLSTÄNDIG ausgeführt
- B) Oder GAR NICHT ausgeführt (in Ausgangszustand zurückversetzt)
  - + Bei Widerspruch
  - + Bei Fehler
  - + Bei Zugriffsverletzung

  Nachteil: Längere Ausführungszeit
  Gegenseitige Behinderung
  Deadlocks (Über-Kreuz-Exklusiv-Zugriffe) möglich

ACID-Eigenschaften einer Transaktion TA:

Eigenschaft	Beschreibung
A)tomicity	TA entweder vollständig oder gar nicht durchgeführt
C)onsistency	DB vor+nach TA konsistent (nicht unbedingt während!)
I)solation	Gleichzeitig ablaufende TAs beeinflussen sich nicht
D)urability	Ergebnis einer erfolgreichen TA steht dauerhaft in DB

Transaktionen über mehrere SQL-Anweisungen:
- NUR mit Engines "InnoDB" + "BDB" (MY!)
- Bei allen anderen Engines ist jede einzelne SQL-Anweisung eine TA (sofern der MySQL-Server nicht abstürzt)
- Mit "MyISAM" nur Locken kompletter Tabellen möglich (LOCK/UNLOCK)
- Engine pro Tabelle wählbar --> Optimale Kombination auswählen
- TA-Locking-Verhalten
  - + InnoDB: Row-Level
  - + BDB: Page-Level (evtl. mehrere "benachbarte" Rows auch gesperrt)

Standardmäßig ist bei MySQL "Autocommit"-Modus eingeschaltet (MY!):
- D.h. JEDE Anweisung für sich stellt Transaktion dar
- Startet eine Transaktion mit "BEGIN", wird Autocommit-Modus abgeschalten (d.h. COMMIT/ROLLBACK zum Abschluss/Abbruch der Transaktion notwendig)
- Deaktivieren sorgt dafür, dass "BEGIN" nicht mehr notwendig ist und alles bis zum nächsten "COMMIT/ROLLBACK" automatisch eine Transaktion darstellt
- Ändern des Autocommit-Modus durch:

```
    SET AUTOCOMMIT = 1;   # Jede Aktion ist TA, BEGIN startet TA bis COMMIT
    SET AUTOCOMMIT = 0;   # Ständig TA, COMMIT/ROLLBACK beendet+startet neue
```

Transaktionen manuell einleiten und mit COMMIT abschließen bzw. mit ROLLBACK abbrechen (AUTOCOMMIT von BEGIN automatisch deaktiviert):

```
START TRANSACTION;                  # TA beginnen (auch BEGIN [WORK])
...                                 #
SAVEPOINT <Name>;                   # Marke <Name> setzen (Label)
...                                 #
```

```
  RELEASE SAVEPOINT <Name>;              # Savepoint <Name> freigeben
  ...                                    #
  ROLLBACK [WORK] TO SAVEPOINT <Name>;   # Bis Marke <Name> zurücknehmen
  ...                                    #
  COMMIT [WORK];                         # TA abschließen (ALLES!)
  ROLLBACK [WORK];                       # TA abbrechen (ALLES!)
```

Folgende Anweisungen (meist DDL = Data Definition Language) beenden Transaktion ebenfalls (neben COMMIT/ROLLBACK), d.h. führen automatisch "COMMIT" durch:

CREATE DATABASE ...	Datenbank anlegen
DROP DATABASE ...	Datenbank löschen (inkl. Tabellen)
CREATE TABLE ...	Tabelle anlegen
DROP TABLE ...	Tabelle löschen (inkl. Struktur)
TRUNCATE TABLE ...	Tabellendaten löschen (da DROP + CREATE TABLE) (DELETE FROM <Tbl>; ohne WHERE beendet TA nicht!)
ALTER TABLE ...	Tabellen-Struktur ändern
RENAME TABLE ...	Tabelle umbenennen
CREATE INDEX ...	Index anlegen
DROP INDEX ...	Index löschen
LOCK TABLES ...	Tabellen sperren (bis UNLOCK TABLES)
BEGIN ...	TA sind nicht verschachtelbar!
FLUSH LOGS ...	Alle/einige Caches auf Platte schreiben

Hinweise:
- Transaktionen sind nicht verschachtelbar (MY!)
- Statt "BEGIN [WORK]" auch "START TRANSACTION" möglich
  (in Prozeduren verwenden, da "BEGIN" dort einen Block einleitet!)
- Tritt KEIN EINZIGER Fehler auf, werden ALLE Änderungen in TA durchgeführt
- Tritt EIN Fehler auf, werden ALLE bisherigen Änderungen in TA widerrufen
  (und die noch folgenden Anweisungen nicht mehr durchgeführt)
- Bei Anwendung einer Transaktion auf mind. 1 transaktions-UN-sichere Tabelle
  wird jede Änderung sofort durchgeführt (MyISAM, entspricht AUTOCOMMIT=1)
- "ROLLBACK" auf nicht transaktionssicherer Tabelle führt zu Fehlermeldung
- Protokolldateien
  + ASCII: Zeichnet "ROLLBACK" auf
  + Binär: Zeichnet "ROLLBACK" NICHT auf

Transaktions-Level (Isolations-Ebene) einstellen (STD: 2 = REPEATABLE READ):
- SESSION = Für AKTUELLE Sitzung
  GLOBAL = Für ALLE danach neu gestartete Sitzungen (nicht für bestehende!)
  Sonst   = Für NÄCHSTE Transaktion (1x!)
- Beim Start des MySQL-Servers festlegen durch:
  ```
  --transaction-isolation = <Level>       # STD: 2
  ```

```
SET [SESSION | GLOBAL] TRANSACTION ISOLATION LEVEL <Level>;
```

			Fehlermöglichkeit				
Level	Nr	Datensatz-sperre	Lost Update	Dirty Read	Nonrepeatable Read	Phantom Read	
--	-	--	ja	ja	ja	ja	
READ UNCOMMITTED	0	--	--	ja	ja	ja	
READ COMMITTED	1	write	--	--	ja	ja	Oracle-STD
REPEATABLE READ	2	read/write	--	--	--	ja	Mysql-STD
SERIALIZABLE	3	read/write	--	--	--	--	

Hinweise:
- Standard von MySQL ist Level 2 (REPEATABLE READ)
- Oracle

```
    + Standard ist Level 1 (READ COMMITTED)
    + Kennt nicht Level 0 (READ UNCOMMITTED) und Level 2 (REPEATABLE READ)
    + Alternative Level-Namen bei Oracle
        READ UNCOMMITTED = SNAPSHOT
        REPEATABLE READ  = SNAPSHOT TABLE STABILITY
    + Weitere Angaben nach TRANSACTION möglich:
        READ WRITE | READ ONLY   # Daten ändern (STD) / nur lesen
        WAIT | NO WAIT           # Auf Abschluss anderer TA mit Zugriff auf gl.
                                 # Tab. warten (STD), sonst TA sofort abbrechen
```

Beispiel:

```
  @ueberweisung = 1000;
  SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
  BEGIN;
    UPDATE konto SET betrag = betrag - @ueberweisung WHERE nr = 123456789;
    UPDATE konto SET betrag = betrag + @ueberweisung WHERE nr = 987654321;
  COMMIT;
```

15) Locking

Tabellen/View-Lock:

* Tabellen/Views für exklusiven Zugriff sperren + wieder freigeben
* Differenzierung: Schreib-Lock oder Lese/Schreib-Lock
* Alle angegebenen Tabellen werden vollständig gesperrt
* Funktioniert bei jeder Engine!
* WRITE hat höhere Priorität als READ

```
  LOCK TABLES                          # Sperren mehrerer Tabellen gleichzeitig:
    table1              WRITE,         # Lese+Schreibschutz für alle anderen
    table1 AS alias1    READ,          # Schreibschutz (Lesen für alle erlaubt)
    table2 AS alias2    READ LOCAL,    # INSERT zulassen solange kein Konflikt
    table3 LOW_PRIORITY WRITE;         # Erst sperren wenn kein READ-Lock (warten)
  ...                                  # Operationen auf gelockten Tabellen ...
  UNLOCK TABLES;                       # Freigeben aller Locks
```

Lock-Typ	Bedeutung
WRITE	Lese- und Schreibschutz für alle außer Anforderer
READ	Schreibschutz für alle außer Anf. (Lesen für alle OK)
READ LOCAL	Schreibschutz, aber INSERT OK solange kein Konflikt
LOW_PRIORITY WRITE	Erst sperren wenn kein READ-Lock (warten)

Tabellen gegen Zugriffe durch andere Threads sperren
(alle benötigten Tabellen gleichzeitig!):

* Wartet, bis alle anderen Zugriffe auf die Tabellen beendet sind
  LOCK TABLES <Tbl> <Typ>, ...;
* <Typ>

```
    IN SHARE MODE              #
    IN SHARE MODE NOWAIT       #
    IN EXCLUSIVE MODE          #
    IN EXCLUSIVE MODE NOWAIT   #
```

Beispiel:

```
  LOCK TABLES pers WRITE;
  ALTER TABLE pers DISABLE KEYS;
  INSERT INTO pers (nr, vorname, name) VALUES
    (1, "Thomas",  "Birnthaler"),
    (2, "Markus",  "Mueller"),
    (8, "Andrea",  "Bayer");
  ALTER TABLE pers ENABLE KEYS;
```

```
  UNLOCK TABLES;

Hinweise:
  * LOCK/UNLOCK beziehen sich auf aktuelle Sitzung
  * LOCK führt implizites COMMIT durch
  * LOCK (und Transaktions-Beginn) führt implizites UNLOCK durch
  * Wartet, bis ALLE Tabellen gemeinsam gelockt werden können
  * Deadlock-frei (Reihenfolge der Locks durch Datenbank gewählt)
  * Lock auf temp. Tabelle erlaubt aber ignoriert (sowieso sitzungsbezogen)
  * LOCK auf View lockt alle ihre Basis-Tabellen!
  * Nach LOCK nur auf gelockte Tabellen zugreifbar (auf andere nicht)!
  * Mehrfachzugriff per Tabellen-Alias braucht Mehrfachlock mit diesen Aliasen!
  * Transaktion-Simulation für MyISAM (und andere nicht TA-fähige Engines)
  * Tabelle löschen nach LOCK möglich, aber nicht Tabelle anlegen oder TRUNCATE

Advisory Lock (anwendungsbezogen):
  * Frei definierbare Locks (per Name)
  * Anwendungen müssen Lock selbst anfordern
  * Anwendungen, die sich nicht daran beteiligen, werden nicht mit einbezogen
    (können machen was sie wollen)
  * Deadlock möglich (DB kümmert sich nicht um ihre Auflösung)

  +-------------------------+-------------------------------------------------+
  | Funktion                | Bedeutung                                       |
  +-------------------------+-------------------------------------------------+
  | GET_LOCK(<Name>, <sec>) | Lock <Name> beantragen (Timeout nach <Sec> Sek.|
  |                         | (Erg: 1=erhalten, 0=nicht erh., NULL=Fehler)    |
  | IS_FREE_LOCK(<Name>)    | Lock <Name> frei? (Erg: 1=ja, 0=nein)           |
  | IS_USED_LOCK(<Name>)    | Lock <Name> belegt? (Erg: 1=ja, 0=nein)         |
  | RELEASE_LOCK(<Name>)    | Lock <Name> freigeben                           |
  +-------------------------+-------------------------------------------------+


16) Views (Sichten)
-------------------

Views (Sichten) sind vordefinierte gespeicherte benannte Abfragen (SELECT-
Anweisungen), sie werden auch "virtuelle" Tabellen (derived table) genannt.

Zweck:
  * Tabellen und Spalten umbenennen (um Änderungen nach Außen zu verbergen)
  * Zusätzliche berechnete Spalten einführen
  * Zugriffsbeschränkung (per Sichtbarkeit von Spalten und Datenzeilen)
  * Zugriff vereinfachen (Verknüpfung mehrerer Tab. sieht wie eine Tab. aus)
  * Verbergen konkrete Tabellenstruktur (änderungsfreundlich)
  * Schrittweises "Refactoring" eines Schemas (alte Tab. wg. Alt-SW beibehalten)
    + Migration/Umprogrammierung von Applikationen ermöglichen

Eigenschaften:
  * Ableitung aus einer oder mehreren Basis-Tabellen oder anderen (Unter-)Views
    auf der Basis von Joins, Unions und Unterabfragen
    + Spalten oder Ausdrücke mit Funktionen, Konstanten, Spalten, Operatoren...
    + ORDER BY ist möglich (aber evtl. bei weiterem ORDER BY ignoriert)
  * Verwendbar wie eine normale Tabelle (Pseudotabelle)
    + Tabellen und Views teilen sich selben Namensraum in einer Datenbank
  * Spaltennamen <Col>
    + Weglassen --> Spaltennamen automatisch durch SELECT gebildet
    + Angeben   --> Anzahl Spalten von View und SELECT MUSS gleich sein!
  * Über Views möglich:
    + Daten abfragen: Immer
    + Daten ändern:   Unter bestimmten Bedingungen
    + Daten einfügen: Unter bestimmten Bedingungen
    + Daten löschen:  Unter bestimmten Bedingungen
  * View wird zum Definitionszeitpunkt "eingefrohren"
    + "Updatability Flag" wird erstellt (UPDATE/INSERT prinzipiell möglich)
    + Änderungen an Unter-Tabellen oder -Views wirken sich nicht aus
```

```
    + Löschen von Unter-Tab/Views --> Fehlermeldung erst bei View-Verwendung
  * Beschränkung der Einfüge-Daten auf Erfüllung der View-WHERE-Klausel möglich
    (WITH CHECK OPTION --> Prüft, ob Daten mit WHERE-Bed. der View ausgefiltert)

Beschränkungen:
  * Nur dann aktualisierbar ("Updatable View", d.h. INSERT/UPDATE/DELETE), wenn
    1:1-Beziehung zwischen Datensätzen in View und Basis-Tabellen vorhanden
    + Nicht möglich bei Verwendung von Aggregat-Funktionen SUM, MIN, MAX, COUNT,
      sowie DISTINCT, GROUP BY, HAVING, UNION, UNION ALL, Subqueries, ...
    + Per Ausdruck abgeleitete View-Spalten sind nicht updatebar
    + Alle zu ändernden Spalten müssen in EINER Basis-Tabelle liegen
    + Nicht möglich bei Realisierung der View per temp. Tabelle (TEMPTABLE)
  * Neue Sätze einfügbar ("Insertable", d.h. INSERT) wenn (zusätzlich zu oben)
    + Kein Basis-Spaltenname doppelt verwendet
    + Alle Spalten der Basis-Tabelle ohne Defaultwert in View enthalten
    + Keine per Ausdruck abgeleiteten View-Spalten
    + Nur EINE Basis-Tabelle ist betroffen
  * Kein Index auf Views möglich
    + Ausnutzung von Indices bei MERGE möglich, bei TEMPTABLE nicht
  * Keine temporäre Tabelle als Basis möglich
  * Kein Trigger mit View assozierbar (manche DB erlauben "INSTEAD OF" Trigger)
  * Ableitungs-Algorithmus MERGE/TEMPTABLE ist Eigenschaft der View-Definition
    und unabhängig von der darauf auszuführenden SQL-Anweisung (MY!)
  + Führen evtl. Entwickler in die Irre (sieht einfach aus obwohl sehr komplex)
  + Eigene Optimierung (getrennt von Tabellen) --> Nicht so gut ausgetestet
  + Schlecht editierbar da Originaltext verloren geht (MY!)
    --> S281 (Roland Bouman)
  * In Liste der Tabellen mit enthalten --> Mit Präfix "v_" und "t_" unterscheidbar
machen

"Materialized" View (nicht in MY!):
  + Unsichtbare Tabelle die periodisch upgedated wird
  * Kostet Speicherplatz
  * Wird nicht bei jeder Veränderung der Basis-Tabellen aktualisiert
  * Verhält sich wie statische Tabelle --> Schneller als normale View
  * Für häufige View-Abfragen, deren Basis-Tabellen sich selten ändern
  * Simulierbar per EVENT/TRIGGER, der View asynchron/synchron als echte Tabelle
erstellt (MY!)
  * "Permanente temporäre" Tabelle

Definition von Views:

  CREATE [OR REPLACE]                                 # REPLACE=View ersetzen
    [ALGORITHM = (MERGE | TEMPTABLE | UNDEFINED)]     # (STD: UNDEFINED, MY!)
    [DEFINER = {<User> | CURRENT_USER}]               #
    [SQL SECURITY {DEFINER | INVOKER}]                #
      VIEW <View> [(<Col>, ...)]                      # Name + opt. Spaltennamen
        AS <Select>                                   # Beliebige SELECT-Anweisung
          [WITH [CASCADED | LOCAL] CHECK OPTION];     # Erfüllt INSERT WHERE-Bed.?

Konstruktions-Algorithmus (ALGORITHM = ...):
  * MERGE:     View-Definition textuell in View-verwendende Anweisung einsetzen
               ("Query Rewrite", sogar für verschachtelte Views möglich)
               + Effizienter
               + Updatebar
               + Locks länger gehalten
               + Nicht verwendbar bei Aggregat-Funktionen SUM, MIN, MAX, COUNT
                 DISTINCT, GROUP BY, HAVING, LIMIT, UNION, UNION ALL, Subqueries
                 (allgemein falls 1:1-Beziehung zwischen Zeilen in
                  Basis-Tabellen und Ergebnis-Tabelle verloren geht)
  * TEMPTABLE: Temporäre Tabelle für View-Ergebnis erzeugen (jedesmal!)
               + Weniger Effizient
               + Nicht updatebar
               + Locks kürzer
  * UNDEFINED: MERGE (vorzugsweise) oder TEMPTABLE autom. aussuchen (MY!)
```

```
Prüfung ob INSERT die View-WHERE-Bedingung erfüllt (WITH ... CHECK OPTION):
  * CASCADED: Definierte View und alle Unter-Views prüfen (STD)
  * LOCAL:    Nur definierte View prüfen, nicht Unter-Views

Alle Views auflisten (zusammen mit Tabellen, gleicher Namespace):

  SHOW TABLES;

View-Definition anzeigen:

  SHOW CREATE VIEW <View>;

View ändern (Charakteristika und Inhalt, nicht Name):

  ALTER VIEW <View> [(<Col>, ...)]
    [ALGORITHM = (MERGE | TEMPTABLE | UNDEFINED)]
      AS <Select>
        [WITH [CASCADED | LOCAL] CHECK OPTION];

View(s) umbenennen (aber nicht in andere Datenbank verschieben!,
stellt eine "atomare" Operation dar, auch bei Umbenennung mehrerer Views):

  RENAME TABLE <View> TO <NewView> [, ...];
  ALTER TABLE <View> RENAME TO <NewView>;

View löschen:

  DROP VIEW <View>;
  DROP VIEW IF EXISTS <View>;   # Kein Fehler falls nicht existent

Algorithmus ermitteln den View benutzt:

  EXPLAIN SELECT * FROM <View>;            # "TEMPTABLE" falls "DERIVED" in Ausgabe
  EXPLAIN EXTENDED SELECT * FROM <View>;   # "TEMPTABLE" falls "DERIVED" in Ausgabe

Beispiele für Views:

  #------------------------------------------------------------------------
  # View 1
  #------------------------------------------------------------------------
  DROP TABLE IF EXISTS t1;
  CREATE TABLE t1 (
    anz   INT,
    preis INT
  );

  INSERT INTO t1 VALUES
    (3, 50),
    (7, 80),
    (9, 99),
    (1, 12);

  DROP VIEW IF EXISTS v1;

  CREATE VIEW v1 AS
    SELECT anz, preis, anz * preis AS "wert"
      FROM t1;

  SELECT * FROM v1;

  DROP VIEW v1;
  DROP TABLE t1;

  #------------------------------------------------------------------------
  # View 2
  #------------------------------------------------------------------------
```

```
  DROP TABLE IF EXISTS t2;
  CREATE TABLE t2 (
    nr   INT,
    name CHAR(30)
  );

  INSERT INTO t2 VALUES
    (1, "a"),
    (2, "abcde"),
    (5, "test"),
    (4, "text");

  DROP VIEW IF EXISTS v2;

  CREATE VIEW v2 AS
    SELECT *, "konstant"
      FROM t2 WHERE nr = LENGTH(name);

  SELECT * FROM v2;

  DROP VIEW v2;
  DROP TABLE t2;

  #-----------------------------------------------------------------------------
  # View 3
  #-----------------------------------------------------------------------------
  DROP VIEW IF EXISTS v3;

  CREATE VIEW v3 (Artikel, Hersteller) AS
    SELECT a.name, h.name
      FROM artikel AS a, hersteller AS h
        WHERE a.name = h.name;

  #-----------------------------------------------------------------------------
  # View 4
  #-----------------------------------------------------------------------------
  DROP VIEW IF EXISTS v4;

  CREATE VIEW v4 (Name, Artikel) AS
    SELECT p.name, a.name
      FROM artikel a, bestellung b, pers p
        WHERE a.nr     = b.nr
         AND p.nr      = b.nr
         AND p.name    = "Müller"
         AND p.vorname = "Oskar";

  #-----------------------------------------------------------------------------
  # View 5
  #-----------------------------------------------------------------------------
  DROP VIEW IF EXISTS v5;

  CREATE VIEW v5 AS
    SELECT * FROM artikel       # ALLE Spaltennamen übernommen
      WHERE preis > 200         # Bedingung an Datensätze
        WITH CHECK OPTION;      # Prüfen ob Bed. bei Einf./Ändern erfüllt


17) Variablen
-------------

MySQL kennt folgende Variablen-Arten:

  * @@SERVER: Einstellungen/Attribute MySQL-Server (fix) --> SHOW VARIABLES;
                                                        --> SHOW GLOBAL VARIABLES;
                                                         --> SHOW LOCAL VARIABLES;
  * @SESSION: Sitzungsbezogen (frei wählbar, read-write)
```

```
  * LOKAL:    Stored Procedure/Event/Trigger (frei wählbar, read-write)
  * STATUS:   Messwerte MySQL-Server (fix, read-only) --> SHOW/FLUSH STATUS;

<Var> ist im Folgenden ein Variablen-Bezeichner (Identifier), d.h. er kann aus
den Zeichen A-Z, a-z, 0-9, _ und $ bestehen, führende Ziffern sind nicht
erlaubt --> 2f) Identifier (Bezeichner).

  +---------+-------------------------------------------------------------+
  | Typ     | Bedeutung                                                   |
  +---------+-------------------------------------------------------------+
  | @@<var> | SERVER-Variable (klein geschrieben mit "_")                 |
  |         | + Vom Server festgelegt                                     |
  |         |   - Feste Anzahl (etwa 450 je nach MySQL-Version)           |
  |         |   - FIXE Namen <var>                                        |
  |         |   - Enthalten Einstellungen/Attribute des MySQL-Servers     |
  |         |   - Neue Version --> kennt neue zusätzlich / veraltete nicht mehr|
  |         |   - Server-Neustart --> Neu initialisiert                   |
  |         |   - Lesbar + sehr viele schreibbar (evtl. mit SUPER-Berechtigung)|
  |         | + Startwert                                                 |
  |         |   - Fest eingebaut in "mysqld"-Server                ODER   |
  |         |   - Per Options-Parameter beim Server-Start übergeben ODER  |
  |         |   - Aus Server-Konfig "my.cnf/ini" beim Start               |
  |         | + Viele doppelt vorhanden                                   |
  |         |   - GLOBAL:  Server-weiter (Default)wert                    |
  |         |   - SESSION: Für akt. Session (GLOBAL-Kopie bei Verb.-Aufbau)|
  |         | + Gültigkeitsbereich:                                       |
  |         |   - Einige GLOBAL STATISCH (nur beim Server-Start einstellbar)|
  |         |   - Einige GLOBAL DYNAMISCH (jederzeit mit SUPER-Recht änderbar)|
  |         |   - Viele SESSION-bezogen (jederzeit für akt. Sitzung änderbar)|
  |         | + Session-Wert wird mit globalem Wert initialisiert         |
  |         |   - Session-Wert geändert --> "überdeckt" Globalen Wert     |
  |         |   - Verschwindet am Sitzungsende                            |
  |         | + Auflisten per SHOW [GLOBAL|SESSION] VARIABLES [LIKE "..."]|
  +---------+-------------------------------------------------------------+
  | @<var>  | SESSION-Variable (Sitzungsbezogen):                         |
  |         | + Benutzerdefiniert                                         |
  |         |   - Name <var> frei wählbar                                 |
  |         |   - Beliebig viele                                          |
  |         |   - NICHT ZU DEKLARIEREN (Datentyp beliebig, 1 Wert)        |
  |         |   - Lesbar + schreibbar                                     |
  |         | + Gültigkeitsbereich:                                       |
  |         |   - Auf Sitzung beschränkt                                  |
  |         |     --> Gleichnamige versch. Sitzungen überschneiden sich NICHT|
  |         |   - Verschwinden am Ende der Sitzung (Verbindungsende)      |
  |         | + Nicht per SQL-Anweisung auflistbar                        |
  +---------+-------------------------------------------------------------+
  | <var>   | LOKALE Variable + Aufruf-Parameter in Stored Proc./Event/Trigger|
  |         | + Benutzerdefiniert                                         |
  |         |   - Name <var> frei wählbar                                 |
  |         |   - Beliebig viele                                          |
  |         |   - Vor Verwendung zu DEKLARIEREN (mit SQL-Datentyp)        |
  |         |     (z.B. DECLARE betrag FLOAT DEFAULT 0.0;)                |
  |         |   - Lesbar + schreibbar                                     |
  |         | + Gültigkeitsbereich:                                       |
  |         |   - Auf Stored Procedure/Event/Trigger/... beschränkt       |
  |         |   - Verschwinden am Ende von Stored Procedure/Event/Trigger/...|
  |         | + Nicht per SQL-Anweisung auflistbar                        |
  +---------+-------------------------------------------------------------+
  | <Var>   | STATUS-Variable (großer Anfangsbuchstabe, Rest klein mit "_")|
  |         | + Vom Server festgelegt                                     |
  |         |   - Feste Anzahl (etwa 350 je nach MySQL-Version)           |
  |         |   - FIXE Namen <Var>                                        |
  |         |   - Enthalten Messwerte des MySQL-Servers                   |
  |         |   - Neue Version --> kennt neue zusätzlich / veraltete nicht mehr|
  |         |   - Server-Neustart --> Neu initialisiert                   |
  |         |   - Nur lesbar                                              |
```

```
  |         | + Gültigkeitsbereich:                                          |
  |         |   - Serverweit gültig                                          |
  |         |   - Mit FLUSH STATUS auf 0-Wert zurücksetzen (SUPER-Berechtigung)|
  |         | + Auflisten per SHOW STATUS [LIKE "..."]                       |
  +---------+----------------------------------------------------------------+

Eigenschaften:
  * Variablenname darf kein Objektname und SQL-Schlüsselwort sein
    (außer er wird durch `...` eingerahmt --> 2f) Identifier (Bezeichner))
  * Bis MY!4.1 wird GROSS/kleinschreibung unterschieden, ab MY!5.0 nicht mehr
  * Änderung von Globalen Server-Variablen --> SUPER-Recht notwendig

Session-Variable @v einen Wert zuweisen (Variable dabei evtl. neu angelegt):

  SET @v = 3;                         # Schlecht, besser ":=" verwenden!
  SET @v := 3;                        # Besser als Verwendung von "="!
  @v = 3;                             # Fehler (SET davor notwendig)
  @v := 3;                            # Fehler (SET davor notwendig)
  SELECT @v := 3;                     # Zuweisung eines SELECT-Ergebnisses
  SELECT @v := COUNT(*) FROM pers;    #   (analog)
  SELECT COUNT(*) FROM pers INTO @v;  #   (analog)
  SELECT name, vorname FROM pers      # Mehrere Variablen gleichzeitig
    WHERE nr = 1 INTO @v1, @v2;       #   MY!5.0

ACHTUNG: Folgende Anweisung ist KEINE Zuweisung sondern ein VERGLEICH der
Session-Variablen @v mit dem Ergebnis von "SELECT COUNT(*) FROM pers". Gibt je
nach Wert von @v und dem SELECT-Ergebnis TRUE oder FALSE zurück (Wert von @v
bleibt unverändert):

  SELECT @v = COUNT(*) FROM pers;     # Fehler: Vergleich (statt Zuweisung)
  SELECT @v := COUNT(*) FROM pers;    # OK: Zuweisung

Server-Variable (Session oder Global) auflisten:

  SHOW SESSION VARIABLES;             # ALLE sessionbezogenen Server-Var.
  SHOW LOCAL   VARIABLES;             # ALLE sessionbezogenen Server-Var.
  SHOW GLOBAL  VARIABLES;             # ALLE globalen Server-Var.
  SHOW         VARIABLES;             # ALLE Server-Var. (Session, sonst Global)
  SHOW SESSION VARIABLES LIKE "b%";   # PASSENDE Server-Var. (Session)
  SHOW LOCAL   VARIABLES LIKE "b%";   # PASSENDE Server-Var. (Session)
  SHOW GLOBAL  VARIABLES LIKE "b%";   # PASSENDE Server-Var. (Global)
  SHOW         VARIABLES LIKE "b%";   # PASSENDE Server-Var. (Sess., sonst Global)

Status-Variablen auflisten:

  SHOW STATUS;                        # Alle Status-Var. (Server)
  SHOW STATUS LIKE "b%";              # Einige Status-Var. (Server)

Variablen-Wert anzeigen:

  SELECT wait_timeout;                # Lokale Var. in Stored Procedure/Event
  SELECT @wait_timeout;               # Session
  SELECT @@SESSION.wait_timeout;      # Server-Session
  SELECT @@GLOBAL.wait_timeout;       # Server-Global
  SELECT @@wait_timeout;              # Server-Session ODER -global
                                      # (Session "überdeckt" Global)

Einige Server-Variable sind nur Global verfügbar:

  SELECT @@SESSION.binlog_cache_size; # Fehler da nur Global!
  SELECT @@GLOBAL.binlog_cache_size;  # Global
  SELECT @@binlog_cache_size;         # Automatisch Global

Belegung von Variablen mit einem Wert:

  SET wait_timeout          = 900;    # 0) Lokal (nur in Stored Procedure)
```

```
  SET @wait_timeout          = 901;   # 1) Session
  SET @@SESSION.wait_timeout = 902;   # 2a) Server-Session
  SET SESSION wait_timeout   = 903;   # 2b) Server-Session, überschreibt 2a)
  SET @@GLOBAL.wait_timeout  = 904;   # 3a) Server-Global
  SET GLOBAL wait_timeout    = 905;   # 3b) Server-Global, überschreibt 3a)

Erneut Variablen-Wert anzeigen:

  SELECT wait_timeout;               # 0) Stored Procedure  --> 900
  SELECT @wait_timeout;              # 1) Session           --> 901
  SELECT @@SESSION.wait_timeout;     # 2) Server-Session    --> 903
  SELECT @@GLOBAL.wait_timeout;      # 3) Server-Global     --> 905
  SELECT @@wait_timeout;             # 4) Server-Session ODER -global --> 903
                                     #    (Session "überdeckt" Global)


18) Prepared Statements (Vorbereitete Anweisungen)
--------------------------------------------------

Prepared Statements sind vorbereitete SQL-Anweisungen ("Template/Skeleton") mit
PLATZHALTERN der Form "?" (oder ":name", die erst beim Aufruf mit WERTEN (aus
Variablen) gefüllt werden. Ein Prepared Statement ist entweder
  * ein String-Literal "..." oder
  * eine Text-Variable mit einer SQL-Anweisung als Inhalt

Eigenschaften:
  * Nur lokal pro Sitzung existent (liegen nicht auf Platte!)
    --> Gleichnamige verschiedener Sitzungen überschneiden sich nicht!
  * Prepared Statement --> Statement-Handle <PrepStmt>
  * Hauptzweck
    + In Stored Procedures einsetzbar ("dynamic SQL")
    + Einschränkungen:
      - Datenbank-, Tabellen- und Spalten-Namen sind Identifier
        --> Nicht per PLATZHALTER parametrisierbar
      - LIMIT und OFFSET parametrisierbar seit MY!5.6.x
  * Geschwindigkeitsgewinn
    + SQL-Anweisung nur 1x zu kompilieren
    + Optimierung nur 1x durchzuführen
    + Daten binär statt ASCII-Text (DATE 10 --> 3, BLOB, TEXT, ..)
    + Bei jeder Ausführung nur Parameter für Platzhalter senden
  * Sicherheitsgewinn
    + Escape/Quote unnötig
    + "SQL-Injection" unmöglich (Benutzer-Eingabe kann SQL-Anweisung nicht manip)
    + REGEL: "Never trust incoming Data!"
    + Reguläre Ausdrücke zum Prüfen der Benutzer-Daten bieten sich an!
  * Erneut Prepared Statement mit gleichem Namen --> Altes wird überschrieben
  * Binäres Protokoll nur von API genutzt, nicht vom SQL-Interface

Drei prinzipielle Typen:
  * Client-side: Client schickt Endergebnis der Ersetzung (Emulation)
  * Server-side: Statement Identifier, Binäres Protokoll, Parameter (API)
  * SQL-Interface: Trennung PREPARE + EXECUTE, Textprotokoll, SQL-Variablen

Optimierung mehrstufig:
  * Bei Definition:
    + SQL-Text parsen
    + Negationen entfernen
    + Subqueries umschreiben
  * Erste Ausführung:
    + Nested Joins vereinfachen
    + OUTER JOIN --> INNER JOIN falls möglich
  * Jede Ausführung:
    + Partitionen weglassen
    + COUNT(), MIN(), MAX() weglassen wo möglich
    + Konstante Subexpression weglassen
    + Konstante Tabellen erkennen
```

```
    + Identitäten verteilen
    + Zugriffsmethoden "ref", "range" und "index_merge" analysieren + optimieren
    + Join Reihenfolge optimieren

Beschränkungen:
  * Nur lokal pro Sitzung existent (liegen nicht auf Platte!)
    + Statement-Handle <PrepStmt> nur darin verwendbar
  * In Prozeduren verwendbar, aber NICHT in Funktionen, Triggern und Events
  * Nicht im Query-Cache (vor MY!5.1, später schon)
  * Platzhalter nur für Datenwerte, NICHT aber für SQL-Schlüsselworte und
    für Datenbank-, Tabellen- und Spalten-Namen einsetzbar
    + LIMIT/OFFSET-Werte möglich seit MY!5.6.X
  * Nur Variable in Platzhalter einsetzbar, keine Ausdrücke oder Literale
    + Anzahl Variable muss gleich Anzahl Platzhalter sein
  * Nur eine SQL-Anweisung pro Prepared Statement erlaubt (kein ";")
  * Nicht verschachtelbar
  * Einmaliger Einsatz langsamer als äquiv. normale SQL-Anweisung auszuführen
  * "Memory Leak" möglich, falls Freigabe vergessen wird
  * Maximal "max_prepared_stmt_count" pro Server möglich (Wert 0 --> keine)

Prepared Statement mit Platzhaltern (ergibt Statement-Handle <PrepStmt>):

  PREPARE <PrepStmt>      # Beim Ausführen ausgefüllt, Anführungszeichen nötig,
    FROM <String>;        # "?" als Wert-Platzhalter, Textvariable erlaubt

Prepared Statement mit Variablenwerten füllen und ausführen:

  EXCECUTE <PrepStmt>;    # Ohne Platzhalter
  EXCECUTE <PrepStmt>     # Falls Platzhalter "?" verwendet
    USING <Var1>, ...;    # MÜSSEN Variable sein, keiner Ausdr./Literale!

Prepared-Statement anzeigen gibt es nicht!

Prepared Statement löschen (Speicher freigeben, "Memory Leak" falls vergessen):

  DEALLOCATE PREPARE <PrepStmt>;   # OK
  DROP PREPARE <PrepStmt>;         # OK

Beispiel (Platzhalter "?" im Statement NICHT in "..." oder '...' setzen,
auch wenn der repräsentierte Wert dies eigentlich erfordern würde):

  # Beispiel 1
  DROP TABLE IF EXISTS buecher;
  CREATE TABLE buecher (
    titel CHAR(50),
    autor CHAR(30),
    isbn  INT
  );

  PREPARE neues_buch
    FROM "INSERT INTO buecher(titel, autor, isbn)
          VALUES (?, ?, ?)";

  SET @titel = "MySQL für Anfänger";   # Variable setzen
  SET @autor = "Thomas Birnthaler";    # Variable setzen
  SET @isbn  = "3-987654-321-0";       # Variable setzen

  EXECUTE neues_buch USING @titel, @autor, @isbn;               # OK
  EXECUTE neues_buch USING "Carrie", "King", "9-876543-210-X";  # Falsch, nur
                                                                # Variable OK!
  SELECT * FROM buecher;

  DEALLOCATE PREPARE neues_buch;         # OK
  DROP PREPARE neues_buch;               # Ebenfalls OK

  # Beispiel 2
```

```
  PREPARE stmt1 FROM "SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse";
  SET @a = 3;
  SET @b = 4;
  EXECUTE stmt1 USING @a, @b;
  DEALLOCATE PREPARE stmt1;

  # Beispiel 3
  SET @s = "SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse";
  PREPARE stmt2 FROM @s;
  SET @a = 6;
  SET @b = 8;
  EXECUTE stmt2 USING @a, @b;
  DEALLOCATE PREPARE stmt2;

  # Beispiel 4
  DROP TABLE IF EXISTS t1;
  CREATE TABLE t1 (a INT NOT NULL);
  INSERT INTO t1 VALUES (4), (8), (11), (32), (80), (90), (100);
  SELECT * FROM t1;

  SET @table  = "t1";
  SET @limit  = 2;
  SET @offset = 3;
  SET @s = CONCAT("SELECT * FROM ", @table, " LIMIT ", @limit,
                                          " OFFSET ", @offset);
  SELECT @s as "Statement1";
  PREPARE stmt3 FROM @s;
  EXECUTE stmt3;
  DEALLOCATE PREPARE stmt3;

  /*!50700
  SET @s = CONCAT("SELECT * FROM ", @table, " LIMIT ? OFFSET ?");
  SELECT @s as "Statement2";
  PREPARE stmt4 FROM @s;
  EXECUTE stmt4 USING 1, 3;
  EXECUTE stmt4 USING "1", "3";
  DEALLOCATE PREPARE stmt4; */

  # Beispiel 5
  # 1x an den Server schicken
  PREPARE select_boot
    FROM "SELECT b.boot_id, bd.laenge
          FROM boot AS b JOIN boot_detail AS bd
          WHERE b.name = ? AND bd.typ = CONCAT("fest_", ?);

  # Nx ausfuehren mit unterschiedlichen Werten
  EXECUTE select_boot USING @name, @typ;


19) Stored Routines (Stored Procedures/Functions)
-------------------------------------------------

Eigenschaften:
  * Dienen der Kapselung von Geschäftslogik in der Datenbank (nur 1x),
    viele verschiedene Anwendungen rufen sie nur auf (statt Nx programmiert)
  * Routine = Prozedur oder Funktion
    + Funktion hat Rückgabewert
      --> In Ausdrücken oder per SELECT aufzurufen (gerne vergessen!)
    + Prozedur hat KEINEN Rückgabewert (außer per INOUT/OUT-Parameter),
      --> Mit "CALL" aufzurufen (gerne vergessen!)
  * An eine Datenbank <Db> GEBUNDEN (werden mit dieser Datenbank gelöscht!),
    eigentliche Definition liegt in zentraler Datenbank "mysql"
  * Anweisungen in Funk./Proz. beziehen sich auf eine einzige Datenbank
    (auf beim Aufruf aktuell mit USE ausgewählte)
  * Körper (Body) besteht aus einer/mehreren SQL-Anweisungen
    (mehrere in BEGIN...END einschließen)
```

```
  * Erlaubt sind die meisten DDL-, DML- und DCL-Anweisungen
    (aber keine Transaktionssteuerung!), verboten sind:
    + LOCK/UNLOCK TABLES
    + ALTER VIEW
    + LOAD DATA/TABLE
    + PREPARE, EXECUTE, DEALLOCATE PREPARE   # nicht in Functions/Trigger
    + Alle in Prepared Statements nicht erlaubten Anweisungen
  * Rechte zum Erstellen/Ändern (CREATE ROUTINE, ALTER ROUTINE)
    und zum Ausführen (EXECUTE) notwendig
  * Lokale Variablen möglich
    + Definition:                 DECLARE <Var> <Typ> DEFAULT <Wert>;
    + Initialisierung:            SET <Var> := <Wert>;
    + Wert in Anweisung verwenden: ... <Var> ...
    + Parameter sind ebenfalls automatisch lokale Variablen
  * Aufruf:
      CALL <Proc>;          # In Default/Standard-Datenbank
      CALL <Db>.<Proc>      # In anderer Datenbank <Db>
      SELECT <Func>;        # Oder im Rahmen eines Ausdrucks
      SELECT <Db>.<Func>;   # Analog in anderer Datenbank <Db>
  * Eigene Funktionen wie eingebaute MySQL-Funktionen verwendbar
  * Rekursiver Aufruf in Funktionen/Prozeduren erlaubt (Fakultät)

Beschränkungen:
  * Anzahl und Typ der Parameter muss beim Aufruf eingehalten werden
  * Nur EIN Wert pro Variable übergebbar (keine Arrays/Objekte/Strukturen)
  * Variablen NICHT für Datenbank/Tabellen/Spaltennamen nutzbar
    (aber für LIMIT/OFFSET-Werte!)
  * Macht "Prelocking" aller darin benötigten Tabellen!
  * Replikation des Aufrufs oder der Änderungen durch den Aufruf?
    + MY!5.0: Binary Logging, alle DETERMINISTIC oder log_bin_trust_function_creators
  * Bei jedem Aufruf geparst und optimiert (auch wenn DETERMINISTIC)

Hinweise:
  * SET vor Variablenzuweisung gerne vergessen
  * CALL vor Prozeduraufruf gerne vergessen
  * SELECT vor Funktionsaufruf notwendig
  * Das Abschlusszeichen von SQL-Anweisungen ";" (SQL-Delimiter) ist VOR der
    Definition einer Prozedur/Funktion temporär auf "//" oder ähnliches setzen
    und DANACH wieder auf ";" zurücksetzen, damit ";" in der Definition zur
    Trennung der SQL-Statements verwendbar ist (MY!).

      DELIMITER //
      ... # Prozedur/Funktions-Definition, die SQL-Anweisungen mit ";" enthält,
      ... # ohne die soforige Ausführung dieser SQL-Anweisungen auszulösen
      ...
      DELIMITER ;

Vorteile/Nachteile:   !!!
  * S283

HINWEIS: Beispiele für alle Kontrollstrukturen folgen am Ende des Abschnitts


19a) Lokale Variablen
---------------------

Eigenschaften:
  * Nur innerhalb Prozeduren/Funktionen/Event/Trigger erlaubt
  * Nur am Anfang deklarierbar (direkt nach BEGIN)
  * Jeder SQL-Datentyp in Deklaration verwendbar (INT, FLOAT, CHAR(20), ...)
  * Initialisierbar per DEFAULT-Anweisung (sonst NULL als Startwert)
  * Kein "@" oder "@@" vor Variablenname (--> Session- oder Server-Variable)
  * Verschwinden beim Verlassen von Stored Procedure/Event

  DECLARE anz  INT;                       # Mit NULL initialisiert
  DECLARE max  FLOAT    DEFAULT 0.0;      # Mit 0.0 initialisiert
```

```
  DECLARE sum  FLOAT    DEFAULT 0.0;       # Mit 0.0 initialisiert
  DECLARE str  CHAR(30) DEFAULT "";        # Mit "" initialisiert
  DECLARE str2 CHAR(30) DEFAULT "Summe";   # Mit "Summe" initialisiert
```

19b) Wertzuweisung an Variable

Direkte Wertangabe oder Ergebnis einer Berechnung zuweisen
(Berechnung geht in DECLARE wahrscheinlich nicht):

```
  SET max = 9999;                    # Mit Vergleich "=" verwechselbar
  SET str = "Hallo";                 # Mit Vergleich "=" verwechselbar
  SET str = CONCAT(str, " welt!");   # Mit Vergleich "=" verwechselbar
  SET anz = anz + 1;                 # Mit Vergleich "=" verwechselbar
  SET max := 9999;                   # NICHT mit Vergleich "=" verwechselbar
  SET str := "Hallo";                # NICHT mit Vergleich "=" verwechselbar
  SET str := CONCAT(str, " welt!");  # NICHT mit Vergleich "=" verwechselbar
  SET anz := anz + 1;                # NICHT mit Vergleich "=" verwechselbar
```

HINWEISE:
- \+ Zugewiesener Wert muss zu Datentyp passen
- \+ In Ausdruck Aufruf von MySQL-Funktionen und eigener Funktionen erlaubt

SELECT-Ergebnis zuweisen:

```
  SELECT COUNT(*)   FROM pers    INTO anz;        # @anz und @@anz auch möglich
  SELECT MAX(preis) FROM artikel INTO max;        #
  SELECT SUM(anz)   FROM artikel INTO sum;        #
  SELECT SUM(anz)   INTO sum     FROM artikel;    # Auch möglich
  SELECT MIN(preis), MAX(preis), SUM(anz)         # Mehrere Variablen auf einmal
    FROM artikel INTO min, max, sum;              #
  SELECT SUM(anz)   INTO sum     FROM artikel;    # Auch möglich
  SELECT func(1, 3.14, "text")   INTO str;        # Funktionsergebnis
```

HINWEISE:
- * GENAU EIN Datensatz muss selektiert werden (Trick: LIMIT 1 anhängen),
  (mehrere Sätze müssen per CURSOR-Zugriff in einer Schleife behandelt werden)
- * Anzahl selektierter Spalten muss gleich Anzahl Variablen sein.

19c) Ausgabe von Variablen oder Daten

```
  SELECT @str AS "Titel";                                  # Einzelwert
  SELECT "Fester Text: ", @wert, " Euro";                  # Werteliste
  SELECT @sum / @anz * 100;                                # Ausdruck
  SELECT CONCAT("Fester Text: ", 17.00, @str) AS Text;     # Verketteter Text
  SELECT func(1, 3.14, @str2) AS "Funktionswert"           # Funktionsergebnis
```

19d) Kontrollstrukturen (Block/Compound Statement)

Eigenschaften:
- * Ein BEGIN...END-Block fasst mehrere SQL-Anweisungen zusammen
  (statt einzelner SQL-Anweisung verwendbar)
- * Benennung per Marke <Label> möglich
  (für ITERATE, LEAVE, END, END LOOP, END WHILE, END REPEAT)
- * Zu Beginn lokale Variablen per DECLARE-Anweisungen einrichtbar
  - \+ Definiert Geltungsbereich
  - \+ Ausserhalb nicht mehr vorhanden
- * Optionale Marke <Label> vor BEGIN kann max. 16 Zeichen lang sein
  - \+ Falls verwendet, am Kontrollstruktur-Ende zu wiederholen
  - \+ Als Zielname von benannten ITERATE- und LEAVE-Anweisungen

```
  [<Label>:] BEGIN   # Evtl. Marke setzen
    ...
  END [<Label>];     # Marke wiederholen falls verwendet!
```

19e) Kontrollstrukturen (Verzweigungen)

IF-Verzweigung: Anweisungen nach THEN der von oben nach unten ersten wahren Bedingung <CondN> werden ausgeführt; oder falls keine <CondN> TRUE wird, die Anweisungen nach ELSE; oder falls ELSE nicht vorhanden: gar keine Anweisung):

```
  IF <Cond1> THEN       #
    <Stmt>;             # Anweisung
    ...                 # ...
  ELSEIF <Cond2> THEN   # Beliebig oft (auch weglassbar)
    <Stmt>;             # Anweisung
    ...                 # ...
  ELSE                  # Weglassbar
    <Stmt>;             # Anweisung
    ...                 # ...
  END IF;               # ";" nicht vergessen!
```

CASE-Verzweigung: Anweisungen nach THEN der von oben nach unten ersten wahren Bedingung <CondN> werden ausgeführt; oder falls keine <CondN> TRUE wird, die Anweisungen nach ELSE; oder falls ELSE nicht vorhanden: gar keine Anweisung):

```
  CASE                              # Variante A: Bedingungen prüfen
    WHEN <Cond1> THEN <Stmt>; ...   # Bedingung <Cond1> prüfen
    WHEN <Cond2> THEN <Stmt>; ...   # Bedingung <Cond2> prüfen
    ...                             #
    ELSE <Stmt>; ...                # Besser immer angeben!
  END CASE;                         # ";" nicht vergessen!
```

CASE-Fallunterscheidung: <Expr> wird ausgewertet und das Ergebnis von oben nach unten mit den Werten <ValN> verglichen und Anweisungen nach zugehörigem THEN ausgeführt; oder falls kein WHEN-Fall zutrifft, die Anweisungen nach ELSE; oder falls ELSE nicht vorhanden: gar keine Anweisung):

```
  CASE <Expr>                       # Variante B: Ergebnis mit Wert vergleichen
    WHEN <Val1> THEN <Stmt>; ...    # Ergebnis mit Wert <Val1> vergleichen
    WHEN <Val2> THEN <Stmt>; ...    # ...
    ...                             # ...
    ELSE <Stmt>; ...                # Besser immer angeben!
  END CASE;                         # ";" nicht vergessen!
```

ACHTUNG: Tritt bei CASE ohne ELSE-Zweig ein Fall auf, der nicht von einem WHEN abgefragt wird, wird eine Fehlermeldung ausgegeben (seltsames Verhalten!)
--> Immer ELSE-Zweig angeben!

19f) Kontrollstrukturen (Schleifen)

Hinweise (Beispiele folgen weiter unten):
- Marke <Label> kann fehlen
  (falls verwendet, ist sie am Ende der Konstrollstruktur zu wiederholen!)
- LEAVE bricht Schleife vorzeitig ab (es geht danach weiter)
- ITERATE springt zum Schleifenanfang (nächster Durchlauf)
- Bei verschachtelten Schleifen ist Marke <Label> notwendig,
  um eine der äußeren Schleifen neu zu starten oder abzubrechen
- Es gibt kein GOTO <Label> (wurde kurzzeitig getestet, aber wieder entfernt)

Endlosschleife:

```
  [<Label>:] LOOP        # Evtl. Marke setzen
```

```
    <Stmt>;              # Anweisung in Schleife
    ...                  # ...
    ITERATE [<Label>];   # Schleife neu starten
    LEAVE [<Label>];     # Schleife verlassen
    ...                  # ...
  END LOOP [<Label>];    # Marke wiederholen falls verwendet
```

Nichtabweisende Schleife (Inneres mind. 1x ausgeführt!):

```
  [<Label>:] REPEAT                   # Evtl. Marke setzen
    <Stmt>;                           # Anweisung in Schleife
    ...                               # ...
    ITERATE [<Label>];                # Schleife neu starten
    LEAVE [<Label>];                  # Schleife verlassen
    ...                               # ...
  UNTIL <Cond> END REPEAT [<Label>];  # Marke wiederholen falls verwendet
```

Abweisende Schleife (Inneres evtl. gar nicht ausgeführt!):

```
  [<Label>:] WHILE <Cond> DO   # Evtl. Marke setzen
    <Stmt>;                    # Anweisung
    ...                        # ...
    ITERATE [<Label>];         # Schleife neu starten
    LEAVE [<Label>];           # Schleife verlassen
    ...                        # ...
  END WHILE [<Label>];         # Marke wiederholen falls verwendet
```

19g) Prozeduren

Hinweise:

- Name, Parameter (leere Liste erlaubt) und Prozedurkörper sind anzugeben
  - Parameter sind in der Form [<InOut>] <Name> <Typ> anzugeben,
    (z.B. INOUT wert TINYINT, IN ist Standard)
  - <Typ> ist ein beliebiger SQL-Datentyp (OHNE Längenangabe!)
- Eingabeparameter verhalten sich wie initialisierte lokale Variablen:
  DECLARE <Var> <Typ> DEFAULT <Wert_von_aussen>;
- Kann per SELECT-Anweisung oder per OUT/INOUT-Parameter Ergebnis zurückgeben
- Prozedur ohne Parameter darf ohne "()" aufgerufen werden

Parameter-Eigenschaft <InOut>:

Form	Typ	Bemerkung
IN	Eingabe (STD)	Variable/Ausdruck/Literal ("abc", 123.45, abs(-1))
OUT	Ausgabe	Muss Variable sein (z.B. @var)
INOUT	Ein/Ausgabe	Muss Variable sein (z.B. @var)

Definition einer Prozedur:

```
  DELIMITER //
  CREATE PROCEDURE <Proc> ([<InOut>] <Param> <Typ>, ...)
    [[NOT] DETERMINISTIC]
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
    [SQL SECURITY {DEFINER | INVOKER}]
    [COMMENT <String>]
  BEGIN
    <Stmt>;
    ...
  END //
  DELIMITER ;
```

Aufruf einer Prozedur:

```
  * Anzahl übergebener Argumente muss zu Definition passen
  * Für OUT- und INOUT-Parameter muss eine Variable übergeben werden
  * Für IN-Parameter darf auch Wert/Konstante übergeben werden

Beispiel:

  # Definition
  DROP PROCEDURE IF EXISTS eins;
  DELIMITER //
  CREATE PROCEDURE eins(IN anz INT, IN preis FLOAT, INOUT text CHAR(20))
  BEGIN
    SELECT anz AS "1.", preis AS "2.", text AS "3.";
    SET text = CONCAT(ROUND(anz * preis, 2), " Euro");
  END //
  DELIMITER ;

  # Aufruf
  SET @str = "Text";             # Variable belegen
  CALL eins(1, 3.14, @str);      # Variable übergeben
  SELECT @str;                   # Variable ausgeben

  # Definition
  DROP PROCEDURE IF EXISTS zwei;
  DELIMITER //
  CREATE PROCEDURE zwei(IN name CHAR(20))
  BEGIN
    SELECT user, host, password FROM mysql.user WHERE user = name LIMIT 1;
  END //
  DELIMITER ;

  # Aufruf
  CALL zwei("root");                                          # Variable übergeben
  CALL zwei("tom");                                           # Variable übergeben
  #SET @user, @host, @password = CALL zwei("tom");            # Geht leider nicht!
  #SELECT CALL zwei("tom") INTO @user, @host, @password;      # Geht leider nicht!

Informationen über alle Prozeduren auflisten
(Datenbank, Name, Typ, Ersteller, Erstellungs- und Änderungsdatum):

  SHOW PROCEDURE STATUS;
  SHOW PROCEDURE STATUS LIKE "<Muster>";

Prozedur-Definition anzeigen:

  SHOW CREATE PROCEDURE <Proc>;         # OK
  SHOW PROCEDURE CODE <Proc>;           # Fehler!

Charakteristik einer gespeicherten Prozedur ändern (nicht Name oder Inhalt!):

  ALTER PROCEDURE <Proc>
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA} |
    SQL SECURITY {DEFINER | INVOKER} |
    COMMENT <String>

Löschen einer Prozedur:

  DROP PROCEDURE <Proc>;
  DROP PROCEDURE IF EXISTS <Proc>;   # Kein Fehler falls nicht existent

Beispiele für Prozeduren:

  DELIMITER //   # Nicht vergessen!

  #--------------------------------------------------------------------------
  # Prozedur 1
  #--------------------------------------------------------------------------
```

```
DROP PROCEDURE IF EXISTS bench //     # ACHTUNG: ; führt zu Fehler

CREATE PROCEDURE bench(IN anz INT)
BEGIN
  SELECT NOW();
  WHILE anz > 0 DO
    SET anz = anz - 1;
  END WHILE;
  SELECT NOW();
END //

CALL bench(10) //
CALL bench(1000) //
CALL bench(100000) //
CALL bench(10000000) //

#--------------------------------------------------------------------------
# Prozedur 2
#--------------------------------------------------------------------------
DROP PROCEDURE IF EXISTS conv_xml_special //

CREATE PROCEDURE conv_xml_special(INOUT str CHAR(255))
BEGIN
  SET str = REPLACE(str, "&", "&amp;");
  SET str = REPLACE(str, "<", "&lt;");
  SET str = REPLACE(str, ">", "&gt;");
END //

SET @str = "a && (b <> c)" //
SELECT @str //
CALL conv_xml_special(@str) //
SELECT @str //

#--------------------------------------------------------------------------
# Prozedur 3
#--------------------------------------------------------------------------
DROP TABLE IF EXISTS pers //

CREATE TABLE pers (                  # Variante A (Engine = InnoDB = STD)
  nr      INT,
  vorname VARCHAR(30),
  name    VARCHAR(30)
) //

INSERT INTO pers (nr, vorname, name)
  VALUES (1, "Thomas",  "Birnthaler"),
         (2, "Markus",  "Mueller"),
         (8, "Andrea",  "Bayer"),
         (9, "Richard", "Seiler"),
         (7, "Heinz",   "Bayer") //

DROP PROCEDURE IF EXIST select_limited //

CREATE PROCEDURE select_limited(IN grenze INT)
BEGIN
  DECLARE anz INT DEFAULT 0;
                                       # Leerzeile zur Übersicht
  SELECT COUNT(*) INTO anz FROM pers;
  IF anz <= grenze THEN
    SELECT * FROM pers;
  ELSE
    SELECT * FROM pers LIMIT grenze;   # Erlaubt als Parameter!
  END IF;
END //

CALL select_limited(100) //
```

```
CALL select_limited(6) //
CALL select_limited(5) //
CALL select_limited(4) //

#-------------------------------------------------------------------------
# Prozedur 4
#-------------------------------------------------------------------------
DROP TABLE IF EXISTS artikel //

CREATE TABLE artikel (
  nr     INT,
  bez    CHAR(30),
  anz    INT,
  preis  FLOAT(6,2)
) //

INSERT INTO artikel (nr, bez, anz, preis)
  VALUES (1, "Artikel A", 100,   7.99),
         (2, "Artikel B",  10,  19.50),
         (3, "Artikel C",   5,  79.80),
         (4, "Artikel D",   1, 123.00),
         (5, "Artikel E",   0,   2.49) //

DROP PROCEDURE IF EXISTS max_preis //

CREATE PROCEDURE max_preis(IN grenze FLOAT)
BEGIN
  DECLARE max  FLOAT    DEFAULT 0.0;
                                        # Leerzeile zur Übersicht
  SELECT MAX(preis) INTO max FROM artikel;
  IF max > grenze THEN
    SELECT "Maximaler Preis: ", max;
  END IF;
END //

CALL max_preis( 20) //
CALL max_preis(100) //
CALL max_preis(200) //

#-------------------------------------------------------------------------
# Prozedur 5
#-------------------------------------------------------------------------
DROP TABLE IF EXISTS artikel //

CREATE TABLE artikel (
  nr     INT,
  bez    CHAR(30),
  anz    INT,
  preis  FLOAT(6,2)
) //

INSERT INTO artikel (nr, bez, anz, preis)
  VALUES (1, "Artikel A", 100,   7.99),
         (2, "Artikel B",  10,  19.50),
         (3, "Artikel C",   5,  79.80),
         (4, "Artikel D",   1, 123.00),
         (5, "Artikel E",   0,   2.49) //

DROP PROCEDURE IF EXISTS summe_artikel //

CREATE PROCEDURE summe_artikel(IN artnr INT)
BEGIN
  DECLARE artanz INT DEFAULT 0;
                                          # Leerzeile zur Übersicht
  SELECT SUM(anz) INTO artanz FROM artikel
    WHERE nr = artnr
```

```
      GROUP BY nr;

  # SELECT "Artanz: ", artanz;    # DEBUG-Ausgabe

  IF artanz > 1 THEN
    SELECT DISTINCT nr, anz
      FROM artikel
        WHERE nr = artnr;
  END IF;
END //

CALL summe_artikel(1) //
CALL summe_artikel(2) //
CALL summe_artikel(3) //
CALL summe_artikel(4) //
CALL summe_artikel(5) //
CALL summe_artikel(6) //

#-------------------------------------------------------------------------
# Prozedur 6
#-------------------------------------------------------------------------
DROP TABLE IF EXISTS artikel //

CREATE TABLE artikel (
  nr     INT,
  bez    CHAR(30),
  anz    INT,
  preis  FLOAT(6,2)
) //

INSERT INTO artikel (nr, bez, anz, preis)
  VALUES (1, "Artikel A", 100,   7.99),
         (2, "Artikel B",  10,  19.50),
         (3, "Artikel C",   5,  79.80),
         (4, "Artikel D",   1, 123.00),
         (5, "Artikel E",   0,   2.49) //

DROP PROCEDURE avg_sum_max_min //

CREATE PROCEDURE avg_sum_max_min(IN typ CHAR)
BEGIN
  # CASE-Variante A
  CASE
    WHEN typ = "G" THEN SELECT AVG(preis) AS "Durchschnitt" FROM artikel;
    WHEN typ = "S" THEN SELECT SUM(preis) AS "Summe"        FROM artikel;
    WHEN typ = "A" THEN SELECT MAX(preis) AS "Maximum"      FROM artikel;
    WHEN typ = "I" THEN SELECT MIN(preis) AS "Minimum"      FROM artikel;
    ELSE SELECT "Geben Sie G, S, A oder I ein";   # Muss da sein!
  END CASE;

  # CASE-Variante B
  CASE typ
    WHEN "G" THEN SELECT AVG(preis) AS "Durchschnitt" FROM artikel;
    WHEN "S" THEN SELECT SUM(preis) AS "Summe"        FROM artikel;
    WHEN "A" THEN SELECT MAX(preis) AS "Maximum"      FROM artikel;
    WHEN "I" THEN SELECT MIN(preis) AS "Minimum"      FROM artikel;
    ELSE SELECT "Geben Sie G, S, A oder I ein";   # Muss da sein!
  END CASE;
END //

CALL avg_sum_max_min("G") //
CALL avg_sum_max_min("S") //
CALL avg_sum_max_min("A") //
CALL avg_sum_max_min("I") //
CALL avg_sum_max_min("X") //
```

```
  DELIMITER ;     # Nicht vergessen!
```

19h) Funktionen

Hinweise:

* Name, Parameter (leere Liste erlaubt), Rückgabetyp, Funktionskörper nötig
  + Parameter sind in der Form <Name> <Typ> anzugeben (z.B. Wert FLOAT)
  + <Typ> ist ein beliebiger SQL-Datentyp (OHNE Längenangabe!)
  + MUSS mind. 1x per RETURN <Expr> Wert vom pass. Typ zurückgeben (oder NULL)
* In einer Funktion sind KEINE Tabellenzugriffe erlaubt
* Ebenso dürfen darin KEINE Prozeduren aufgerufen werden
* Gibt per Rückgabewert RETURN <Wert> Ergebnis zurückgeben
* Funktion ohne Parameter darf ohne "()" aufgerufen werden

Definition einer Funktion:

```
  DELIMITER //
  CREATE FUNCTION <Func> (<Param> <Typ>, ...)
    RETURNS <Typ>
    [[NOT] DETERMINISTIC]
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA}
    [SQL SECURITY {DEFINER | INVOKER}]
    [COMMENT <String>]
  BEGIN
    <Stmt>;
    ...
    RETURN <Expr>;   # Werterückgabe beliebig oft (mind. 1x!)
    ...
  END //
  DELIMITER ;
```

Aufruf einer Funktion:

```
  SELECT func(1, 3.14, "text");                # A) Ergebnis als Datensatz
  SET @var = SELECT func(1, 3.14, "text");     # B) Ergebnis in Variable speich.
  SELECT func(1, 3.14, "text") INTO @var;      # C) Ergebnis in Variable speich.
  SELECT 3.14 * func(1);                       # D) Als Teil eines Ausdrucks
```

Informationen über alle Funktionen auflisten
(Datenbank, Name, Typ, Ersteller, Erstellungs- und Änderungsdatum):

```
  SHOW FUNCTION STATUS;
  SHOW FUNCTION STATUS LIKE "<Muster>";
```

Funktions-Definition anzeigen:

```
  SHOW CREATE FUNCTION <Func>;          # OK
  SHOW FUNCTION CODE <Func>;            # Fehler!
```

Charakteristik einer gespeicherten Funktion ändern
(nicht Name, Parameter oder Inhalt!):

```
  ALTER FUNCTION <Func>
    [[NOT] DETERMINISTIC]
    {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA} |
    SQL SECURITY {DEFINER | INVOKER} |
    COMMENT <String>;
```

Löschen einer Funktion:

```
  DROP FUNCTION <Func>;
  DROP FUNCTION IF EXISTS <Func>;   # Kein Fehler falls nicht existent
```

Beispiele für Funktionen:

```
DELIMITER //

#-------------------------------------------------------------------------
# Funktion 1
#-------------------------------------------------------------------------
DROP FUNCTION IF EXISTS brutto_netto //

CREATE FUNCTION brutto_netto(typ CHAR, wert1 FLOAT, wert2 FLOAT)
  RETURNS FLOAT
  NO SQL
BEGIN
  CASE
    WHEN typ = "G" THEN RETURN wert1 * 100 / wert2;     # G=P*100/S
    WHEN typ = "S" THEN RETURN wert1 * 100 / wert2;     # S=P*100/G
    WHEN typ = "P" THEN RETURN wert1 * wert2 / 100;     # P=G*S/100
    ELSE RETURN 0.0;
  END CASE;
END //

SELECT brutto_netto("G", 1234, 100) //    # G)rundwert
SELECT brutto_netto("S", 1234, 100) //    # S)atz
SELECT brutto_netto("P", 1234, 1000) //   # P)rozentwert
SELECT brutto_netto("X", 1234, 100) //

DELIMITER ;

#-------------------------------------------------------------------------
# Funktion 2
#-------------------------------------------------------------------------
DROP FUNCTION IF EXISTS sumup_to_n //

CREATE FUNCTION sumup_to_n(n INT)
  RETURNS INT
  NO SQL
BEGIN
  DECLARE i   INT DEFAULT 1;
  DECLARE sum INT DEFAULT 0;
                                          # Leerzeile zur Übersicht
  IF n = 0 THEN
    RETURN 0;
  ELSE
    WHILE i <= n DO
      SET sum = sum + i;
      SET i = i + 1;
    END WHILE;
  END IF;
  RETURN sum;
END //

SELECT sumup_to_n(0) //
SELECT sumup_to_n(100) //
SELECT sumup_to_n(1000) //
SELECT sumup_to_n(10000) //

#-------------------------------------------------------------------------
# Funktion 3
#-------------------------------------------------------------------------
DROP FUNCTION IF EXISTS area_or_volume //

CREATE FUNCTION area_or_volume(typ CHAR, radius FLOAT)
  RETURNS FLOAT
  NO SQL
BEGIN
  CASE
    WHEN typ = "A" THEN RETURN PI() * POW(radius, 2);           # A=pi*r^2
```

```
    WHEN typ = "V" THEN RETURN 4 * PI() / 3 * POW(radius, 3);   # V=4/3*pi*r^3
    ELSE RETURN 0.0;
  END CASE;
END //

SELECT area_or_volume("A", 100) //
SELECT area_or_volume("V", 100) //
SELECT area_or_volume("X", 100) //

#-------------------------------------------------------------------------------
# Funktion 4
#-------------------------------------------------------------------------------
DROP FUNCTION IF EXISTS roman //

CREATE FUNCTION roman(n INT)
  RETURNS CHAR(30)
  NO SQL
BEGIN
  DECLARE erg CHAR(30) DEFAULT "";
                                          # Leerzeile zur Übersicht
  WHILE n >= 1000 DO
    SET erg = CONCAT(erg, "M");
    SET n = n - 1000;
  END WHILE;
  # SELECT "ONE: ", erg, n;   # Nicht erlaubt!
  CASE
    WHEN n >= 900 THEN SET erg = CONCAT(erg, "CM"); SET n = n - 900;
    WHEN n >= 500 THEN SET erg = CONCAT(erg, "D");  SET n = n - 500;
    WHEN n >= 400 THEN SET erg = CONCAT(erg, "CD"); SET n = n - 400;
    ELSE SET erg = erg;   # Sonst Fehlermeldung für 300-000
  END CASE;

  WHILE n >= 100 DO
    SET erg = CONCAT(erg, "C");
    SET n = n - 100;
  END WHILE;
  CASE
    WHEN n >= 90 THEN SET erg = CONCAT(erg, "XC"); SET n = n - 90;
    WHEN n >= 50 THEN SET erg = CONCAT(erg, "L");  SET n = n - 50;
    WHEN n >= 40 THEN SET erg = CONCAT(erg, "XL"); SET n = n - 40;
    ELSE SET erg = erg;   # Sonst Fehlermeldung für 30-00
  END CASE;

  WHILE n >= 10 DO
    SET erg = CONCAT(erg, "X");
    SET n = n - 10;
  END WHILE;
  CASE
    WHEN n >= 9 THEN SET erg = CONCAT(erg, "IX"); SET n = n - 9;
    WHEN n >= 5 THEN SET erg = CONCAT(erg, "V");  SET n = n - 5;
    WHEN n >= 4 THEN SET erg = CONCAT(erg, "IV"); SET n = n - 4;
    ELSE SET erg = erg;   # Sonst Fehlermeldung für 3-0
  END CASE;

  WHILE n >= 1 DO
    SET erg = CONCAT(erg, "I");
    SET n = n - 1;
  END WHILE;

  RETURN erg;
END //

SELECT roman(0) //
SELECT roman(1) //
SELECT roman(4) //
SELECT roman(5) //
```

```
SELECT roman(6) //
SELECT roman(9) //
SELECT roman(10) //
SELECT roman(11) //
SELECT roman(49) //
SELECT roman(50) //
SELECT roman(51) //
SELECT roman(99) //
SELECT roman(100) //
SELECT roman(101) //
SELECT roman(222) //
SELECT roman(333) //
SELECT roman(444) //
SELECT roman(499) //
SELECT roman(500) //
SELECT roman(501) //
SELECT roman(666) //
SELECT roman(999) //
SELECT roman(1000) //
SELECT roman(1001) //
SELECT roman(1962) //
SELECT roman(1999) //
SELECT roman(2000) //
SELECT roman(2007) //
SELECT roman(4567) //
SELECT roman(9999) //

#-----------------------------------------------------------------------------
# Funktion 5 (iterative Variante)
#-----------------------------------------------------------------------------
DROP FUNCTION IF EXISTS ifakultaet //

CREATE FUNCTION ifakultaet(n INT)
  RETURNS DECIMAL(65)
  NO SQL
BEGIN
  DECLARE erg DECIMAL(65) DEFAULT 1;
                                              # Leerzeile zur Übersicht
  IF n < 0 OR n > 50 THEN
    SET erg = NULL;
  ELSE
    WHILE n > 0 DO
      SET erg = erg * n;
      SET n = n - 1;
    END WHILE;
  END IF;
  RETURN erg;
END //

SELECT ifakultaet(-10) AS "Fakultät von -10" // //
SELECT ifakultaet(0)   AS "Fakultät von 0" //
SELECT ifakultaet(1)   AS "Fakultät von 1" //
SELECT ifakultaet(5)   AS "Fakultät von 5" //
SELECT ifakultaet(10)  AS "Fakultät von 10" //
SELECT ifakultaet(50)  AS "Fakultät von 50" //
SELECT ifakultaet(100) AS "Fakultät von 100" //

#-----------------------------------------------------------------------------
# Funktion 6 (rekursive Variante)
# ERROR 1424 (HY000): Recursive stored functions and triggers are not allowed
#-----------------------------------------------------------------------------
DROP FUNCTION IF EXISTS rfakultaet //

CREATE FUNCTION rfakultaet(n INT)
  RETURNS DECIMAL(65)
  NO SQL
```

```
BEGIN
  DECLARE erg DECIMAL(65) DEFAULT 1;
                                              # Leerzeile zur Übersicht
  IF n < 0 OR n > 50 THEN
    RETURN NULL;
  ELSEIF n <= 1 THEN
    RETURN n;
  ELSE
    RETURN (n * rfakultaet(n - 1));
  END IF;
END //

SELECT rfakultaet(-10) AS "Fakultät von -10" // //
SELECT rfakultaet(0)   AS "Fakultät von 0" //
SELECT rfakultaet(1)   AS "Fakultät von 1" //
SELECT rfakultaet(5)   AS "Fakultät von 5" //
SELECT rfakultaet(10)  AS "Fakultät von 10" //
SELECT rfakultaet(50)  AS "Fakultät von 50" //
SELECT rfakultaet(100) AS "Fakultät von 100" //

#-------------------------------------------------------------------------------
# Funktion 7 (Ersatz für DATEDIFF(CURDATE(), geburts_datum))
# Alter zu einem Geburtsdatum bezogen auf heute = NOW() ausrechnen
# Einfache Version: Nur die Jahre voneinander abziehen (Hilfsfkt. YEAR(d))
# Komplexe Version: Auch Monate und Tage voneinander abziehen
#                   (Hilfsfkt. MONTH(d), DAY()) und Kommazahl zurückliefern
# ACHTUNG: SQL-Schlüsselwort (z.B. "alter" nicht als Fktname verwendbar!)
#-------------------------------------------------------------------------------
DROP FUNCTION IF EXISTS age //

CREATE FUNCTION age(geburts_datum DATE)
  RETURNS FLOAT
  NO SQL
BEGIN
  DECLARE year_diff  INT DEFAULT 0;
  DECLARE month_diff INT DEFAULT 0;
  DECLARE day_diff   INT DEFAULT 0;
                                              # Leerzeile zur Übersicht
  # Differenzen der 3 Komponenten ermitteln
  SET year_diff  = YEAR(CURDATE())  - YEAR(geburts_datum);
  SET month_diff = MONTH(CURDATE()) - MONTH(geburts_datum);
  SET day_diff   = DAY(CURDATE())   - DAY(geburts_datum);

  # Negative Differenz erfordert "Übertrag" von nächstgrößerer Komponente
  IF day_diff < 0 THEN
    SET day_diff   = day_diff + 30;
    SET month_diff = month_diff - 1;
  END IF;

  IF month_diff < 0 THEN
    SET month_diff = month_diff + 12;
    SET year_diff  = year_diff - 1;
  END IF;

  # Debugausgabe in Funktionen nicht erlaubt!
  # SET diff = "Differenz: ", DATEDIFF(CURDATE(), geburts_datum);

  # Ganze Jahre + "Bruchteil" eines Jahres summieren
  RETURN year_diff + (month_diff / 12) + (day_diff / 360);
END //

SELECT age("1962.07.01") //
SELECT age("2007.05.09") //
SELECT age("2006.05.09") //
SELECT age("2005.05.09") //
SELECT age("2004.04.01") //
```

```
  SELECT age("2004.05.01") //
  SELECT age("2004.05.09") //
  SELECT age("2004.05.17") //
  SELECT age("2004.06.17") //

  DELIMITER ;
```

20) Trigger

Löst automatische Reaktion beim Einfügen, Ändern, Löschen von Datensätzen in Tabellen aus. Kann VOR und/oder NACH der auslösenden SQL-Anweisung erfolgen.

Zweck:
- * Prüfung/Korrektur/Berechnung von Spaltenwerten mit BEFORE-Trigger
  (z.B. "Matchcode" aus PLZ, Vorname, Nachname zur Dublettenerkennung erstellen)
- * Andere Tabellen updaten (History, Denormalisierung) mit AFTER-Trigger
- * Constraints oder Business Rules realisieren

Eigenschaften:
- * EINER Tabelle zugeordnet (NICHT View, NICHT temp. Tabelle!)
  (Tabelle mit passenden Spalten muss existieren)
- * VOR/NACH SQL-Anweisung auf einer Tabelle autom. ausgeführt
  (BEFORE/AFTER: INSERT, UPDATE, DELETE, REPLACE = INSERT/UPDATE)
  + Pro Datensatz: FOR EACH ROW   (Standard in MY!)
  + Pro SQL-Anweisung: FOR EACH STATEMENT (nicht vorhanden in MY!)
- * Gespeichert pro Tabelle <Tbl> in Datei "<Tbl>.TRG" (+ "<Tbl>.TRN")
- * Name eines Triggers muss pro Datenbank (Schema) eindeutig sein
- * TIMESTAMP- und AUTO_INCREMENT-Spalten haben eine Art "Default-Trigger"

Beschränkungen:
- * Ein Trigger LOCKT während Ausführung ALLE beteiligten Tabellen vollständig!
  --> So schnell wie möglich durchlaufen!
- * NUR EIN Trigger pro Kombination <Time> + <Event> möglich
  + Mehrere Trigger zu unterschiedlichem <Time> + <Event> pro Tabelle möglich
  + In MY!5.7.x auch mehrere Trigger pro Kombination <Time> + <Event> erlaubt
- * Trigger haben gleiche Einschränkungen wie Stored Procedures
  (als <Stmt> zw. BEGIN...END alle in Stored Procedures erlaubten möglich)
  + Keine Prepared Statements darin erlaubt
  + Keine Transaktionen oder Locks können darin beginnen/enden
- * Im Trigger Zugriff auf andere Tabellen erlaubt
  (schreiben auf Tabelle für die Trigger ausgelöst wurde nicht erlaubt)
- * Im Trigger Stored Procedures per CALL und Funktionen aufrufbar
  + Dürfen keine Daten per SELECT produzieren
  + Datenaustausch nur per OUT/INOUT- und Rückgabe-Parameter
- * Derzeit nicht durch kaskadierte Foreign Key Aktionen ausgelöst (kommt noch!)
- * Per LEAVE vorzeitiges Trigger-Ende möglich (nicht mit RETURN)
- * Nur in BEFORE ist NEW.<Col> schreibbar
- * In BEFORE hat NEW.<Col> einer AUTO_INCREMENT-Spalte den Wert "0"
- * Für Compound-Statements als Trigger-Code ist Delimiter ";" zu ändern
  (analog Stored Procedures)
- * Row-based Replication: Trigger auf Slave werden NICHT ausgelöst
  Statement-based Replication: Trigger auf Slave werden ausgelöst
- * TRICK: ROW_COUNT() ist 1 außer für den ersten Datensatz
  --> Trigger nur 1x pro Statement ausführen --> FOR EACH STATEMENT Emulation

Fehlerbehandlung:
- * Scheitert BEFORE-Trigger, wird eigentliche SQL-Anweisung NICHT durchgeführt
- * BEFORE-Trigger durchgeführt, auch wenn eigentliche SQL-Anweisung scheitert
- * AFTER-Trigger nur durchgeführt, wenn BEFORE-Trigger (falls vorhanden)
  UND SQL-Anweisung klappen
- * Fehler im BEFORE/AFTER-Trigger führt zu Fehler in auslösender SQL-Anweisung
  + Transaktion:
    - Fehler der SQL-Anweisung nehmen ausgelöste Trigger wieder zurück
    - Trigger-Fehler lösen Rücknahme der ganzen SQL-Anweisung aus

```
    + Nicht-transakt. Tabellen:
      - Ausgelöste Trigger bei Fehler in SQL-Anweisung nicht zurückgenommen
      - Ebenso umgekehrt
```

Zeitlicher Ablauf (ro=read-only, rw=read-write) beim Abarbeiten einer SQL-Anweisung <Stmt>:

Zeitliche Reihenfolge der ausgelöste Trigger			Daten Zugriff
SQL-Anweisung	Key fehlt	Key vorhanden	Daten
INSERT ...	BEFORE INSERT <Stmt> AFTER INSERT	BEFORE INSERT --- ---	OLD ro
UPDATE ...	--- --- ---	BEFORE UPDATE <Stmt> AFTER UPDATE	OLD ro NEW rw
DELETE ...	--- --- ---	BEFORE DELETE <Stmt> AFTER DELETE	OLD ro
REPLACE ...	BEFORE INSERT --- <Stmt> --- AFTER INSERT	BEFORE INSERT BEFORE DELETE <Stmt> AFTER DELETE AFTER INSERT	NEW rw
INSERT ... ON DUPLICATE KEY UPDATE ...	BEFORE INSERT --- <Stmt> --- AFTER INSERT	BEFORE INSERT BEFORE UPDATE <Stmt> AFTER UPDATE ---	NEW rw

Definition eines Triggers:

```
DELIMITER //
CREATE
  [DEFINER {<User> | CURRENT USER}]
  TRIGGER <Trig> <Time> <Event>
ON <Tbl>                                    # Genau EINE Tabelle
FOR EACH ROW                                # MySQL-Einschränkung
BEGIN                                       # Oder nur <Stmt>;
  <Stmt>;
  ...
END //
DELIMITER ;
```

Mögliche Trigger-Zeitpunkte <Time>:

Time	Bedeutung
BEFORE	Vor Tabellenänderung
AFTER	Nach Tabellenänderung

Mögliche Trigger-Ereignisse <Event>:

Event	Bedeutung
INSERT	Einfügen einer neuen Zeile (INSERT, LOAD DATA, REPLACE)

```
  | UPDATE | Ändern einer Zeile (UPDATE)                                          |
  | DELETE | Löschen einer Zeile (DELETE, REPLACE, nicht DROP TABLE, TRUNCATE) |
  +--------+-----------------------------------------------------------------+
```

Zugriff auf alte/neue Werte der Tabellenspalten während Trigger-Durchführung:

```
  +-----------+-------------------------------------------------------+
  | Name      | Bedeutung                                             |
  +-----------+-------------------------------------------------------+
  | OLD.<Col> | Alter Wert vor  DELETE und UPDATE (read only)         |
  | NEW.<Col> | Neuer Wert nach INSERT und UPDATE (read write in BEFORE) |
  +-----------+-------------------------------------------------------+
```

Trigger anzeigen:

```
  SHOW TRIGGERS;                          # Default <Db>
  SHOW TRIGGERS LIKE "a%";                # Default <Db>, Tabellenname
  SHOW TRIGGERS FROM <Db> LIKE "a%";      # Spezielle <Db>, Tabellenname
  SELECT TRIGGER_NAME, EVENT_MANIPULATION, EVENT_OBJECT_TABLE, ACTION_STATEMENT
    FROM INFORMATION_SCHEMA.TRIGGERS
      WHERE TRIGGER_SCHEMA = <Db>;
```

Trigger ändern = mit gleichem Namen einen neuen Trigger definieren

Trigger löschen (kein Bezug auf Tabellenname!):

```
  DROP TRIGGER <Trig>;
  DROP TRIGGER IF EXISTS <Trig>;    # Kein Fehler falls nicht existent
```

Alle Trigger einer Tabelle werden mit der Tabelle gelöscht:

```
  DROP TABLE <Tbl>;
```

Beispiele für Trigger:

```
  #--------------------------------------------------------------------------
  # Trigger 0 (Summe aller eingefügten Spaltenwerte ermitteln)
  #--------------------------------------------------------------------------
  DROP TABLE IF EXISTS account;             # Löscht auch Trigger

  CREATE TABLE account (
    acct_num INT,
    amount   DECIMAL(10,2)
  );

  SET @sum = 0;
  SET @old = 0;

  DROP TRIGGER IF EXISTS t0;

  DELIMITER //

  CREATE TRIGGER t0 BEFORE UPDATE ON account
  FOR EACH ROW
  BEGIN
    SET @sum = @sum + NEW.amount;
    SET @old = @old + OLD.amount;
  END //

  DELIMITER ;

  SHOW TRIGGERS LIKE "account%";   # Tabellename
  INSERT INTO account VALUES
    (1,   14.98),
    (2, 1937.50),
    (3, -100.00);
```

```
UPDATE account SET amount = amount * 1.19;
SELECT @old, @sum AS "Gesamt";
SET @sum = 0;
SET @old = 0;
UPDATE account SET amount = amount * 2.0 WHERE acct_num = 2;
SELECT @old, @sum AS "Gesamt";

#-------------------------------------------------------------------------
# Trigger 1 (2 Spalten immer in 1. Zeichen GROSS, Rest klein umwandeln)
# CONSTRAINT-Ersatz!
#-------------------------------------------------------------------------
DROP TRIGGER IF EXISTS t1;

CREATE TRIGGER t1 BEFORE INSERT ON pers   # Für UPDATE gl. Trigger definieren
FOR EACH ROW
BEGIN
  SET NEW.vorname = CONCAT(UCASE(LEFT(NEW.vorname, 1)),
                           LCASE(SUBSTR(NEW.vorname, 2)))
  SET NEW.name    = CONCAT(UCASE(LEFT(NEW.name, 1)),
                           LCASE(SUBSTR(NEW.name, 2)))
END //

#-------------------------------------------------------------------------
# Trigger 2 (Preis hat minimal Wert 0 und maximal Wert 100)
# CONSTRAINT-Ersatz!
#-------------------------------------------------------------------------
DROP TRIGGER IF EXISTS t2;

CREATE TRIGGER t2 BEFORE INSERT ON artikel   # Für UPDATE gl. Trigger definieren
FOR EACH ROW
BEGIN
  IF NEW.preis < 0 THEN
    SET NEW.preis = 0;
  ELSEIF NEW.preis > 100 THEN
    SET NEW.preis = 100;
  END IF;
END //

#-------------------------------------------------------------------------
# Trigger 3 (Alter in Jahren aus Akt. Datum - Geburtsdatum ausrechnen)
#-------------------------------------------------------------------------
DROP TRIGGER IF EXISTS t3;

CREATE TRIGGER t3 BEFORE INSERT ON age
FOR EACH ROW
BEGIN
  SET NEW.jahre = YEAR(CURDATE()) - YEAR(NEW.geburtsdatum);
END //

#-------------------------------------------------------------------------
# Trigger 4 (Bestellsumme aus Preis * Anzahl - Rabatt ausrechnen)
#-------------------------------------------------------------------------
DROP TRIGGER IF EXISTS t4;

CREATE TRIGGER t4 BEFORE UPDATE ON bestellung
FOR EACH ROW
BEGIN
  SET NEW.summe = NEW.anz * NEW.preis * (100.0 - NEW.rabatt) / 100.0;
END //

#-------------------------------------------------------------------------
# Trigger 5 (alte Spaltenwerte vor Löschen der Datensätze in Var. merken)
#-------------------------------------------------------------------------
DROP TRIGGER IF EXISTS t5;
```

```
  CREATE TRIGGER t5 AFTER DELETE ON pers
  FOR EACH ROW
  BEGIN
    SET @vorname = OLD.vorname;    # Bei jeder Löschoperation überschrieben
    SET @name    = OLD.name;       # Nach Tabellen löschen noch verfügbar
    SET @plz     = OLD.plz;
    SET @ort     = OLD.ort;
  END //

  #-------------------------------------------------------------------------
  # Trigger 6 (Summe aller in eine Tabelle eingefügten Werte bilden)
  #-------------------------------------------------------------------------
  DROP TABLE IF EXISTS bestellung;

  CREATE TABLE bestellung (
    nr     INT,
    anz    INT,
    preis  FLOAT,
    summe  FLOAT,
    rabatt FLOAT
  );

  DROP TRIGGER IF EXISTS t6;

  CREATE TRIGGER t6 BEFORE INSERT ON bestellung
  FOR EACH ROW
  BEGIN
    # Bei jeder Einfügeoperation überschrieben
    SET @total = @total + NEW.anz * NEW.preis * (100.0 - NEW.rabatt) / 100.0;
  END //

  SET @total = 0;
  INSERT INTO bestellung VALUES (1, 1, 65.00, 1 * 65.00, 0),
                                (2, 5, 10.99, 5 * 10.90, 2),
                                (3, 2,  5.49, 2 *  5.49, 2);
  SELECT @total AS "Gesamtsumme";

  #-------------------------------------------------------------------------
  # Trigger 7 (Daten in andere Tabellen weitergeben)
  #-------------------------------------------------------------------------
  DROP TRIGGER IF EXISTS t7;

  CREATE TRIGGER t7 BEFORE INSERT ON test1
  FOR EACH ROW
  BEGIN
    INSERT INTO test2 SET a2 = NEW.a1;
    DELETE FROM test3                   WHERE a3 = NEW.a1;
    UPDATE      test4 SET sum = sum + 1 WHERE a4 = NEW.a4;
  END //

  # TODO

  DELIMITER ;
```

21) Condition Handler (Error/Warning)

Fehler- oder Warnungs-Situationen als Ergebnis-Status einer SQL-Anweisung erfordern gelegentlich eine besondere Reaktion. Mit "Condition Handlern" wird festgelegt, wie auf einen bestimmten SQL-Status reagiert werden soll. Insbesondere in Triggern, Routinen (Stored Procedures) und bei der Verwendung eines Cursors ist das häufig sinnvoll.

SQL-Fehler und -Warnungen werden durch einen 5-stelligen SQL-Status "XXXXX" (SQLSTATE: in SQL standardisiert) dargestellt (mit externem Programm "perror"

in entsprechende Textmeldung umwandeln). Ihm entspricht ein MySQL-Fehlercode (ERROR-Code: MySQL-spezifisch) aus dem er generiert wird. Die ersten beiden Zeichen des SQL-Status "XXXXX" legen die FEHLERKLASSE fest:

Klasse	Bedeutung	Beendet Programmfluss
00...	Erfolg	Nein
01...	Warnung	Nein
02...	Nicht gefunden	Ja
XX...	Fehler (sonstige Präfixe)	Ja

Beispiele:
* SQL-Status "00000" bzw. MySQL-Fehlercode 0 zeigen die ERFOLGREICHE Ausführung einer SQL-Anweisung an (nicht in Condition Handler verwenden!)
* SQL-Status "02000" bzw. MySQL-Fehlercode 1329 bedeutet "No Data" und tritt auf, wenn ein Cursor das Datenende erreicht bzw. eine SELECT...INTO <Var> Anweisung keine Ergebnisdaten produziert.

Mit einer Condition-Definition wird einem SQL-Status ein frei wählbarer Name <CondName> zugeordnet, der später in einem Condition Handler benutzt werden kann. Dies dient ausschließlich der Bequemlichkeit und Dokumentation, es ist auch möglich, den SQL-Status direkt im Condition Handler zu verwenden.

```
DECLARE <CondName> CONDITION FOR
  { SQLSTATE [VALUE] "XXXXX"        # SQL-Status "XXXXX"
  | <Zahl> };                       # MySQL-Fehlercode <Zahl>
```

Definition eines Condition Handlers für einen (oder mehrere) SQL-Status. Trifft eine der Bedingungen zu (tritt ein SQL-Status auf), dann wird die zugehörige einfache oder Verbund-Anweisung ausgeführt. Ein Handler bezieht sich immer auf den umgebenden BEGIN-END-Block, in dem er deklariert wurde. Tritt ein SQL-Status auf, für den kein Handler deklariert wurde, so ist EXIT die Standardaktion (verlassen des umgebenden BEGIN-END-Blocks).

```
# Variante A (EIN Statement)        # Variante B (MEHRERE Statements)
DECLARE <HandlerType> HANDLER       # DECLARE <HandlerType> HANDLER
  FOR <CondValue> {, <CondValue>}   #   FOR <CondValue> {, <CondValue>}
    <Stmt>;                         # BEGIN
                                    #   <Stmt>;
                                    #   ...
                                    # END;

                            # Umgebender Block (BEGIN ... END) ...
<HandlerType> = CONTINUE    # ... läuft nach Handler-Anweisung weiter
              | EXIT        # ... wird nach Handler-Anweisung verlassen (STD)
              | UNDO        # ... (nicht unterstützt)

<CondValue> = SQLSTATE [VALUE] "XXXXX"     # SQL-Status "XXXXX"
            | <Zahl>                       # MySQL-Fehlercode <Zahl>
            | <CondName>                   # Name einer vorher def. Condition
            | SQLWARNING                   # SQL-Status "01..."
            | NOT FOUND                    # SQL-Status "02..."
            | SQLEXCEPTION                 # Rest außer OK, "01...", "02..."
```

Zum Ignorieren eines SQL-Status einen leeren BEGIN-END-Block verwenden:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;   # Leerer BEGIN-END-Block
```

Typische MySQL-Fehlercodes / SQL-Stati sind durch ANSI SQL und ODBC standardisiert; MySQL-Fehlercodes sind nur für MySQL relevant (vollständige Liste --> 25a) MySQL-Fehlercode / SQL-Status):

Code	Status	Message (Bedeutung)

```
+------+---------+-----------------------------------------------------+
|    0 | "00000" | Erfolg (kein Fehler)                                |
+------+---------+-----------------------------------------------------+
| 1311 | "01000" | Referring to uninitialized variable %s              |
+------+---------+-----------------------------------------------------+
| 1329 | "02000" | No data -- zero rows fetched, selected, or processed |
+------+---------+-----------------------------------------------------+
| 1022 | "23000" | Can't write; duplicate key in table ...             |
| 1037 | "HY001" | Out of memory; restart server and try again         |
| 1038 | "HY001" | Out of sort memory; increase server sort buffer size |
| 1040 | "08004" | Too many connections                                |
| 1041 | "HY000" | Out of memory                                       |
| 1042 | "08S01" | Can't get hostname for your address                 |
| 1044 | "42000" | Access denied for user '...' to database '...'      |
| 1046 | "3D000" | No database selected                                |
| 1047 | "08S01" | Unknown command                                     |
+------+---------+-----------------------------------------------------+
|
+------+---------+-----------------------------------------------------+
```

Beispiel 1:

```
  DELIMITER //
  CREATE PROCEDURE HandlerDemo1()
  BEGIN
    DECLARE uninit INT;
    DECLARE dummy  CHAR(100);

    DECLARE CONTINUE HANDLER FOR SQLSTATE "01000" BEGIN
      SELECT "Uninitialisierte Variable!";
      SET @cnt := @cnt + 1;
    END;
    DECLARE CONTINUE HANDLER FOR SQLSTATE "02000" BEGIN
      SELECT "Keine Daten selektiert!";
      SET @x := 9;
    END;

    SET @x = 1; SELECT user FROM mysql.user WHERE TRUE  LIMIT 1 INTO dummy;
    SELECT @x, @cnt, dummy, uninit + 1;
    SET @x = 2; SELECT user FROM mysql.user WHERE FALSE LIMIT 1 INTO dummy;
    SELECT @x, @cnt, dummy, uninit + 1;
    SET @x = 3; SELECT user FROM mysql.user WHERE FALSE LIMIT 1 INTO dummy;
    SELECT @x, @cnt, dummy, uninit + 1;
    SET @x = 4; SELECT user FROM mysql.user WHERE TRUE  LIMIT 1 INTO dummy;
    SELECT @x, @cnt, dummy, uninit + 1;
  END //
  DELIMITER ;

  CALL HandlerDemo1();
  DROP PROCEDURE HandlerDemo1;
```

Beispiel 2:

```
  CREATE TABLE t (
    id INT PRIMARY KEY
  );

  DELIMITER //
  CREATE PROCEDURE HandlerDemo2()
  BEGIN
    DECLARE DuplicateKeyError CONDITION FOR SQLSTATE "23000";

    DECLARE CONTINUE HANDLER FOR DuplicateKeyError SET @err = "dup1";   # Proz.

    SET @x = 1;
    INSERT INTO t VALUES (1);
```

```
    SELECT @x, @err;
    SET @x = 2;
    INSERT INTO t VALUES (2);
    SELECT @x, @err;

    BEGIN
      DECLARE EXIT HANDLER FOR DuplicateKeyError SET @err = "dup2";   # Block
      SET @x = 3;
      INSERT INTO t VALUES (3);
      SELECT @x, @err;
      SET @x = 4;
      INSERT INTO t VALUES (1);   # Ups! ---+ EXIT HANDLER
      SELECT @x, @err;            #         |
      SET @x = 5;                 #         |
      INSERT INTO t VALUES (5);   #         |
      SELECT @x, @err;            #         |
    END;                          #         |
                                  #         |
    SET @x = 6;                   # <-------+
    INSERT INTO t VALUES (6);
    SELECT @x, @err;
    SET @x = 7;
    INSERT INTO t VALUES (1);     # Ups! ---+ CONTINUE HANDLER
    SELECT @x, @err;              # <-------+
    SET @x = 8;
    INSERT INTO t VALUES (8);
    SELECT @x, @err;
  END //
  DELIMITER ;

  CALL HandlerDemo2();
  DROP PROCEDURE HandlerDemo2;
  DROP TABLE t;

  SELECT @x, @err;
```

Beispiel 3 (Eine Handler-Anweisung kann kein ITERATE oder LEAVE mit dem Label eines die Handler-Anweisung umgebenden Blockes enthalten; über eine Status-Variable, die der Handler verändert, kann der umgebende Block trotzdem prüfen, ob der Handler ausgelöst wurde):

```
  # So geht es nicht!
  DELIMITER //
  CREATE PROCEDURE HandlerDemo3a()
  BEGIN
    DECLARE i INT DEFAULT 3;

    retry: REPEAT
      BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN ITERATE retry; END;
      END;
      SET i := i - 1;
      SELECT i, warn;
    UNTIL i < 0 END REPEAT;
  END //
  DELIMITER ;

  CALL HandlerDemo3a();
  DROP PROCEDURE HandlerDemo3a;

  # So ist es OK!
  DELIMITER //
  CREATE PROCEDURE HandlerDemo3b()
  BEGIN
    DECLARE i    INT  DEFAULT 3;
    DECLARE warn BOOL DEFAULT FALSE;
```

```
    retry: REPEAT
      BEGIN
        DECLARE CONTINUE HANDLER FOR SQLWARNING SET warn = TRUE;
      END;
      SET i := i - 1;
      SELECT i, warn;
    UNTIL warn OR i < 0 END REPEAT;
  END //
  DELIMITER ;

  CALL HandlerDemo3b();
  DROP PROCEDURE HandlerDemo3b;
```

22) Cursor (Zeiger)

Zum SEQUENTIELLEN Lesen (und evtl. Schreiben) von Datensätzen, d.h. das MENGENORIENTIERTE Verhalten von SQL wird umgangen. Sinnvoll z.B. in Stored Procedures und in Anwendungen mit sequentiellem Lese/Schreibverhalten (COBOL).

Einschränkungen:
- * Nur in Stored Routine, Trigger und Event erlaubt
- * Asensitive (Kopie der Ergebnistabelle möglich aber nicht unbedingt nötig)
- * Read-only (keine Updates möglich, MY!)
- * Forward-only
- * Nonscrollable (nur eine Leserichtung, keine Datensätze überspringbar, MY!)
- * Nonholdable (nach Commit geschlossen)
- * Namenslos (Handler = Cursor-ID)
- * Server-side (materialisierte temporäre Tabelle)   # Keine Client-side
- * Mehrere gleichzeitig öffenbar
- * Verschachtelbar

Syntax:

```
DECLARE <Cursor> CURSOR FOR <SelectStmt>;   # Cursor mit SQL-Anw. verknüpfen
OPEN <Cursor>;                              # Abfrage durchführen (Puffer)
FETCH <Cursor> {INTO | USING} <Var>, ...;   # EINEN Ergebnissatz in Var. holen
CLOSE <Cursor>;                             # Pufferinhalt verwerfen
```

Hinweise:
- + Führt vollständige Abfrage beim Öffnen des Cursors aus (evtl. LIMIT!)
  --> Temporäre Tabelle
- + Zielvariablen müssen zu Spalten passenden Datentyp haben
- + Prüfung per Handler über Statusvariable ob letzter Datensatz gelesen
- + Verknüpfung des Cursors bleibt bestehen (erneut öffenbar)
- + Deklarations-Reihenfolge: Erst Variablen, dann Cursor, dann Handler
- + Cursor muss nicht durch alle Datensätze laufen

HANDLER zur Ende-Erkennung notwendig (SQLSTATE 02000 bzw. NOT FOUND):

```
DECLARE done INT DEFAULT 0;                                 # A) Var. initial.
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;        # A) Var. setzen
DECLARE CONTINUE HANDLER FOR SQLSTATE "02000" BEGIN END;    # B) Ignorieren
```

Beispiel:

```
CREATE TABLE IF NOT EXISTS pers (
  nr      INT NOT NULL,
  vorname VARCHAR(30),
  name    VARCHAR(30)
);

# Tabelle Personen
#CREATE TABLE copy SELECT * FROM pers;   # +Struktur, -Indices, +Daten
CREATE TABLE copy LIKE pers;             # +Struktur, +Indices, -Daten
```

```
  INSERT INTO  copy SELECT * FROM pers;    # -Struktur, -Indices, +Daten
  CREATE TABLE res  LIKE pers;             # +Struktur, +Indices, -Daten

  DELIMITER //

  CREATE PROCEDURE cursor_demo()
  BEGIN
    DECLARE done     INT DEFAULT 0;
    DECLARE n1, n2   INT;
    DECLARE vn1, nn1 CHAR(20);
    DECLARE vn2, nn2 CHAR(20);

    DECLARE cur1 CURSOR FOR SELECT nr, vorname, name FROM pers;
    DECLARE cur2 CURSOR FOR SELECT nr, vorname, name FROM copy;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

    OPEN cur1;
    OPEN cur2;

    REPEAT
      FETCH cur1 INTO n1, vn1, nn1;
      FETCH cur2 INTO n2, vn2, nn2;
      IF NOT done THEN
        IF n1 < n2 THEN
          INSERT INTO res VALUES (n1, CONCAT("A", vn2), n2);
        ELSE
          INSERT INTO res VALUES (n2, CONCAT("B", vn2), n2);
        END IF;
      END IF;
    UNTIL done END REPEAT;

    CLOSE cur1;
    CLOSE cur2;
  END //

  DELIMITER ;

  CALL cursor_demo;
  SELECT * FROM res;
  DROP PROCEDURE cursor_demo;
  TRUNCATE TABLE res;
```

23) Table Handler

Zum DIREKTEN Lesen von den in einer Storage-Engine gespeicherten Datensätzen EINER Tabelle (direkter Zugriff auf Storage Engine Interface analog ISAM ohne Umweg über SQL). Verhält sich wie SELECT mit folgenden Unterschieden:

* Nur EINE Tabelle zugreifbar
* Es wird immer auf ALLE Spalten der Tabelle zugegriffen
* Sortierung der Datensätze gemäß physischer Speicherung oder EINEM Index
* Positionierung in Datensätzen möglich

Eigenschaften:

* Zugriff auf EINE Tabelle gleichzeitig
* Direkter Zugriff auf Storage Engine Interface (analog ISAM)
* Verfügbar für MyISAM- und InnoDB-Engine
* Pro Session getrennt
* Holt Datensätze in "natürlicher" Reihenfolge (wie gespeichert)
* Zum ersten/letzten Datensatz springen und rückwärts durchlaufen möglich
* Tabelle NICHT gesperrt zwischen zwei Handleraufrufen
  --> Lesekonsistenz wird nicht erzwungen (z.B. dirty read möglich)
* Kein Schreibzugriff (read only)

Gründe für die Nutzung:
  * Schneller "Full Table Scan" = Zugriff auf ALLE Spalten EINER Tabelle
  * Schneller und flexibler als SELECT (solange nur Zugriff auf eine Tabelle)
    + Weniger SQL-Anweisungs-Parsing notwendig
    + Kein Overhead für Optimierer und Query-Prüfung
  * Zur Portierung von Anwendungen, die low-level ISAM-Zugriffe einsetzen
    + Durchlaufen mehrerer/aller Datensätze einfacher als mit SELECT
      ("Full Table Scan", z.B. für GUI-Anwendungen)

Tabelle öffnen für Lesezugriffe (Alias statt Tabellenname verwendbar):

```
  HANDLER <Tbl> OPEN [[AS] <Alias>];
```

Hinweise:
  * EINZELNEN Datensatz aus EINER Tabelle (Zugriff über EINEN Index) einlesen
  * Datensatz erfüllt WHERE-Bedingung
  * Statt einem bis zu LIMIT Datensätze einlesbar

A) ALLE Datensätze in "natürlicher" Reihenfolge (wie gespeichert) durchlaufen:

```
 HANDLER <Tbl> READ {FIRST | NEXT}
   [WHERE <Cond>]
   [LIMIT ...];
```

B) ALLE Datensätze in Index-Reihenfolge (wie sortiert) durchlaufen (Name
  `PRIMARY` (Backquotes!) verwenden, um den PRIMARY KEY für den Zugriff zu
  benutzen):

```
 HANDLER <Tbl> READ <Idx> {FIRST | NEXT | PREV | LAST}
   [WHERE <Cond>]
   [LIMIT ...];
```

C) NUR Datensätze ab Indextreffer in Index-Reihenfolge (wie sortiert)
   durchlaufen. Anzahl in (...) übergebener Werte <Vali> muss kleiner gleich
   Anzahl Index-Komponenten sein (leftmost part). Die Werte müssen passend zur
   Reihenfolge der Index-Komponenten sein:

```
 HANDLER <Tbl> READ <Idx> {= | < | <= | > | >=} (<Val1>, ...)
   [WHERE <Cond>]
   [LIMIT ...];
```

Handler schließen (erfolgt auch automatisch am Sessionende):

```
  HANDLER <Tbl/Alias> CLOSE;
```

Beispiel:

```
  HANDLER pers OPEN AS h_pers;

  HANDLER h_pers READ idx2 = ("Bayer");
  HANDLER h_pers READ idx2 > ("Bayer", "Heinz");
  HANDLER h_pers READ idx2 < ("Bayer");
  HANDLER h_pers READ idx2 <= ("Bayer");
  HANDLER h_pers READ idx2 >= ("Bayer") LIMIT 2;
  HANDLER h_pers READ idx2 >= ("Bayer") WHERE nr >= 9;

  HANDLER h_pers READ `PRIMARY` FIRST;
  HANDLER h_pers READ `PRIMARY` NEXT;
  HANDLER h_pers READ `PRIMARY` NEXT LIMIT 2;

  HANDLER h_pers READ `PRIMARY` LAST;
  HANDLER h_pers READ `PRIMARY` PREV;
  HANDLER h_pers READ `PRIMARY` PREV LIMIT 3;

  HANDLER h_pers READ FIRST;
  HANDLER h_pers READ NEXT;
```

```
  HANDLER h_pers READ NEXT LIMIT 3;

  HANDLER h_pers CLOSE;
```

24) Events (Ereignisse)

MySQL besitzt ab MY!5.1 einen SCHEDULER, der benutzergesteuert zu bestimmten Zeitpunkten (Events) EINMALIG oder WIEDERHOLT Aufgaben (Tasks) in Form von SQL-Anweisungen durchführt (analog Linux "crontab"/"cronjob").

Zweck:
- * Periodische DB-Verwaltungstätigkeiten
  - + Tabellen-Statistik periodisch aktualisieren
  - + Protokollierung des DB-Zustandes
  - + Backup oder Synchronisation/Replikation auslösen
  - + Datenbank-Pflege auslösen
- * Periodische Generierung von Tabellen mit Zwischenergebnissen (Top10)
  --> "Materialized" Views

Eigenschaften:
- * Ab MY!5.1 verfügbar
- * Event-Name muss pro Datenbank <Db> eindeutig sein
- * Einmalig (AT) oder wiederholt (EVERY) möglich (<Schedule>)
- * EVERY wiederholt im Zeitraum STARTS bis ENDS
  - + Keine STARTS-Angabe --> Definitionszeitpunkt = CURRENT_TIMESTAMP()
  - + Keine ENDS-Angabe --> Gilt ewig
- * Event automatisch löschbar, wenn erledigt (Zeitraum verstrichen)
  - + ON COMPLETION NOT PRESERVE
- * Events haben die gleichen Einschränkungen wie Stored Procedures
  - + Kann keinen Trigger / Stored Procedure / Event erzeugen
- * Wählbar ob repliziert oder nicht (DISABLE ON SLAVE)
- * Erneutes Auslösen während Durchlauf von MySQL nicht verhindert
  (manuell Mutex/Lock in Event-Aktion --> GET/IS_FREE/IS_USED/RELEASE_LOCK)
- * Mehrere können gleichzeitig laufen (CONNECTION_ID() ist eindeutig)
- * Kein SQL-Standard
- * Übersicht: INFORMATION_SCHEMA.EVENTS

Syntax:

```
  CREATE
    [DEFINER = {<User> | CURRENT_USER}]         # Ersteller
    EVENT [IF NOT EXISTS] [<Db>.]<Event>        # Event-Name (DB-eindeutig)
      ON SCHEDULE <Schedule>                    # AT/EVERY ...
        [ON COMPLETION [NOT] PRESERVE]          # Autom. löschen bei Abschluss
        [ENABLE | DISABLE | DISABLE ON SLAVE]   # Aktivieren/Deaktivieren
        [COMMENT <String>]                      # Kommentar
      DO <Stmt>;                                # Oder DO BEGIN <Stmt>;... END;

  <Schedule> = AT <Timestamp> [+ INTERVAL <N> <Interval>]
             | EVERY <Interval> [STARTS <Timestamp>] [ENDS <Timestamp>]

  <Timestamp> = CURRENT_TIMESTAMP()     # Aktueller Zeitpunkt
              | "YYYY-MM-DD hh:mm:ss"   # Datum + Uhrzeit
              | YYYYMMDDhhmmss          # Datum + Uhrzeit

  <Interval> = YEAR | QUARTER | MONTH | WEEK | DAY
             | HOUR | MINUTE | SECOND
             | YEAR_MONTH
             | DAY_HOUR | DAY_MINUTE | DAY_SECOND
             | HOUR_MINUTE | HOUR_SECOND
             | MINUTE_SECOND
```

Beispiele für Intervalle (A=Auflösung: M=Monat, S=Sekunde):

```
  +---------------------------+---+---------------------------------------+
```

```
  |          Intervall          | A | Bedeutung                                |
  +-----------------------------+---+------------------------------------------+
  |             1 YEAR          | M | 1x pro Jahr                              |
  |             2 QUARTER       | M | Alle 2 Quartale                          |
  |             3 MONTH         | M | Alle 3 Monate                            |
  |             5 WEEK          | S | Alle 5 Wochen                            |
  |             3 DAY           | S | Alle 3 Tage                              |
  |             2 HOUR          | S | Alle 2 Stunden                           |
  |            15 MINUTE        | S | Alle 15 Minuten                          |
  |            45 SECOND        | S | Alle 45 Sekunden                         |
  +-----------------------------+---+------------------------------------------+
  |         "2-2" YEAR_MONTH    | M | Alle 2 Jahre und 2 Monate                |
  |       "10 05" DAY_HOUR      | S | Alle 10 Tage und 5 STD                   |
  |    "10 02:10" DAY_MINUTE    | S | Alle 10 Tage, 2 STD und 10 Min           |
  | "10 02:10:30" DAY_SECOND    | S | Alle 10 Tage, 2 STD, 10 Min und 30 Sek   |
  |       "10:05" HOUR_MINUTE   | S | Alle 10 STD und 5 Min                    |
  |    "10:05:30" HOUR_SECOND   | S | Alle 10 STD, 5 Min und 30 Sek            |
  |       "10:30" MINUTE_SECOND | S | Alle 10 Min und 30 Sek                   |
  +-----------------------------+---+------------------------------------------+

Übliche Praxis: Komplexe Aufgabe als Stored Procedure schreiben +
                per CALL in Event aufrufen
                --> Testen einfacher
                --> Manueller Aufruf jederzeit möglich

Beispiel (Stored Procedure "optimize_tables()"):

  DROP EVENT IF EXISTS optimize_db;
  CREATE EVENT optimize_db
    ON SCHEDULE
      EVERY 1 WEEK
    DO
      CALL optimize_tables('db');

Erneutes Auslösen während langem Durchlauf verhindern:

  DROP EVENT IF EXISTS optimize_db;
  CREATE EVENT optimize_db
    ON SCHEDULE
      EVERY 1 WEEK
  BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION   # "dummy" continue handler
      BEGIN /*empty*/ END;
    IF GET_LOCK('optimize_db', 0) THEN
      DO CALL optimize_tables('db');
    END IF;
    DO RELEASE_LOCK('optimize_db');
  END

Beispiel (in einer Stunde 1x UPDATEn --- AT):

  DROP EVENT IF EXISTS e_next_hour;
  CREATE EVENT e_next_hour
    ON SCHEDULE
      AT CURRENT_TIMESTAMP() + INTERVAL 1 HOUR
    DO
      UPDATE myschema.mytable SET mycol = mycol + 1;

Beispiel (stündlich die Sitzungstabelle löschen --- EVERY):

  DROP EVENT IF EXISTS e_next_hour;
  CREATE EVENT e_hourly_sess_delete
    ON SCHEDULE
      EVERY 1 HOUR
      COMMENT "Stündlich die Sitzungstabelle löschen"
    DO
```

```
      DELETE FROM site_activity.sessions;
      # TRUNCATE site_activity.sessions;   # Evtl. nicht erlaubt
```

Beispiel (am nächsten Tag eine Stored Procedure aufrufen --- AT):

```
  DROP EVENT IF EXISTS e_call_myproc;
  CREATE EVENT e_call_myproc
    ON SCHEDULE
      AT CURRENT_TIMESTAMP() + INTERVAL 1 DAY
    DO
      CALL myproc(5, 27);
```

Beispiel (alle 5 Sekunden eine Anweisungsfolge durchführen --- EVERY):

```
  DELIMITER //
  DROP EVENT IF EXISTS e_every_5_sec //
  CREATE EVENT e_every_5_sec
    ON SCHEDULE
      EVERY 5 SECOND
    DO BEGIN
      DECLARE CONTINUE HANDLER FOR SQLEXCEPTION BEGIN END;
      INSERT INTO t1 VALUES (0);
      UPDATE t2 SET s1 = s1 + 1;
    END //
  DELIMITER ;
```

Beispiel (täglich Anzahl Sitzungen summieren und Sitzungstabelle löschen --- EVERY):

```
  DELIMITER //
  DROP EVENT IF EXISTS e_daily_aggregate_sess //
  CREATE EVENT e_daily_aggregate_sess
    ON SCHEDULE
      EVERY 1 DAY
      COMMENT "Summe Anzahl Sitzungen sichern und Sitzungstabelle löschen"
    DO BEGIN
      INSERT INTO site_activity.totals (when, total)
        SELECT CURRENT_TIMESTAMP(), COUNT(*)
          FROM site_activity.sessions;
      DELETE FROM site_activity.sessions;
    END //
  DELIMITER ;
```

Beispiel (zwei Tabellen wöchentlich um 1 Uhr Mitternacht optimieren --- EVERY):

```
  DELIMITER //
  DROP EVENT IF EXISTS e_opt_tables //
  CREATE EVENT e_opt_tables
    ON SCHEDULE
      EVERY 1 WEEK
      STARTS "2015-01-01 00:00:00"
      ENDS   "2015-12-31 00:00:00"
      ON COMPLETION NOT PRESERVE
    DO BEGIN
      OPTIMIZE TABLE test.t1;
      OPTIMIZE TABLE test.t2;
    END //
  DELIMITER ;
```

Beispiel (jeden Sonntag etwas durchführen --- EVERY):

```
  DELIMITER //
  DROP EVENT IF EXISTS e_every_sunday_do_something //
  CREATE EVENT e_every_sunday_do_something
    ON SCHEDULE
      -- Taeglich den Event ausfuehren
      EVERY 1 DAY
```

```
      -- Immer um 23:00 soll der Event stattfinden
      STARTS "2014-01-01 23:00:00"
    DO BEGIN
      -- Heute Sonntag (Wert 6)? --> Nur dann etwas durchfuehren!
      IF WEEKDAY(CURRENT_DATE()) = 6 THEN
        -- Eigentliche Taetigkeit
        OPTIMIZE TABLE test.t1;
        OPTIMIZE TABLE test.t2;
      END IF;
    END //
  DELIMITER ;
```

Beispiel (jeden Arbeitstag Mo-Fr etwas durchführen --- EVERY):

```
  DELIMITER //
  DROP EVENT IF EXISTS e_every_workday_do_something //
  CREATE EVENT e_every_workday_do_something
    ON SCHEDULE
      -- Taeglich den Event ausfuehren
      EVERY 1 DAY
      -- Immer um 23:30 soll der Event stattfinden
      STARTS "2014-01-01 23:30:00"
    DO BEGIN
      -- Heute Wochentag (Wert 0-4)? --> Nur dann etwas durchfuehren!
      IF WEEKDAY(CURRENT_DATE()) >= 0 AND WEEKDAY(CURRENT_DATE()) <= 4 THEN
        -- Eigentliche Taetigkeit
        -- Schleife ueber alle Tab. einer Db. (als Parameter übergeben)
        PREPARE ...
        EXEC
      END IF;
    END //
  DELIMITER ;
```

Zustand des Event-Schedulers anzeigen/ändern (ein Thread):

```
  SET GLOBAL event_scheduler = 1;     # 1=ON, 0=OFF/DISABLED
  SELECT @@event_scheduler;
  SHOW VARIABLES LIKE "event_scheduler%";
```

Liste aller/einiger Events mit Eigenschaften anzeigen:

```
  SHOW EVENTS;
  SHOW EVENTS LIKE "optimize_%";
```

Definition eines Events anzeigen:

```
  SHOW CREATE EVENT <Event>;
```

Event ändern (z.B. Ausführungszeitraum und -zeitpunkte ändern):

```
  ALTER
    [DEFINER = {<User> | CURRENT_USER}]
    EVENT <Event>
      [ON SCHEDULE <Schedule>]
        [ON COMPLETION [NOT] PRESERVE]
        [RENAME TO <NewEvent>]
        [ENABLE | DISABLE | DISABLE ON SLAVE]
        [COMMENT <String>]
        [DO <Stmt>];                # Oder DO BEGIN <Stmt>;... END;
```

Event deaktivieren/aktivieren:

```
  ALTER EVENT <Event> DISABLE;
  ALTER EVENT <Event> ENABLE;
```

Event umbenennen:

```
  ALTER EVENT <Event> RENAME TO <NewEvent>;

Event löschen:

  DROP EVENT <Event>;
  DROP EVENT IF EXISTS <Event>;   # Kein Fehler falls nicht existent

Recht vergeben, Events zu erstellen, ändern und löschen:

  GRANT EVENT ON <Db>.* TO <User>@<Host>;

Recht wegnehmen, Events zu erstellen, ändern und löschen:

  REVOKE EVENT ON <Db>.* TO <User>@<Host>;


25) Signale
-----------

Signale dienen zur Rückgabe eines Fehlerwertes an einen Handler, einen Benutzer
einer Anwendung oder einen Client (MY!5.5).
  * Legen folgende Fehler-Eigenschaften fest:
    + Fehler-Nummer
    + Status-Wert
    + Fehler-Nachricht
  * Signale sind nicht nur in Compound Statements, sondern überall verwendbar
    (z.B. in Event, Trigger, SQL-Anweisung: MY!-Erweiterung)
  * Signale können nur SQL-Status verwenden, keine MySQL-Fehlercodes
    + Vollständige Liste --> 25a) MySQL-Fehlercode / SQL-Status
  * Signal-Klassen:

  +-------+--------------------------+-----------------------+
  | Klasse| Bedeutung                | Beendet Programmfluss |
  +-------+--------------------------+-----------------------+
  | 00... | Erfolg                   |          Nein         |
  | 01... | Warnung                  |          Nein         |
  | 02... | Nicht gefunden           |           Ja          |
  | XX... | Fehler (sonstige Präfixe) |           Ja          |
  +-------+--------------------------+-----------------------+

Syntax:
  SIGNAL <CondValue>                       # Zurückzugebener Fehlerwert
    [SET <SignalInfo> {, <SignalInfo>}]

  <CondValue> = SQLSTATE [VALUE] "XXXXX"   # SQL-Status "XXXXX"
              | <CondName>                 # Name einer vorher def. Condition

  <SignalInfo> = <CondInfoItem> = <SimpelValueSpec>

  <CondInfoItem> = { CLASS_ORIGIN         #
                   | SUBCLASS_ORIGIN      #
                   | CONSTRAINT_CATALOG   #
                   | CONSTRAINT_SCHEMA    #
                   | CONSTRAINT_NAME      #
                   | CATALOG_NAME         #
                   | SCHEMA_NAME          #
                   | TABLE_NAME           #
                   | COLUMN_NAME          #
                   | CURSOR_NAME          #
                   | MESSAGE_TEXT         #
                   | MYSQL_ERRNO }        #

  <SimpleValueSpec> =

Beispiel:
```

```
  CREATE PROCEDURE p(err INT)
  BEGIN
    DECLARE user CONDITION FOR SQLSTATE "45000";
    IF err = 0 THEN
      SIGNAL SQLSTATE "01000";                    # Warnung --> Weiter
    ELSEIF err = 1 THEN
      SIGNAL SQLSTATE "45000"                     # Fehler --> Abbruch
        SET MESSAGE_TEXT = "An error occurred";
    ELSEIF err = 2 THEN
      SIGNAL user                                 # Fehler --> Abbruch
        SET MESSAGE_TEXT = "An error occurred";
    ELSE
      SIGNAL SQLSTATE "01000"                     # Warnung --> Weiter
        SET MESSAGE_TEXT = "A warning occurred",
            MYSQL_ERRNO  = 1000;
      SIGNAL SQLSTATE "45000"                     # Fehler --> Abbruch
        SET MESSAGE_TEXT = "An error occurred",
            MYSQL_ERRNO  = 1001;
    END IF;
  END;

  SIGNAL SQLSTATE "77777";
  CREATE TRIGGER t_bi BEFORE INSERT ON t
    FOR EACH ROW SIGNAL SQLSTATE "77777";
  CREATE EVENT e ON SCHEDULE EVERY 1 SECOND
    DO SIGNAL SQLSTATE "77777";

  CREATE PROCEDURE p (divisor INT)
  BEGIN
    DECLARE divide_by_zero CONDITION FOR SQLSTATE "22012";
    IF divisor = 0 THEN
      SIGNAL divide_by_zero;
    END IF;
  END;
```

25a) MySQL-Fehlercode / SQL-Status

SQL-Fehler und -Warnungen werden durch einen 5-stelligen SQL-Status "XXXXX" dargestellt (per externem Programm "perror" in entsprechende Textmeldung umwandelbar). Ihm entspricht ein MySQL-Fehlercode, aus dem er generiert wird. Die ersten beiden Zeichen des SQL-Status "XXXXX" legen die FEHLERKLASSE fest:

Klasse	Bedeutung	Beendet Programmfluss
00...	Erfolg	Nein
01...	Warnung	Nein
02...	Nicht gefunden	Ja
NN...	Fehler (sonstige Präfixe)	Ja

Beispiele:
- SQL-Status "00000" bzw. MySQL-Fehlercode 0 zeigen die ERFOLGREICHE Ausführung einer SQL-Anweisung an (nicht in Condition Handler verwenden!)
- SQL-Status "02000" bzw. MySQL-Fehlercode 1329 bedeutet "No Data" und tritt auf, wenn ein Cursor das Datenende erreicht bzw. eine SELECT...INTO <Var> Anweisung keine Ergebnisdaten produziert.

Mit einer "Condition-Definition" kann einem SQL-Status ein frei wählbarer Name <CondName> zugeordnet werden, der später in einem "Condition Handler" genutzt werden kann. Dies dient ausschließlich der Bequemlichkeit und Dokumentation, es ist auch möglich, den SQL-Status direkt im Condition Handler zu verwenden.

```
  DECLARE <CondName> CONDITION FOR
    { SQLSTATE [VALUE] "XXXXX"      # SQL-Status "XXXXX"
    | <Zahl> };                     # MySQL-Fehlercode <Zahl>
```

Mit "Condition Handlern" wird festgelegt, wie auf bestimmte SQL-Status reagiert werden soll. Zum Ignorieren eines SQL-Status einen leeren BEGIN-END-Block verwenden:

```
  DECLARE CONTINUE HANDLER FOR SQLWARNING BEGIN END;   # Leerer BEGIN-END-Block
```

Signale dienen zur Rückgabe eines Fehlerwertes an einen Handler, einen Benutzer einer Anwendung oder einen Client (MY!5.5). Sie können nur einen SQL-Status verwenden, keine MySQL-Fehlercodes

Zu finden in folgenden Dateien:
* SQL-Status (5-stellige Nummern + Namen), durch ANSI SQL und ODBC standardisiert
  --> "include/sql_state.h"
* MySQL-Fehlercode (mapped auf SQL-Status)
  --> "include/mysqld_error.h"
* Messages
  --> "mysql/share/german/errmsg.txt"
  --> "mysql/share/english/errmsg.txt"

26) Begrenzungen von MySQL (Limits)

MySQL kennt folgende Begrenzungen (teilweise Engine-abhängig):

Limit	Bedeutung
65535	Byte pro Datensatz (InnoDB: 8000)
4096	Spalten pro Tabelle (InnoDB: 1000)
4 Mrd	Datenbanken (InnoDB)
4 Mrd	Tabellen pro Datenbank (InnoDB)
64 TB	Tablespace-Größe (InnoDB)
64 TB	Tabellen-Größe (InnoDB)
4 Mrd	Datensätze pro Tabelle (32 Bit Zeiger, alt)
2x10^19	Datensätze pro Tabelle (64 Bit Zeiger, neu)
64	Indices pro Tabelle (bei Neukompilierung bis zu 128)
16	Spalten zu "Composite Index" kombinierbar (mind.)
1000	Byte Index-Länge (InnoDB: ersten 767)
61	Tabellen pro View
61	Tabellen pro Join
8192	Partitionen pro Tabelle (1024 bis MY!5.6.6)
1 GB	Statementlänge (Protokoll, "--max-allowed-packet")
4 GB	Kommentarlänge
16384	Verbindungen ("--max_connections", STD: 151)

Maximale Länge von Bezeichnern (siehe 6.2.2 Privilege System Grant Tables):

Max	Bezeichner	
64	Datenbank	
64	Tabelle	
64	Spalte	
64	Routine	
64	Alias	# Vor MY!5.6 255 Zeichen erlaubt

```
|  60 | Host       |
|  16 | Benutzer   |
|  41 | Passwort   |
+-----+------------+
```

Begrenzungen der InnoDB-Engine:

```
+------+-----------------------------------------------+
| Max. | Bedeutung                                     |
+------+-----------------------------------------------+
| 8000 | Byte pro Datensatz (etwa halbe Speicherseite) |
| 1000 | Spalten pro Tabelle                           |
|  767 | Byte Index-Länge                              |
+------+-----------------------------------------------+
```

27) MySQL testen

Zum Testen eines MySQL-Server oder seiner Client-API stehen einige Programme zur Verfügung:

```
+---------------------------+-----------------------------------------------+
| Programm                  | Beschreibung                                  |
+---------------------------+-----------------------------------------------+
| mysqlslap                 | Server-Lasttest per Simulation vieler Clients |
| mysql-test-run.pl         | Server-Test per "mysqltest" steuern (Skript)  |
| mysqltest                 | Server-Test und Vergleich mit anderen Ergeb.  |
| mysqltest_embedded        | Server-Test und Vergleich mit anderen Ergeb.  |
| mysql_client_test         | Test der MySQL-Client-API (Skript)            |
| mysql_client_test_embedded | Test der MySQL-Client-API (Skript)           |
+---------------------------+-----------------------------------------------+
```

Ein typischer Datenbank-Lasttest des MySQL-Servers per "mysqlslap" (mögliche Werte von "sql-load-type": read, write, key, update, mixed):

```
mysqlslap -utom -pgeheim \
          --engine=myisam \
          --iterations=200 \
          --concurrency=20 \
          --number-char-cols=10 \
          --number-int-cols=4 \
          --detach=10000 \
          --auto-generate-sql \
          --auto-generate-sql-load-type=mixed \
          --debug-check \
          --debug-info \
          --verbose
```

==========